Jacqueline Beaulac

Department of Music
McGill University, Montreal

June 2000

Interactive multimedia composition on the World Wide Web: a solution for musicians using Java.

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfilment of the requirements of the degree of Master's of Arts.

0-612-70580-3

Abstract

This thesis attempts to gauge the strengths and limitations of the Java programming language in terms of its use in the production of multimedia compositions: in particular, the ways in which Java supports the creation of interactive, non-deterministic musical works. An original solution to the problem of multimedia design is presented: a hierarchically defined, basic, yet flexible scripting language that is interpreted using Java. This scripting language allows the user to incorporate his/her own media into a coherent and interactive form using a small set of simple keywords and basic operators. It also allows new functionality to be added by advanced users with a basic knowledge of Java. By investigating how such a scripting language may be implemented, the extent to which Java may be applied towards multimedia applications in general is revealed.

Précis

Cette thèse met en évidence les forces et les faiblesses du langage de programmation Java à propos de son utilité dans la production d'oeuvres multimedia ayant une composante intéractive ou non-déterministe. Un nouveau outil créé dans ce but est ainsi présenté: un langage simple mais extensible, permettant de définir de façon hiérarchique les oeuvres de média mixte. Les oeuvres ainsi specifiées seront réalisées à partir d'une application Java, et pourront incorporer divers médias fournis par le compositeur. Les utilisateurs plus avancés auront aussi la possibilité de construire des extensions de ce, en créant leurs propres algorithmes en Java. Par le biais d'une telle application, cette thèse montre un aperçu des possibilités offertes par Java dans ce domaine.

Table of Contents

Acknowledgements

I should like to acknowledge all those who contributed their knowledge, effort, and time to this work:

Prof. Bruce Pennycook, for providing me with guidance and helping to push me to the completion of my studies;

My employer, Ericsson Communications Inc., for providing me with the opportunity to learn the skills needed for my research, and with the means to finance my degree;

All the members of my team at Ericsson, for their technical assistance, their indispensable aid in advancing my programming skills, and most importantly, their kindness and support through the difficult times;

Thanks to Natalie Hasell and Katherine Rother, for their constant encouragement;

And all the thanks in the world to my husband Gilbert, for his love and endless patience.

Introduction

In this technological age, the arts are turning more and more towards multimedia as a new means of self-expression. The computer has become by far the most popular tool for integrating different types of creative endeavours—such as animations, musical compositions, artwork, and interactive games—into cross-media creative works. However, almost all of the programming languages used to define multimedia works are focussed on the same goal: the creation of static, wholly pre-defined productions. While some multimedia applications, such as Shockwave movies, allow a limited amount of user input into the finished work, the initial production of such a work requires a great deal of time and knowledge. Yet, for most people, the main rationale for using a computer as a tool in the creative process centres on its speed and ease of use. Thus, complex multimedia applications of this type actually end up stifling the same impulse to create which afforded them their original *raison d'être*.

Of the various means used to bring cross-media works to their audience, the World Wide Web is almost certainly the most popular. Despite this, no easy way to construct interactive or non-deterministic works for the Web has been developed yet. While some users have surmounted this obstacle, the majority of non-programmers have restricted themselves to printed-page-like Web productions. Even those Web pages that combine media often do so in a disconnected and static manner, employing animations, sounds, and images that are both uncoordinated and unresponsive to the observer. Because of this, the possibilities of this potentially ideal forum for the development of integrated media works have been, for the most part, left barely investigated.

The Java programming language is one of the most promising avenues of research for interactive multimedia. This thesis will attempt to gauge the strengths and limitations of Java in terms of its applicability to the production of multimedia compositions. In particular, it will examine the ways in which Java supports the creation of interactive multimedia works.

In order to do this, an original solution to the problem of easy-to-learn multimedia design will be presented: a hierarchically defined, basic, yet flexible scripting language that is interpreted using a Java application. This scripting language allows the user to incorporate his/her own musical cross-media works into a coherent and interactive form. It consists of a small set of simple keywords and basic operators that first-time users can learn quickly. However, it also allows new functionality to be added easily by more advanced users with a basic knowledge of Java. Rather than using an absolute time scale, as is common in most deterministic multimedia languages, scheduling of events is dependent on user input and on relative timing. This allows users with a musical background to create works that are not bound by the notion of frames, a concept designed for animation but far from ideal for music.

Having defined this scripting language, it will be shown that it is possible to implement it in Java. In examining this implementation, it will be ascertained whether Java is sufficiently powerful and flexible to be used to define the kind of structures necessary to create serious interactive multimedia works.

Chapter 1: Current multimedia formats and tools for the World Wide Web

1.1 Static file formats

The simplest way of presenting a musical composition on the Web is to capture an audio recording of the work and encode it in a format such as WAV, AIFF, or MP3[1]. Such an audio file can be posted on a Web page for downloading or, in the case of streaming audio formats such as RealAudio[2], can be made available for immediate listening. Evidently, this technique allows only simple playback, since the audio is pre-recorded. Similarly, for cross-media works, the most basic option available is a video format that allows playback only, such as AVI or MPEG[3]. In many cases, these options are adequate for presenting works made for a live audience without computer mediation, such as concert recordings, studio works, and videotapes of live performances. However, for a composer who wants to create works that can be manipulated and changed by a Web audience, such files are not sufficient. Likewise, such static formats cannot be used to define aleatoric or algorithmic compositions that may differ from one playback to the next.

[1] WAV is the standard audio file format for DOS/Microsoft Windows; likewise, AIFF is the standard audio file format for the Apple Macintosh OS. MP3 is the most popular of the audio file formats that use the MPEG standardised compression scheme.

[2] RealAudio is a file format developed by Progressive Networks specifically for streaming audio.

[3] AVI is the standard video file format for Microsoft Windows. MPEG is a standardised file format for compressed video.

1.2 Interactive file formats and their development environments

Other multimedia formats have been created which allow greater flexibility in the Web environment than a simple recording can offer. In particular, QuickTime and Shockwave both allow a composer to define certain user actions and algorithmic components, which can be used to alter the course of a multimedia composition.

1.2.1 QuickTime[4]

QuickTime is a library of fairly low-level code used to play back media files in a large number of formats. This code is accessed using an interface for C, Pascal, or Java. Thus, to use QuickTime directly entails quite a bit of programming. However, many QuickTime development tools are available which hide the details of the QuickTime code behind a friendlier interface. QuickTime movies are simply files that contain a set of QuickTime commands. As such, QuickTime movie files may contain media data in themselves, but may also refer to a number of other media files. In this case, the movie file becomes a means of organising and synchronising the data under its control. In fact, QuickTime movies may be linked, with one master movie controlling aspects of other movies in a hierarchical fashion. The QuickTime code also generates "events" in response to user input. These events can cause actions to occur within the movie:

[4] QuickTime is the standard video file format for the Apple Macintosh. Apple has also ported the QuickTime API to Microsoft Windows and developed browser plugins for QuickTime.

for example, passing the mouse over a particular area on the screen might cause a sound to play or a button to change colour.

1.2.2 Shockwave

Shockwave is a Web-based application developed by Macromedia and used for the playback of files authored in Macromedia Director. These files often incorporate a variety of media, usually in a compressed format. The media are synchronised to each other by means of a score. Shockwave movies also may contain scripts written in Lingo, the scripting language used in Director. Lingo is used not only to control many standard aspects of animation, but also to provide interactivity by modifying aspects of the movie based on user events.

1.2.3 Similarities of QuickTime and Shockwave

It is clear that both QuickTime and Shockwave are oriented towards animation, not music. For both, the structural organisation of the media is in terms of frames, which divide the passage of time into uniform segments. Such a division into completely undifferentiated blocks of time is far from conducive to most forms of musical expression. Also, in both of these multimedia formats, there is a heavy emphasis on "sprites": visual elements that contain rules for their motions on-screen. Audio, when it is mentioned, is viewed mainly in terms of effects or sounds that are "attached" to these visual elements; it is clearly subordinate to video and image. Clearly, an alternative is necessary for composers who would like to write serious musical works that incorporate other media.

The remaining chapters address this need. Chapter 2 explores the salient features of the Java programming language; Chapter 3 investigates the characteristics of a new scripting language for interactive multimedia; and finally, Chapter 4 describes the implementation of that language in Java.

Chapter 2: Salient features of the Java programming language

2.1 Object-oriented programming

Object-orientation is a relatively new concept in programming. In an object-oriented language like Java, the pieces of data needed by a program are grouped into sets called objects, which resemble *structs* in the C programming language. Each object belongs to a class that defines the type of data that can be contained in the object, with each separate piece of data being stored in a field (a variable associated with that class). The class also defines the operations (known as "methods") that can be used to manipulate that data.

For example, an object belonging to an "AudioClip" class might contain the following fields:

- an array of digital audio data;
- the sampling rate of the audio data;
- an indication of the encoding of the audio data (ex. AIFF, WAV, raw audio data);
- the duration in milliseconds of the sound;
- the name of a file used to retrieve and/or save the audio data.

The same object might have the following methods:

- play, which causes the audio data to be played to some audio device;
- setSamplingRate, which changes the sampling rate to be used in playback;
- setFileName, which sets the name of the audio file to be used;

- loadFile, which causes the data in the file to be loaded into memory;
- saveFile, which causes the data in memory to be written into the file.

A class usually contains all the code necessary to perform the operations described by its methods, but a particular form of class, called an "interface", does not. An interface is, in a way, a potential class; it states what a class should be able to do, without actually being able to do these things. Interfaces are very useful when defining several classes that share similar characteristics. For example, all of the different formats of audio file can be played; however, the details of how they are to be played may differ greatly from one to the next. Looking back at the above example, if AudioClip is defined as an interface, several classes could implement that interface, one for each audio format desired. Each of these would be played by calling its "play" method; this would allow all to be treated in the same manner, regardless of the details of their audio formats.

Certain operations that a class of objects may perform are independent of the data that may be in any particular instance. For example, a class may want to translate a given string into some other form of data, and may be able to do that without consulting its own internal data. To support this, classes in Java may have methods declared "static", which are independent of any instance.

There are several advantages to the object-oriented approach to programming. By storing as much as possible of the program data within objects, the program as a whole becomes more structured. Related data is grouped together, and can be copied and manipulated as one unit. A complex operation that involves several pieces of data no longer needs to involve several different variables and functions spread throughout the code, as was sometimes the case in earlier programming languages. Instead, the operation can be implemented within a single method that belongs to an object that contains all the data needed. This makes the code for that operation easier to recognise, because it is all localised within the method. Often, the code within the method can be considered separately from code outside its object. This is because the method only operates on data within its own object and on data passed explicitly to it. Each object can thus be written, tested, and changed independently of the surrounding code.

Musical structure can be expressed quite easily in terms of objects. At the lowest level, audio samples, MIDI notes, and notated pitches can all be represented as basic objects. Simple phrases, chords, and sequences can be defined as lists of these basic objects. Then, complex musical structures can be expressed using several of these intermediate lists organised in some fashion. Also, as noted above, many different formats of sound data can be treated as equivalent, because they can be operated on in the same way: playing the sound, for example, or changing volume, pitch, or tempo.

2.2 High-level structures included in the basic APIs

The standard Java API (Application Program Interface)[5] includes a number of objects that provide high-level functionality. The Vector class provides a replacement for the linked list; the hashtable is a ready-to-use object; I/O streams are abstracted so that manipulation of the underlying bits is usually not necessary. This inclusion of high-level structures is very different from the approach of more traditional programming languages such as C that concentrate on the direct manipulation of small chunks of data. Because Java includes such large and complex objects as part of its standard API, rather than relying on third-party libraries, it tends to be easier to learn and more rapidly coded. However, Java code is also more difficult to optimise, since access to the "bits and bytes" is not always provided. This could make it too slow for time-critical uses, such as multi-voice real-time digital audio synthesis. As well, if a particular type of high-level structure is not included, it may be very difficult to create. This is the case with MIDI data, for example, which until recently was impossible to manipulate in Java, because there was no abstraction for it in the Java API.

[5] API is the standard term for a library of functionality in programming. In this case, "interface" does not refer to a particular type of class (as defined in 2.1). Instead, it refers to the abstract concept of a class or function for which the definition is known to the programmers that will use it, but not necessarily the details of its internal code. An API in object-oriented programming presents definitions for a group of classes and the functionality that they provide while hiding their exact implementation.

2.3 Exception handling

Java includes its own mechanisms for error recovery that can eliminate a great deal of repetitive code. In languages like C, in each situation where there is a high probability that errors will occur, a separate mechanism must be built to deal with the errors, such as passing an error code as the return value from a function. In Java, any error can be handled by throwing an instance of the Exception class. An instance of this special class can carry information about an error up from one method to the method that called it. All exceptions can be handled in a uniform manner, regardless of the reason that they were created; or, if this is not appropriate, a particular class of exceptions can be dealt with specifically. For those exceptions that are most likely to occur, it is even possible to declare that the inclusion of code for dealing with the exception is mandatory at any place that the exception might occur.

The exception mechanism was especially interesting to me for two reasons. First, since multimedia requires a great deal of general I/O and file manipulation, both of which are very error-prone, the ability to deal with all such problems uniformly is very useful. Second, the data for a chunk of musical information, such as a phrase expressed in MIDI, can be quite complex, having many parameters. Even if one object is used to encapsulate all this data, checking for consistency or completeness may be quite problematic. The ability to throw an exception up through several levels of data manipulation in order to signal an error immediately is one that I used extensively in my implementation of the scripting language.

2.4 Platform-independence

For a computer to understand the code written by a human programmer, the text of the code must be parsed, or translated to simple instructions that the computer can execute, and then saved in a binary format. This process is known as compilation. In most of the programming languages in use today, compilation produces a file that can be executed on its own. No other utility is needed to run the compiled code. However, the trade-off is that the instructions in the binary file must be specific to a particular platform: that of the machine on which the program was compiled. This is because the simple instructions used in binary files are not consistent from one platform to another. As a result, only a machine of the same kind as the original machine can execute the program. In other words, the compilation is platform-dependent.

In Java, the compilation does not translate the code into instructions that are specific to the compiling computer. Instead, the instructions (called "bytecode") belong to a standard set specific to Java. Every Java compiler, regardless of its platform, is required to keep to the Java standard. These instructions are thus platform-independent. Execution of this compiled code requires a separate tool, called a "Virtual Machine" (VM). The VM reads the compiled files and interprets the bytecode in them. Then, as the program runs, each instruction is "interpreted", or in other words, translated by the VM into machine codes that are understandable to the computer on which it is being run. Thus, in Java, only the VM is platform-dependent.

For applications that run on the Internet, platform-independence is essential. Computers of all types are connected to the Internet. A program intended to run on the Internet, therefore, should run on as many different types of machines as possible. If, however, the compilation of the program is platform-dependent, several versions have to be provided: one for each type of machine that the programmer wants the program to run on. Java's platform-independent approach means that the programmer only needs to provide a single version of the program, since that version will run on any Java-compliant VM.

The scripting language is intended to provide musicians with another way to show off their multimedia works over the Internet. This implies that it should allow the widest distribution possible. Because of this, it was imperative to use a platform-independent solution.

2.5 Applets

Applets are a concept that has no direct analogy in older programming languages. An applet is a complete Java program, but one that can only run within a Java-compatible HTML browser. Because the applet is an actual program, it is possible to do many things within an applet that would not be possible when using a scripting language. However, applets are not easy to program. The applet must take its browser environment into account, particularly since the browser has the ability to control certain parts of the applet's own execution. This makes the design and implementation of this type of program very different from that of an application. In fact, the actual sequence of execution of the applet is partially determined by the browser, and not all browsers will use exactly the same plan of execution for the same applet. Because of this, one of the most fundamental parts of designing an applet is deciding which actions should be performed in which of the standard applet methods "init", "start", and "stop". A further restriction is the expectation that each of these methods should return quickly, so that the operation of the browser is not impaired by the execution of the applet. For this reason, any applet that performs tasks over a long period of time must do so in a separate thread, which executes in parallel with the browser and independently of the browser's direct control (see 2.8 for information on threads).

Because applets run within the context of a browser, and are generally considered "untrusted" code when loaded over the Internet, the passing of parameters to an applet is different than for an application. Since there is no command-line on which to give options, and few environment variables that can be accessed by an applet, parameters are passed in the HTML page that contains the applet. The syntax for each parameter is:

```
<PARAM NAME=name VALUE=string>
```

This syntax is fairly cumbersome: for short values, the extra text that must be typed is long and repetitive, while longer values can become unreadable because they cannot be split easily. There is no way to structure the parameters except through simple text formatting, and each parameter must have a unique name (array syntax is not supported). Because of this, I decided to limit as much as possible the amount of information that would have to be passed to the applet via its HTML page. This led me to define a scripting language of my own that is stored in separate text files, instead of simply generating HTML parameter code that could be used directly by the applet.

2.6 Support for images

In the 1.1 version of Java, images can be displayed easily, but they are not as easily manipulated. Drawing on-screen is also supported, but at a fairly basic level. However, new APIs are now available for image manipulation and for more complex 2D and 3D drawing that extend the functionality of standard Java.

2.7 Support for audio

In the 1.1 version of Java, which is by far the most widespread at the moment, support for audio is quite poor. The only means to present sound is to play an audio file encoded in a very specific format (AU, 8-bit, µlaw encoded, sampled at 8kHz). However, the new 1.2 version of Java supports playback of a much wider range of sound files. As well, the new Sound API allows manipulation of audio data at a fairly low level and provides MIDI I/O. In addition, third party APIs exist that provide enhancements to the audio support capabilities of standard Java, such as the QuickTime for Java API and the JMF implementation for RealSystem G2.

2.8 Multithreading

In many programming languages, events are expected to occur in a particular order, one after the other. Functions are called in a particular sequence, and each must finish its work before the next begins. In Java, this can be implemented by declaring a special method in an object. This method, called "main", can be executed as a sequential program, and is often the starting point of a basic Java application.

Then again, this does not have to be the case. Multithreaded programs allow several different operations to be performed at essentially the same time. For example, while one audio sample is being played, the next sample to be played can be loaded into memory from a file. Each thread has its own sequence of events, and none of its events is guaranteed to precede or follow those of any other thread that is running at the same time. Threads do not, however, exist independently, as each thread uses the objects of the program to which it belongs. As well, threads can interact with one another.

Java supports multithreading, and has many constructs that are helpful when programming concurrent behaviour. First among these is the Thread class, which encapsulates the data and methods needed to define and run a single thread. Each Thread class defines a sequence of actions that should be executed. Once a Thread object has been defined, it can be started at any point in a program. The thread can then be stopped by the thread that executes the main method, or by another thread; it may also simply end when its sequence of actions is finished. A Thread object can also be given a priority, which reflects how much of a share of the total processing power is allotted to that thread of execution. For the above example, the thread that plays the audio data should have a greater priority than the thread loading the next clip. This ensures that the playback of the audio clip will not be interrupted, causing "clicks" in the sound.

Since it is possible for several threads to access and change the same objects in a program simultaneously, an additional mechanism has been included in Java: thread synchronisation. An object that is synchronised has a "lock", which flags that object as being "in use" when a thread calls one of its methods. This ensures that only one thread can change a single object at a time. A second thread that would like to use that same object is forced to wait until the first finishes its changes and relinquishes the lock. Again using the first example, if one thread would like to play an audio sample, but another thread is loading the audio data for that sample, the playback thread would have to wait until loading had finished. Otherwise, without synchronisation, it is very possible that the playback thread might read only half the data, as not all the data had been loaded, or worse, might read "garbage" (uninitialised) data with unpredictable results.

With synchronisation, however, a new problem can occur: thread deadlock. This can occur if two threads each have a lock on one object, and each require the lock on the object that the other is holding. While usually this is a rare occurrence, it becomes more and more likely as more threads are used in a program.

The implementation of the scripting language is very strongly multithreaded. In order to reduce the perceived delay as multimedia data is being downloaded over the Internet, I use several threads to pre-load data some time before it must be played. Multithreading was an absolute necessity as well in the simultaneous playback of several elements. However, because of the large number of threads in use at any one time, I encountered many problems with thread deadlock. Finally, to avoid this, I limited my use of synchronisation very strictly, only including it at the lowest level of loading and presenting the audio and image data.

2.9 Dynamic loading

In most programming languages, in order to use a section of code from someone else, a programmer must know exactly what that section consists of, and can only integrate it into his/her own code by changing that code. In Java, however, there are mechanisms that can allow a program to load classes that were not known originally, i.e. that were neither part of the original code, nor integrated into it, but were added later to the program without changing the original code. This is called dynamic loading. For example, a composer might create a new Java class that will provide a stream of notes based on an algorithmic process and MIDI input. Probably the class would implement a known interface, such that its methods are known and standardised. The composer could then specify the name of his/her class to an existing Java application that would be able to provide MIDI to the class and to process the outgoing note stream, basing itself on the interface methods.

Because I wanted to make my scripting language extremely generic and extensible, I decided to exploit dynamic loading in my implementation of the scripting language. Three parts of the code in particular use this technique. First are the sound players and image viewers. Viewers for new media formats may be created and can be added without changing the main program. Second, there are the algorithms for ordering a set of objects, allowing composers to create their own classes to handle timing interactions between different media elements. Third are the triggers that handle user input, so that as composers desire new means of interactivity, they too can be added automatically.

2.10 Javadoc

Javadoc is a documentation tool that is integrated into the Java environment. The documents are generated using comments written in the Java source code that are notated using a special syntax. The javadoc utility then reads these comments and produces HTML files that contain not only the information in the comments, but their context as well. For example, a javadoc comment for a particular method would contain the programmer's notes, followed by the method declaration as it is coded. The javadoc generated for a class includes all its fields and methods, as well as an indication of the classes from which it inherits.

The generation of javadoc is very useful for groups of several complex classes, especially where there are many interfaces or several levels of inheritance. As well, javadoc is customarily used for the documentation of all libraries of functionality (APIs) written in Java. Furthermore, as a standardised tool, the documents it generates are highly consistent regardless of the environment or platform used to generate them. I found that using javadoc was extremely helpful in clarifying my implementation of the scripting language. In addition, it provided an easy way to produce documentation for my API in a standard format that should aid future programmers who may wish to build atop my work.

Chapter 3: A new scripting language for interactive multimedia

3.1 General design

In the scripting language, each section in a particular composition is defined as an object. Such an object can either contain a single piece of media, such as an image or audio sample, or it can contain a set of subsections. This allows the structure of a piece to be defined in terms of a tree structure, allowing both linear and multi-layered hierarchical structures.

Three pieces of data can be defined for every section:

- An ID that uniquely defines the section. All references to the section use this name.
- A stop trigger that will be used to determine when the section should end. If no trigger is defined, then the section will end when the applet is closed, or when a section higher in the tree ends.
- A driver that is responsible for playing the musical and visual events for this section. If the section contains a single piece of media, then the driver plays the audio and/or shows the visuals. If the section contains subsections, then the driver is responsible for determining the order in which the subsections are played, and for playing them in that order.

From this, it can be seen that sections in my compositional structure are begun and ended based on triggers rather than strict durations. In an interactive composition, unlike in a linearly determined musical piece, the duration of certain events may not be fixed. If the end of a musical event is triggered by some extra-musical occurrence, then there is no way to determine for how long that event will continue. Likewise, if an outside force is needed to trigger the beginning of a musical event, the duration of that event is not defined until the time of performance.

Because of this ambiguity, I decided not to define separately the concept of duration in my language. Instead, the end of each section of the sequence is defined in terms of a stop trigger. This trigger may be associated with a fixed duration, or it may be set off by something that the user does. In either case, when the condition for signalling the end of a musical event is met, a signal is sent to stop the event and start the next. This model can therefore handle events of fixed duration, events whose end is triggered by a user action, and events that are started by a user action.

Stop triggers can also be associated in pairs using a logical operator. Two triggers connected with an "and" operator will only cause the end of a section once the end conditions for both triggers have been met. If two triggers are connected with an "or" operator, the section will end when either end condition has been met. A pair of associated triggers can be treated like a single trigger in an association, and so triggers can be nested to any depth.

3.2 Syntax

In this scripting language, an object that contains a single media file is defined using the "basic" keyword, followed by an identifier that gives the format of the media file, and a unique ID string that can be used later to refer to this object. If any additional parameters are required to play the media file, these parameters can then be specified as a block of name-value pairs. The block of parameters is delimited by square brackets. After this, a trigger is specified that defines the conditions under which playback and/or display of the file should be stopped. A condition for stopping playback could be, for example, the passing of a particular length of time, or a key pressed by the user. The "stopTrigger" keyword starts the definition of the trigger, and is followed first by an identifier that determines the type of condition, and then by an obligatory block of parameters needed to evaluate the condition, delimited by square brackets. (This block may be empty, but the brackets must be present.) If no trigger is required, then the keyword "undefined" may be used; in this case only, the parameter block is not required. Finally, a semicolon terminates the object definition.

For example, the following script defines an object named "picture1" that contains an image file named "flw1.gif", along with the origin, height, and width at which it should be drawn. It also uses a stop trigger to specify that this image should be shown for two seconds.

```
basic jsb.basicimpl.Image picture1
[
     fileName  flw1.gif
     originX   0
     originY   0
     width     200
     height    300
]
stopTrigger jsb.basicimpl.Milliseconds
[
     ms        2000
]
;
```

Basic objects can be grouped together into sets. Furthermore, a set can contain other sets, in a hierarchical or tree-like fashion. The syntax for a set is very similar to that of a basic object. The "basic" keyword is omitted, so the definition of a set starts with its identifier and ID. In this case, rather than determining a media format, the identifier specifies the way in which the objects in this set should be organised at playback time. If any parameters are required to further specify this ordering, they are noted next, similar to the basic object. Then, a list of the IDs of the objects that belong to this set is given, placed within curly brackets. After that, as with the basic object, a stop trigger is indicated, and finally a semicolon ends the definition. For example, the following script specifies a set of objects that will be presented one after the other sequentially, with no stop condition defined.

```
jsb.basicimpl.Sequential imageSequence1
[
]
{
     picture1
     picture2
     picture3
}
stopTrigger undefined
;
```

One final aspect of the scripting language remains: the nesting of stop triggers. This allows multiple stop conditions to be chained together. In either a basic object or a set, a composite stop trigger may replace the identifier and parameters following the "stopTrigger" keyword. A composite stop trigger is bounded by parentheses and consists of a trigger identifier with its parameters, a logical operator that can be "&&" or "||", and a second trigger identifier with parameters. It is also valid to nest a composite stop trigger within another, following the same rules that it must be bounded by parentheses, and that it should take the place of a trigger identifier along with its parameters.

For example, this composite stop trigger would have its stop condition fulfilled after one second if the user pressed a key within that second, or after two seconds otherwise:

```
stopTrigger
   (jsb.basicimpl.Milliseconds
        [
             ms         2000
        ]
   ||
        (jsb.basicimpl.Milliseconds
             [
                  ms         1000
             ]
        &&
        jsb.basicimpl.KeyPress
             [
             ]
        )
   )
```

Chapter 4: Implementation of the scripting language in Java

4.1 The three basic data types

In order to convert the constructs of my scripting language into Java, I had to define three basic classes. For each object in the script, instances of these three classes must be created (with one exception, explained in the StopTrigger class).

4.1.1 SequenceObject

Each object in the script is stored in its own SequenceObject. All the data given in the script for an object is either stored within, or linked to, its particular SequenceObject instance. The ID is stored directly in the instance and is used as a lookup key: a means to find particular objects as they are needed at playback time. References are maintained within the SequenceObject to an instance of SequenceDriver and, in most cases, an instance of StopTrigger.

4.1.2 SequenceDriver

The main class used for playback, both for basic objects and for object sets, must implement the SequenceDriver interface. The name of the particular class is given in the script as the main identifier, preceding the ID. For a basic object, the SequenceDriver will be an instance of a class that is responsible for playing the particular format of media that the object uses. Often, this is a wrapper that does some thread handling and data manipulation, but passes the main responsibility for playing the media to a standard Java class, such as the java.applet.AudioClip class for audio files. For a set of objects, the SequenceDriver will be an instance of a class that contains an algorithm for organising and playing back several objects. For example, a SequenceDriver for a set might contain an algorithm for placing the objects it contains into a sequential order based on a set of Markov rules. In either case, the first set of parameters defined in the script, after the ID, belong to the SequenceDriver and will be used to determine its exact behaviour.

4.1.3 StopTrigger

A StopTrigger implements a condition that will be used to signal the end of playback for some object or set of objects. In the case of an "undefined" StopTrigger, no instance is created (the reference in the corresponding SequenceObject is null). This implies that playback may continue indefinitely. In all other cases, an instance of a class that implements this interface is required. Each StopTrigger implements an algorithm for monitoring a certain condition, such as a particular keystroke or the passing of a particular length of time. The name of the particular class is determined from the identifier that follows the "stopTrigger" keyword in the script. The parameters that follow are used by the StopTrigger to refine further the conditions that should lead to the stop of playback.

4.1.3.1 StopEvent and StopListener

When a StopTrigger determines that its stop condition has been met, it sends out a StopEvent. Only StopListeners can receive a StopEvent; this is why SequenceDrivers implement the StopListener interface, since they may have to listen for such an event. However, the trigger only dispatches its event to those StopListeners that have registered themselves with the trigger. Thus, usually, only the SequenceDriver that belongs to the same SequenceObject as a particular StopTrigger will register itself with that trigger and will be stopped when the trigger's stop condition becomes true.

4.2 Implementation of the tree structure

As was noted earlier, the scripting language contains the notion of sets, which may be collections of objects, of other sets, or both. This type of structure is best described in terms of a tree structure, where each basic object is a leaf node, containing only its own data, and each set is a branch node, with references to one or more leaves.

4.2.1 Leaf nodes

4.2.1.1 BasicSequenceObject

A BasicSequenceObject is simply a SequenceObject that uses a driver to play back its single piece of media. No extra functionality is built into this class, as none is needed. The BasicSequenceObject is defined as a separate class in order to ensure that instances of this class are treated strictly as leaf nodes, not simply as general SequenceObjects. Furthermore, it is expected that the driver used by an instance of this class can only be an ImageDriver or an AudioDriver.

4.2.1.2 ImageDriver and AudioDriver

These two abstract classes have no functionality built into them. However, a class that derives from one or the other is valid for leaf nodes only, as such a class is assumed to be solely for the playback of a particular media format. ImageDriver and AudioDriver are distinct, rather than being merged into a single class, in order to allow audio and image content to be distinguished.

4.2.2 Branch nodes

4.2.2.1 CompositeSequenceObject

A CompositeSequenceObject contains all the fields of the generic SequenceObject, but with additional data that defines the place of each instance in the tree structure. This extra information takes the form of a list of the IDs of all the children of the instance. These children may be any kind of SequenceObject; it is assumed that all may be treated in the same manner at playback time without causing problems. In addition to this child data, it should be noted that the driver for a CompositeSequenceObject must be a SetManager.

4.2.2.2 SetManager

Each class that derives from SetManager encapsulates an algorithm for determining and controlling the playback order of a set of child objects. The SetManager class itself defines playback in terms of a loop that is repeated until a stop signal is received. At each cycle of the loop, the ordering algorithm is requested to play the next child object in its progression. This may trigger the playback of one child, where the child is a leaf node and the algorithm is sequential in nature, or it may trigger a number of media, where the child is itself a set or the algorithm dictates simultaneous playback.

4.3 Enhancements for interactivity

4.3.1 CompositeStopTrigger and StopTriggerOp

A CompositeStopTrigger is simply a logical association between two stop triggers. Each instance contains references to its two child triggers, as well as a StopTriggerOp that represents an "OR" or "AND" relationship. If the relationship is an "OR", then the stop condition for the CompositeStopTrigger is met as soon as either of its children fires a StopEvent. If the CompositeStopTrigger is defined as an "AND" relationship, both children must have fired StopEvents for the stop condition of the parent trigger to be satisfied.

4.3.2 Subclasses of StopEvent

A generic StopEvent gives no information about the condition that triggered it. Yet in some situations, this type of information could be very useful. For example, if a StopTrigger is set off when the user presses a key, the driver that receives the StopEvent may want to react differently depending on the exact key that was pressed. To allow this sort of information to be passed, it is possible to create classes that derive from StopEvent but contain some extra information. Often, the subclass may be bound to a particular type of StopTrigger. For instance, a StopEvent that contains an ASCII character field is useful to a StopTrigger that reacts to user keystrokes, while another that stores co-ordinate information would be appropriate for a StopTrigger that responds to mouse clicks.

4.4 Displaying and storing the objects: SequenceFormat

While my initial Java code was only intended to be used to run my scripts, it became obvious that the data structures of the Java implementation could be far more powerful and flexible than the scripting language itself. This made me decide, in the end, to consider the final text script as a particular representation of the Java objects, rather than the other way around. The SequenceFormat interface embodies this change of direction. An instance of a SequenceFormat is responsible for looking up the data encapsulated in the various SequenceObjects that make up a composition and presenting that data in a different manner.

4.4.1 SequenceText

The original parser for the scripting language was the basis for the SequenceText class. In addition to reading script files and storing their data in SequenceObjects, this class has the capability to write scripts based on object data.

This class could be subclassed or modified to allow the Java code to support other text-based formats for multimedia presentations, such as the SMIL (Synchronized Multimedia Integration Language) standard (W3C, 1998).

4.4.2 SequenceGui

Initially this GUI class was meant as a troubleshooting tool that would allow me to see the contents of the SequenceObjects and compare them to the scripts generated from those objects. However, since I had heard several times that musicians do not enjoy typing long scripts, I decided to develop the SequenceGui class for use within a script building GUI that would alleviate this problem. As a result, this class became a full SequenceFormat, with the ability not only to display but also to store SequenceObject data.

4.5 Playing the sequence: the Runner applet

The applet that handles playback of SequenceObjects is fairly straightforward. It looks for four parameters in its enclosing HTML file:

- FILENAME, the name of the script file;
- STARTID, the ID of the SequenceObject that is at the top of the tree structure to be used for playback;
- BKCOLOR, the background colour of the applet (optional);
- FGCOLOR, the foreground colour of the applet (optional).

After this, the Runner applet traverses the tree of SequenceObjects, starting at the top node given by STARTID and linking each branch node to its children. Once this is done, loading and playback can be initiated on the top node, and both operations will be propagated through all the objects on which that node depends.

Results

After several false starts and much design work, I was able to implement the scripting language in Java, as described in Chapter 4. Additionally, in order to run scripts, test implementations for several abstract classes were added as follows[6]:

- The Image class is an implementation of ImageDriver that serves as a wrapper for the java.awt.Image class.
- The AudioClip class is an implementation of AudioDriver that serves as a wrapper for the java.applet.AudioClip class.
- Three classes extend the SetManager class: Parallel, Sequential, and Shuffle. Parallel plays all its child elements simultaneously, while the Sequential class plays one after the other in a predefined order. Shuffle reorders the children randomly and plays each one by one, recalculating the order once the whole set has been exhausted.
- Two pairs of classes are used to implement StopTriggers: KeyPress together with KeyPressedStopEvent are used to detect keystrokes from the user, and Milliseconds along with TimedStopEvent are used for fixed durations.

Once these classes were added, I was able to judge the viability of the data structures. First, writing these abstract classes and adding them to the main program proved to be quite easy. The main problems that I encountered were not

[6] These classes were simple test examples only, and are not included in the main design.

related to the way in which the data was organised, but instead stemmed from problems with thread deadlock in the SequenceDrivers (see discussion in 2.8). Second, in coding the Runner applet, I noted that use of the tree structure made it much simpler to deal with the various multimedia elements. Since the top element both controls and depends on the child objects below it, any methods called on that top element were propagated properly down the tree without any difficulty. Finally, very little debugging was needed, despite the complexity of the data structures.

Summary and Conclusion

The digital age that we live in has brought us many fresh possibilities. However, these new options are only valuable if they serve our own needs. Unfortunately, in many cases, this fails to be true. The computer has changed all of our lives, but has not necessarily allowed us to do our work more quickly and easily. This is especially true in the realm of creativity, where software that requires a large investment in time and knowledge can frustrate the creative mind, rather than help to liberate it.

It was the apparent lack of viable software for interactive multimedia composition that led to this investigation, which ventures beyond the most prevalent options rather than remaining content with the non-musical paradigms of multimedia tools such as QuickTime and Shockwave. This thesis shows that, with knowledge of Java programming and software design skills, it is possible to create an original program that better suits the needs of some musicians—in particular, myself. After studying the features that Java offered, it was possible to determine which of them would be the most useful for multimedia composition. At the same time, I tried to find ways to define the types of musical structures that I wanted to use in my compositions.

Considering the ease with which it was possible to implement complex musical structures in Java, one must conclude that Java is indeed a viable tool for creating Web-ready, truly interactive compositions that combine several media. As for the "solution" presented in this thesis, it is far from complete and probably reflects the limitations of my personal view of music. Nonetheless, I hope that this Java implementation of a scripting language developed for my own interactive multimedia creations will serve as a starting point for other composers.

Bibliography

Apple Computer, Inc. 1999. "Introduction to Wired Movies, Sprites, and the Sprite Toolbox." Online edition. http://developer.apple.com/techpubs/quicktime/qtdevdocs/REF/refWiredIntro.htm

Apple Computer, Inc. 1999. " Movie Toolbox Fundamentals." Online edition. http://developer.apple.com/techpubs/quicktime/qtdevdocs/RM/rmMTFundamentals.htm

Apple Computer, Inc. 1999. "QuickTime Overview." Online edition. http://developer.apple.com/techpubs/quicktime/qtdevdocs/RM/rmQTOverview.htm

Campione, M., and Walrath, K. 1999. *The JFC Swing Tutorial: A Guide to Constructing GUIs*. Online edition. http://java.sun.com/docs/books/tutorial/

Campione, M., Walrath, K., Huml, A., et al. 1998. *The Java™ Tutorial Continued: The Rest of the JDK™*. Online edition. http://java.sun.com/docs/books/tutorial/

Campione, M., and Walrath, K. 1998. *The Java Tutorial Second Edition: Object-Oriented Programming for the Internet.* Online edition. http://java.sun.com/docs/books/tutorial/

Fisher, S. 1997. *Creating Dynamic Web Sites: a Webmaster's Guide to Interactive Multimedia.* Reading, MA: Addison-Wesley Developers Press.

Flanagan, D. 1997. *Java in a Nutshell, Second Edition.* Sebastopol, CA: O'Reilly.

Gordon, R., and Talley, S. 1999. *Essential JMF: Java™ Media Framework.* Upper Saddle River, NJ: Prentice Hall PTR.

Gosling, J., Joy, B., and Steele, G. 1996. *The Java Language Specification.* Online edition. http://java.sun.com/docs/books/jls/html/index.html

Pawlan, M. 1999. "Java™ Programming Language Basics, Part 2." Online edition. http://developer.java.sun.com/developer/onlineTraining/Programming/BasicJava2/

Pawlan, M. 1999. "Java™ Programming Language Basics, Part 1." Online edition. http://developer.java.sun.com/developer/onlineTraining/Programming/BasicJava1/

Rosenweig, G. 1997. *The Director 6 Book*. Research Triangle Park, NC: Ventana Communications.

Rowe, R. 1993. *Interactive Music Systems*. Cambridge, MA: MIT Press.

Sanchez, R. 1999. "Reviews: LiveStage DR 1.0.1 vs. Electrifier Pro 1.0." *MacAddict* 35: 52-53.

Sanchez, R. 1999. "Reviews: Director 7 Shockwave Internet Studio." *MacAddict* 33: 44-46.

Simmons, M. 1999. "How To: Build Interactive QuickTime Movies." *MacAddict* 34: 72-75.

Sun Microsystems, Inc. 1999. *Java™ Media APIs*. Online edition. http://java.sun.com/products/java-media/

Sun Microsystems, Inc. 1999. *Swing 1.1.1 API Specification*. Online edition. http://java.sun.com/products/jfc/swingdoc-api-1.1.1/

Sun Microsystems, Inc. 1997. *Javadoc Home Page*. Online edition. http://java.sun.com/products/jdk/javadoc/index.html

Thomas, G. 1998. "How To: Build a Shoot-'Em-Up Game in Flash 3." *MacAddict* 28: 88-93.

W3C. 1998. "Synchronized Multimedia Integration Language (SMIL) 1.0 Specification." Online edition. http://www.w3.org/TR/REC-smil/

Appendix A: API guide to the Java implementation

This section contains the API (Application Program Interface) guide generated by the Javadoc tool for the 38 classes used in the implementation of the scripting language. This guide fully documents the programming work for this thesis.

By creating the Java implementations for these classes, it was possible to judge the viability of both the data structures and the scripting language. Furthermore, it was this concrete programming work that allowed the discovery of several important features of Java, which were subsequently exploited in the code. Finally, by creating these classes and thereby achieving the goal of implementing this scripting language in Java, this thesis proves that Java is a viable tool for the creation of interactive multimedia compositions.

package jsb.app

Class Index

- Builder
- Runner

Exception Index

- FileException

Class jsb.app.Builder

```
Object
   |
   +----jsb.app.Builder
```

public class **Builder**
extends Object

The main script-editing application. Its main responsibility is to be a mediation point between the different `SequenceFormats`, and to provide any extra capabilities needed.

In its current version, this class moderates between a `SequenceGui` and several `SequenceTexts`, and provides extra file capabilities to the main `DataFrame`.

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)
See Also:
SequenceFormat, SequenceGui, SequenceText, DataFrame

Constructor Index

- **jsb.app.Builder**()

Method Index

- **constructMenuBar**()
  Constructs and returns a menu bar with File menu suitable for general file management.
- **constructObjectMenu**()
  Constructs and returns an Objects menu suitable for management of `SequenceObjects`.
- **main**(String[])
  Provides the main execution environment.
- **setCurrFile**(File)
  Sets the current save file after verifying that the file is writable.
- **setSaved**(boolean)
  Sets the current save status.

Constructors

- **Builder**

```
public Builder()
```

Methods

- **constructMenuBar**

```
javax.swing.JMenuBar constructMenuBar()
```

Constructs and returns a menu bar with File menu suitable for general file management.

Returns:
a Swing-compatible menu bar to be added to the application window

- **constructObjectMenu**

```
javax.swing.JMenu constructObjectMenu()
```

Constructs and returns an Objects menu suitable for management of `SequenceObjects`.

Returns:
a Swing-compatible Objects menu to be added to the application window, both to its menu bar and its data representation.

- **main**

```
public static void main(String[] args)
```

Provides the main execution environment.

Parameters:
args - the command-line arguments to this application (not used)

- **setCurrFile**

```
protected void setCurrFile(File aFile) throws FileException
```

Sets the current save file after verifying that the file is writable.

Parameters:
aFile - a file to be used as the current save file
Throws: FileException
if aFile is not writable

- **setSaved**

```
protected void setSaved(boolean aSavedFlag)
```

Sets the current save status.

Parameters:
aSavedFlag - `true` if all data is currently saved, `false` otherwise

Class jsb.app.Runner

```
Object
   |
   +----Component
           |
           +----Container
                   |
                   +----Panel
                           |
                           +----Applet
                                   |
                                   +----jsb.app.Runner
```

public class **Runner**
extends Applet

Constructor Index

- jsb.app.Runner()

Method Index

- init()
- start()
- stop()

Constructors

- **Runner**

```
public Runner()
```

Methods

- **init**

```
public void init()
```

Overrides:
init in class Applet

- **start**

```
public void start()
```

Overrides:
start in class Applet

- **stop**

```
public void stop()
```

Overrides:
stop in class Applet

Class jsb.app.FileException

```
Object
   |
   +----Throwable
           |
           +----Exception
                   |
                   +----jsb.app.FileException
```

public class **FileException**
extends Exception

A set of file handling exceptions, each differentiated by its error code.

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)

Variable Index

- NOT_FOUND
  Indicates that the file does not exist.
- NOT_WRITABLE
  Indicates that the file cannot be written into.
- READ_ERROR
  Indicates that an error occured when reading the file.
- WRITE_ERROR
  Indicates that an error occured when writing into the file.

Constructor Index

- jsb.app.FileException(int, String)
  Constructs a file exception with the given error code and message.
- jsb.app.FileException(int)
  Constructs a file exception with the given error code.

Method Index

- getErrorCode()
  Returns the error code, which can be used for error recovery purposes.

Variables

- **NOT_FOUND**

```
public static final int NOT_FOUND
```

Indicates that the file does not exist.

- **NOT_WRITABLE**

```
public static final int NOT_WRITABLE
```

Indicates that the file cannot be written into.

- **READ_ERROR**

```
public static final int READ_ERROR
```

Indicates that an error occured when reading the file.

- **WRITE_ERROR**

```
public static final int WRITE_ERROR
```

Indicates that an error occured when writing into the file.

Constructors

- **FileException**

```
public FileException(int anErrorCode,
                     String aMessage)
```

Constructs a file exception with the given error code and message.

Parameters:
anErrorCode - an int that defines the type of error
aMessage - a message that can be used to display the error

- **FileException**

```
public FileException(int anErrorCode)
```

Constructs a file exception with the given error code.

Parameters:
anErrorCode - an int that defines the type of error

Methods

- **getErrorCode**

```
public int getErrorCode()
```

Returns the error code, which can be used for error recovery purposes.

Returns:
an int that defines the type of error

---

package jsb.basicimpl

Class Index

- AudioClip
- Image
- KeyPress
- KeyPressedStopEvent
- Milliseconds
- Parallel
- Sequential
- Shuffle
- TimedStopEvent

Class jsb.basicimpl.AudioClip

```
Object
   |
   +----ParameterizedObject
           |
           +----SequenceDriver
                   |
                   +----AudioDriver
                           |
                           +----jsb.basicimpl.AudioClip
```

public class **AudioClip**
extends AudioDriver

A single segment of audio, loaded from a file, that can be played as part of a sequence. This acts as a wrapper for the `java.applet.AudioClip` class.

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)

Constructor Index

- **jsb.basicimpl.AudioClip**()
  Constructs a wrapper for a `java.applet.AudioClip`.

Method Index

- **load**(Applet)
  Loads the audio data needed to play this audio clip.
- **play**()
  Plays this audio clip for its full duration, unless interrupted.
- **stop**()
  Stops playback of this audio clip.

Constructors

- **AudioClip**

```
public AudioClip()
```

Constructs a wrapper for a `java.applet.AudioClip`.

Methods

- **load**

```
public synchronized void load(Applet anApplet)
        throws UndefinedException
```

Loads the audio data needed to play this audio clip. This method blocks until all data has been read.

Parameters:
anApplet - the parent applet through which the audio will be played
Overrides:
load in class SequenceDriver

- **play**

```
public synchronized void play()
```

Plays this audio clip for its full duration, unless interrupted. This method blocks during playback.

Overrides:
play in class SequenceDriver

- **stop**

```
public synchronized void stop()
```

Stops playback of this audio clip. This method blocks until playback stops.

Overrides:
stop in class SequenceDriver

Class jsb.basicimpl.Image

```
Object
   |
   +----ParameterizedObject
           |
           +----SequenceDriver
                   |
                   +----ImageDriver
                           |
                           +----jsb.basicimpl.Image
```

public class **Image**
extends ImageDriver

A single image, loaded from a file, that can be shown as part of a sequence. This acts as a wrapper for the `java.awt.Image` class.

Version:
August 1999

Author:
Jacqueline Beaulac (Faculty of Music, McGill University)

Constructor Index

- **jsb.basicimpl.Image**()
  Constructs a wrapper for a `java.awt.Image`.

Method Index

- **load**(Applet)
  Loads the image data needed to show this image.
- **play**()
  Shows this image.
- **stop**()
  Stops showing this image.

Constructors

- **Image**

```
public Image()
```

Constructs a wrapper for a `java.awt.Image`.

Methods

- **load**

```
public synchronized void load(Applet anApplet)
        throws UndefinedException
```

Loads the image data needed to show this image. This method blocks until all data has been read.

Parameters:
anApplet - the parent applet on which the image will be displayed
Overrides:
load in class SequenceDriver

- **play**

```
public synchronized void play()
```

Shows this image. This method blocks during playback.

Overrides:
play in class SequenceDriver

- **stop**

```
public synchronized void stop()
```

Stops showing this image. This method blocks until the image has been removed from the applet.

Overrides:
stop in class SequenceDriver

Class jsb.basicimpl.KeyPress

```
Object
   |
   +----ParameterizedObject
           |
           +----StopTrigger
                   |
                   +----jsb.basicimpl.KeyPress
```

public class **KeyPress**
extends StopTrigger

A stop trigger that goes off when a key is pressed.

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)

Constructor Index

- **jsb.basicimpl.KeyPress**()
  Constructs a stop trigger that goes off when a key is pressed.

Method Index

- **activate**()
  Adds the listener and enables this trigger to throw `KeyPressedStopEvents`.

Constructors

- **KeyPress**

`public KeyPress()`

Constructs a stop trigger that goes off when a key is pressed. The parameters for this instance are initialized here.

Methods

- **activate**

```
public void activate()
```

Adds the listener and enables this trigger to throw `KeyPressedStopEvents`.

Overrides:
activate in class StopTrigger
See Also:
KeyPressedStopEvent

Class jsb.basicimpl.KeyPressedStopEvent

```
Object
   |
   +----StopEvent
           |
           +----jsb.basicimpl.KeyPressedStopEvent
```

public class **KeyPressedStopEvent**
extends StopEvent

An event that indicates the end of a section of a sequence after the user pressed a key.

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)

Constructor Index

- **jsb.basicimpl.KeyPressedStopEvent**(StopTrigger, char)
  Constructs a stop event that will be used to indicate that a key was pressed by the user.

Constructors

- **KeyPressedStopEvent**

```
public KeyPressedStopEvent(StopTrigger aSource,
                           char aKey)
```

Constructs a stop event that will be used to indicate that a key was pressed by the user.

Parameters:
aSource - the stop trigger that will send out this event
aKey - the character typed by the user

Class jsb.basicimpl.Milliseconds

```
Object
   |
   +----ParameterizedObject
           |
           +----StopTrigger
                   |
                   +----jsb.basicimpl.Milliseconds
```

public class **Milliseconds**
extends StopTrigger

A stop trigger that goes off after a specific amount of time.

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)

Constructor Index

- **jsb.basicimpl.Milliseconds**()
  Constructs a stop trigger that goes off after a specific amount of time.

Method Index

- **activate**()
  Starts the timer and enables this trigger to throw `TimedStopEvents`.

Constructors

- **Milliseconds**

```
public Milliseconds()
```

Constructs a stop trigger that goes off after a specific amount of time. The parameters for this instance are initialized here.

Methods

- **activate**

```
public void activate()
```

Starts the timer and enables this trigger to throw `TimedStopEvents`.

Overrides:
activate in class StopTrigger

See Also:
TimedStopEvent

Class jsb.basicimpl.Parallel

```
Object
   |
   +----ParameterizedObject
           |
           +----SequenceDriver
                   |
                   +----SetManager
                           |
                           +----jsb.basicimpl.Parallel
```

public class **Parallel**
extends SetManager

A set manager that plays all its children back simultaneously.

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)

Constructor Index

- **jsb.basicimpl.Parallel**()

Method Index

- **load**(Applet)
  Loads all the child drivers simultaneously.
- **playOneCycle**()
  Plays all child drivers simultaneously.
- **stopCurrCycle**()
  Stops the playback of all children simultaneously.

Constructors

- **Parallel**

```
public Parallel()
```

Methods

- **load**

```
public void load(Applet anApplet)
```

Loads all the child drivers simultaneously. Blocks until all the children have been loaded.

Parameters:
anApplet - the applet on which the child drivers depend to load their data
Overrides:
load in class SequenceDriver

- **playOneCycle**

```
protected void playOneCycle()
```

Plays all child drivers simultaneously. Blocks till the playback of all children has finished.

Overrides:
playOneCycle in class SetManager

- **stopCurrCycle**

```
protected void stopCurrCycle()
```

Stops the playback of all children simultaneously.

Overrides:
stopCurrCycle in class SetManager

Class jsb.basicimpl.Sequential

```
Object
   |
   +----ParameterizedObject
           |
           +----SequenceDriver
                   |
                   +----SetManager
                           |
                           +----jsb.basicimpl.Sequential
```

public class **Sequential**
extends SetManager

A set manager that plays its children back in an ordered sequence, one at a time.

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)

Constructor Index

- **jsb.basicimpl.Sequential**()

Method Index

- **load**(Applet)
  Loads each of the child drivers in the order in which they will be played.
- **playOneCycle**()
  Plays the next child in the sequence.
- **stopCurrCycle**()
  Stops the playback of the current child.

Constructors

- **Sequential**

```
public Sequential()
```

Methods

- **load**

```
public void load(Applet anApplet) throws UndefinedException
```

Loads each of the child drivers in the order in which they will be played.

Parameters:
anApplet - the applet on which the child drivers depend to load their data
Overrides:
load in class SequenceDriver

- **playOneCycle**

```
protected void playOneCycle()
```

Plays the next child in the sequence. Blocks until the playback of the child is finished.

Overrides:
playOneCycle in class SetManager

- **stopCurrCycle**

```
protected void stopCurrCycle()
```

Stops the playback of the current child.

Overrides:
stopCurrCycle in class SetManager

Class jsb.basicimpl.Shuffle

```
Object
   |
   +----ParameterizedObject
           |
           +----SequenceDriver
                   |
                   +----SetManager
                           |
                           +----jsb.basicimpl.Shuffle
```

public class **Shuffle**
extends SetManager

A set manager that plays its children back in a shuffled sequence, one at a time.

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)

Constructor Index

- **jsb.basicimpl.Shuffle**()

Method Index

- **load**(Applet)
  Determines the shuffled order and loads the child drivers in that order.
- **playOneCycle**()
  Plays the next child in the shuffled sequence.
- **stopCurrCycle**()
  Stops the playback of the current child.

Constructors

- **Shuffle**

```
public Shuffle()
```

Methods

- **load**

```
public void load(Applet anApplet) throws UndefinedException
```

Determines the shuffled order and loads the child drivers in that order.

Parameters:
anApplet - the applet on which the child drivers depend to load their data
Overrides:
load in class SequenceDriver

- **playOneCycle**

```
protected void playOneCycle()
```

Plays the next child in the shuffled sequence. Blocks until the playback of the child is finished.

Overrides:
playOneCycle in class SetManager

- **stopCurrCycle**

```
protected void stopCurrCycle()
```

Stops the playback of the current child.

Overrides:
stopCurrCycle in class SetManager

Class jsb.basicimpl.TimedStopEvent

```
Object
   |
   +----StopEvent
           |
           +----jsb.basicimpl.TimedStopEvent
```

public class **TimedStopEvent**
extends StopEvent

An event that indicates the end of a section of a sequence after a specific length of time.

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)

Constructor Index

- **jsb.basicimpl.TimedStopEvent**(StopTrigger, long)
  Constructs a stop event that will be used to indicate that a specific length of time has passed.

Constructors

- **TimedStopEvent**

```
public TimedStopEvent(StopTrigger aSource,
                             long aDuration)
```

Constructs a stop event that will be used to indicate that a specific length of time has passed.

Parameters:
aSource - the stop trigger that will send out this event
aDuration - the duration in milliseconds

package jsb.core

Interface Index

- SequenceFormat
- StopListener

Class Index

- AudioDriver
- BasicSequenceObject
- CompositeSequenceObject
- CompositeStopTrigger
- ImageDriver
- ParameterizedObject
- SequenceDriver
- SequenceObject
- SetManager
- StopEvent
- StopTrigger
- StopTriggerOp

Exception Index

- ClassLoadFailureException
- UndefinedException

Interface jsb.core.SequenceFormat

public interface **SequenceFormat**

An object that is responsible for both constructing and parsing a particular representation of a sequence, ex. text, graphical.

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)

Method Index

- **addElement**(SequenceObject)
  Adds a section to the sequence.
- **elements**()
  Returns an enumerated list of all the sections that can be used for iteration.
- **getAllElements**()
  Returns a table of all the sections, keyed by ID.
- **getFormattedSequence**()
  Returns a representation of the sequence in a unique format.
- **removeElement**(Object)
  Removes a section from the sequence.
- **setFormattedSequence**(Object)
  Sets the formatted object that is to be parsed.

Methods

- **addElement**

```
public abstract void addElement(SequenceObject aSequenceObject)
        throws ClassLoadFailureException
```

Adds a section to the sequence.

Parameters:
aSequenceObject - a section to be added to the current sequence
Throws: ClassLoadFailureException
if a class required by the new section cannot be found

- **elements**

```
public abstract java.util.Enumeration elements()
```

Returns an enumerated list of all the sections that can be used for iteration.

Returns:
a list of all sections

- **getAllElements**

```
public abstract java.util.Hashtable getAllElements()
```

Returns a table of all the sections, keyed by ID.

Returns:
a table of all sections

- **getFormattedSequence**

```
public abstract java.lang.Object getFormattedSequence()
```

Returns a representation of the sequence in a unique format.

Returns:
an object that contains the data for a full sequence

- **removeElement**

```
public abstract jsb.core.SequenceObject removeElement(Object aKey)
```

Removes a section from the sequence.

Parameters:
aKey - an ID used to find the SequenceObject to be removed from the current sequence

- **setFormattedSequence**

```
public abstract void setFormattedSequence(Object aSequence)
        throws Exception
```

Sets the formatted object that is to be parsed.

Parameters:
aSequence - a full sequence
Throws: Exception
if a parsing error occurs

Interface jsb.core.StopListener

public interface **StopListener**

A listener that should be notified when a `StopEvent` occurs.

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)
See Also:
StopEvent

Method Index

- **stopRequested**(StopEvent)
Handles a `StopEvent`.

Methods

- **stopRequested**

```
public abstract void stopRequested(StopEvent aStopEvent)
```

Handles a `StopEvent`. This should perform all the actions required when playback of a sequence, or of a section of a sequence, should stop.

Parameters:
aStopEvent - the event that was dispatched

Class jsb.core.AudioDriver

```
Object
   |
   +----ParameterizedObject
           |
           +----SequenceDriver
                   |
                   +----jsb.core.AudioDriver
```

public abstract class **AudioDriver**
extends SequenceDriver

A driver that plays audio content.

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)

Constructor Index

- **jsb.core.AudioDriver**()

Constructors

- **AudioDriver**

```
public AudioDriver()
```

Class jsb.core.BasicSequenceObject

```
Object
   |
   +----SequenceObject
           |
           +----jsb.core.BasicSequenceObject
```

public class **BasicSequenceObject**
extends SequenceObject

A section of a sequence that contains a single piece of media.

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)

Constructor Index

- **jsb.core.BasicSequenceObject**(String, StopTrigger, SequenceDriver)
  Constructs a sequence object that contains one specific piece of media.

Constructors

- **BasicSequenceObject**

```
public BasicSequenceObject(String anId,
                           StopTrigger aStopTrigger,
                           SequenceDriver aSequenceDriver)
       throws UndefinedException
```

Constructs a sequence object that contains one specific piece of media.

Parameters:
anId - a unique ID
aStopTrigger - a stop trigger that controls the end of playback
aSequenceDriver - a driver that provides media support
Throws: UndefinedException
if any one of the parameters is badly defined or is missing

Class jsb.core.CompositeSequenceObject

```
Object
   |
   +----SequenceObject
           |
           +----jsb.core.CompositeSequenceObject
```

public class **CompositeSequenceObject**
extends SequenceObject

A section of a sequence that contains a set of subsections.

Version:
August 1999

Author:
Jacqueline Beaulac (Faculty of Music, McGill University)

Constructor Index

- **jsb.core.CompositeSequenceObject**(String, Vector, StopTrigger, SetManager)
  Constructs a sequence object that contains several other sequence objects.

Method Index

- **getChildList**()
  Returns the IDs of the child objects on which this object depends.
- **getSetManager**()
  Returns the manager that will control the order and timing of the playback of the child objects.

Constructors

- **CompositeSequenceObject**

```
public CompositeSequenceObject(String anId,
                               Vector aChildList,
                               StopTrigger aStopTrigger,
                               SetManager aSetManager)
        throws UndefinedException
```

Constructs a sequence object that contains several other sequence objects.

Parameters:
anId - a unique ID
aChildList - a list of the IDs of the child `SequenceObjects`
aStopTrigger - a stop trigger that controls end of playback
aSetManager - a manager that controls the ordering and playback of the child objects

Throws: UndefinedException
if any one of the parameters is badly defined

Methods

- **getChildList**

```
public java.util.Vector getChildList()
```

Returns the IDs of the child objects on which this object depends.

Returns:
a list of child IDs

- **getSetManager**

```
public jsb.core.SetManager getSetManager()
```

Returns the manager that will control the order and timing of the playback of the child objects. This is a convenience method that casts the driver to a `SetManager`.

Returns:
the set manager that orders the playback of child objects

Class jsb.core.CompositeStopTrigger

```
Object
   |
   +----ParameterizedObject
           |
           +----StopTrigger
                   |
                   +----jsb.core.CompositeStopTrigger
```

public class **CompositeStopTrigger**
extends StopTrigger
implements StopListener

An association between two `StopTriggers`. The connection between the two is defined by a logical operator ("and" or "or").

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)

Constructor Index

- **jsb.core.CompositeStopTrigger**(StopTrigger, StopTriggerOp, StopTrigger)
  Constructs an association between two stop triggers and based on a logical operator.

Method Index

- **activate**()
  Enables the trigger to throw `StopEvents`.
- **getFirstStopTrigger**()
  Returns the first stop trigger of the association.
- **getOp**()
  Returns the logical operator that defines this association.
- **getSecondStopTrigger**()
  Returns the second stop trigger of the association.
- **stopRequested**(StopEvent)
  Handles a `StopEvent`.

Constructors

- **CompositeStopTrigger**

```
public CompositeStopTrigger(StopTrigger aFirstStopTrigger,
                            StopTriggerOp anOp,
                            StopTrigger aSecondStopTrigger)
```

Constructs an association between two stop triggers and based on a logical operator.

Parameters:
aFirstStopTrigger - one of the two stop triggers
anOp - the logical operator
aSecondStopTrigger - the other stop trigger

Methods

- **activate**

```
public void activate()
```

Enables the trigger to throw `StopEvents`. This must be called when the trigger is supposed to start checking for its stop condition. The trigger is automatically deactivated once the stop condition is met.

Overrides:
activate in class StopTrigger

- **getFirstStopTrigger**

```
public jsb.core.StopTrigger getFirstStopTrigger()
```

Returns the first stop trigger of the association.

Returns:
one of the two stop triggers

- **getOp**

```
public jsb.core.StopTriggerOp getOp()
```

Returns the logical operator that defines this association.

Returns:
the operator that defines the relationship between the two triggers

- **getSecondStopTrigger**

```
public jsb.core.StopTrigger getSecondStopTrigger()
```

Returns the second stop trigger of the association.

Returns:
one of the two stop triggers

- **stopRequested**

```
public void stopRequested(StopEvent aStopEvent)
```

Handles a `StopEvent`. If the operator is an "or", the stop event is always dispatched to the listeners registered with this association. If the operator is an "and", the stop event is dispatched to the listeners only if both child triggers have gone off.

Parameters:
aStopEvent - the event that was dispatched

Class jsb.core.ImageDriver

```
Object
   |
   +----ParameterizedObject
           |
           +----SequenceDriver
                   |
                   +----jsb.core.ImageDriver
```

public abstract class **ImageDriver**
extends SequenceDriver

A driver that displays visual content.

Version:
  August 1999
Author:
  Jacqueline Beaulac (Faculty of Music, McGill University)

Constructor Index

- jsb.core.ImageDriver()

Constructors

- **ImageDriver**

```
public ImageDriver()
```

Class jsb.core.ParameterizedObject

```
Object
   |
   +----jsb.core.ParameterizedObject
```

public abstract class **ParameterizedObject**
extends Object

A generic object that contains a set of parameters and their associated values.

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)

Variable Index

- **defaultParams**
  The table of all legal parameters and their default values.
- **paramValues**
  The table of all parameters and their current values.

Constructor Index

- **jsb.core.ParameterizedObject**()

Method Index

- **getParamValues**()
  Gets the values of all of the parameters used by this class.
- **setParamValues**(Hashtable)
  Sets the values of any of the parameters used by this class.

Variables

- **defaultParams**

```
public final java.util.Hashtable defaultParams
```

The table of all legal parameters and their default values. This is defined `public` so that all subclasses may access it.

- **paramValues**

```
public final java.util.Hashtable paramValues
```

The table of all parameters and their current values. This is defined `public` so that all subclasses may access it.

Constructors

- **ParameterizedObject**

```
public ParameterizedObject()
```

Methods

- **getParamValues**

```
public java.util.Hashtable getParamValues()
```

Gets the values of all of the parameters used by this class.

Returns:
a table containing all legal parameters and their currently associated values

- **setParamValues**

```
public void setParamValues(Hashtable aParamTable)
        throws UndefinedException
```

Sets the values of any of the parameters used by this class.

Parameters:
aParamTable - a table containing some known parameters along with the values with which they should be associated

Class jsb.core.SequenceDriver

```
Object
   |
   +----ParameterizedObject
           |
           +----jsb.core.SequenceDriver
```

public abstract class **SequenceDriver**
extends ParameterizedObject
implements StopListener

An object that is responsible for playing a section of a sequence. If the section contains a single piece of media, then the driver plays the audio and/or shows the visuals. If the section contains subsections, then the driver is responsible for determining the order in which the subsections are played, and for playing them in that order.

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)

Variable Index

- **theStopTrigger**

Constructor Index

- **jsb.core.SequenceDriver**()

Method Index

- **getInstance**(String)
  Returns an instance of the named driver.
- **load**(Applet)
  Loads all data needed to play the media.
- **play**()
  Plays the media for its total duration.
- **setStopTrigger**(StopTrigger)
  Sets the stop trigger which should be activated when playback begins.
- **stop**()
  Stops the playback of the media as soon as possible.

- stopRequested(StopEvent)
  Handles a `StopEvent`.

Variables

- **theStopTrigger**

```
protected jsb.core.StopTrigger theStopTrigger
```

Constructors

- **SequenceDriver**

```
public SequenceDriver()
```

Methods

- **getInstance**

```
public static jsb.core.SequenceDriver getInstance
        (String aClassName)
        throws ClassLoadFailureException
```

Returns an instance of the named driver. This is a convenience method which does type checking and translates any exception to a `ClassLoadFailureException`.

Parameters:
aClassName - the driver to be instantiated
Throws: ClassLoadFailureException
if no instance of the named class can be constructed, or if the named class is not a driver

- **load**

```
public abstract void load(Applet theApplet)
        throws UndefinedException
```

Loads all data needed to play the media.

This method should be blocking. Classes that call this method must perform any thread handling.

Parameters:
theApplet - the applet to be used for playback

- **play**

```
public abstract void play()
```

Plays the media for its total duration. This should also activate the stop trigger, if there is one.

This method should be blocking. Classes that call this method must perform any thread handling.

- **setStopTrigger**

```
public void setStopTrigger(StopTrigger aTrigger)
```

Sets the stop trigger which should be activated when playback begins.

Parameters:
aTrigger - the stop trigger associated with this driver

- **stop**

```
public abstract void stop()
```

Stops the playback of the media as soon as possible.

This method should be blocking. Classes that call this method must perform any thread handling.

- **stopRequested**

```
public void stopRequested(StopEvent aStopEvent)
```

Handles a `StopEvent`. This should perform all the actions required when playback of a sequence, or of a section of a sequence, should stop.

Parameters:
aStopEvent - the event that was dispatched

Class jsb.core.SequenceObject

```
Object
   |
   +----jsb.core.SequenceObject
```

public class **SequenceObject**
extends Object

A section of a sequence.

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)

Constructor Index

- jsb.core.SequenceObject(String, StopTrigger, SequenceDriver)
  Constructs a SequenceObject which contains all the data necessary to be played as a unique section within a sequence.

Method Index

- getId()
  Returns the ID used to identify this section within the sequence.
- getSequenceDriver()
  Gets the main driver class that is used to play this section.
- getStopTrigger()
  Returns the trigger that will cause the playback of this section to stop.
- setStopTrigger(StopTrigger)
  Sets the StopTrigger for this section.

Constructors

- **SequenceObject**

```
 SequenceObject(String anId,
                StopTrigger aStopTrigger,
                SequenceDriver anSequenceDriver)
        throws UndefinedException
```

Constructs a SequenceObject which contains all the data necessary to be played as a unique section within a sequence.

Parameters:
anId - a string that uniquely identifies this section
aStopTrigger - a trigger that controls end of playback
anSequenceDriver - a driver that handles media playback
Throws: UndefinedException
if any one of the parameters is badly defined

Methods

- **getId**

```
public java.lang.String getId()
```

Returns the ID used to identify this section within the sequence.

Returns:
the unique ID for this section

- **getSequenceDriver**

```
public jsb.core.SequenceDriver getSequenceDriver()
```

Gets the main driver class that is used to play this section.

Returns:
a driver that handles media playbakc within this section

- **getStopTrigger**

```
public jsb.core.StopTrigger getStopTrigger()
```

Returns the trigger that will cause the playback of this section to stop. A section higher up in the hierarchy may stop playback before this trigger does.

Returns:
the stop trigger

- **setStopTrigger**

```
public void setStopTrigger(StopTrigger aStopTrigger)
```

Sets the StopTrigger for this section.

Parameters:
aStopTrigger - a trigger that should stop playback of this section

Class jsb.core.SetManager

```
Object
   |
   +----ParameterizedObject
           |
           +----SequenceDriver
                   |
                   +----jsb.core.SetManager
```

public abstract class **SetManager**
extends SequenceDriver

A playback manager for a set of child `SequenceDrivers`.

Version:
August 1999

Author:
Jacqueline Beaulac (Faculty of Music, McGill University)

Variable Index

- **childDriverList**
  The list of child drivers that are under the control of this manager.
- **currCycleNum**
  The number of times that a cycle of playback has occurred.

Constructor Index

- **jsb.core.SetManager**()

Method Index

- **getCurrCycleNum**()
  Returns the number of playback cycles that have occurred.
- **play**()
  Plays the child drivers.
- **playOneCycle**()
  Causes a single playback cycle.
- **setChildDrivers**(Vector)
  Sets the child `SequenceDrivers` that are under the control of this driver.
- **stop**()
  Stops the playback of the child drivers.

- stopCurrCycle()
  Stops the current playback cycle.

Variables

- **childDriverList**

```
protected java.util.Vector childDriverList
```

The list of child drivers that are under the control of this manager.

- **currCycleNum**

```
protected int currCycleNum
```

The number of times that a cycle of playback has occurred. Each implementation of this class is responsible for determining what constitutes a single playback cycle.

Constructors

- **SetManager**

```
public SetManager()
```

Methods

- **getCurrCycleNum**

```
public int getCurrCycleNum()
```

Returns the number of playback cycles that have occurred. Each implementation of this class is responsible for determining what constitutes a single playback cycle.

Returns:
the current playback cycle count

- **play**

```
public final void play()
```

Plays the child drivers. This also handles the stop trigger.

This method should be blocking. Classes that call this method must perform any thread handling.

Overrides:
play in class SequenceDriver

- **playOneCycle**

```
protected abstract void playOneCycle()
```

Causes a single playback cycle.

This method should be blocking.

- **setChildDrivers**

```
public final void setChildDrivers(Vector aChildDriverList)
        throws UndefinedException
```

Sets the child `SequenceDrivers` that are under the control of this driver.

Parameters:
aChildDriverList - a list of child drivers
Throws: UndefinedException
if the list is null

- **stop**

```
public final void stop()
```

Stops the playback of the child drivers.

Overrides:
stop in class SequenceDriver

- **stopCurrCycle**

```
protected abstract void stopCurrCycle()
```

Stops the current playback cycle.

This method should be blocking.

Class jsb.core.StopEvent

```
Object
   |
   +----jsb.core.StopEvent
```

public class **StopEvent**
extends Object

An event that indicates the end of a section of a sequence.

Version:
August 1999

Author:
Jacqueline Beaulac (Faculty of Music, McGill University)

Variable Index

- source
  The stop trigger that is sending out this event.

Constructor Index

- jsb.core.StopEvent(StopTrigger)
  Constructs a stop event that will be used to indicate that the end condition of a particular stop trigger became true.

Method Index

- getSource()
  Returns the stop trigger that sent out this event when its end condition became true.

Variables

- **source**

```
protected jsb.core.StopTrigger source
```

The stop trigger that is sending out this event. This is declared `protected` so that subclasses of this class can access it.

Constructors

- **StopEvent**

```
public StopEvent(StopTrigger aSource)
```

Constructs a stop event that will be used to indicate that the end condition of a particular stop trigger became true.

Parameters:
aSource - the stop trigger that will send out this event

Methods

- **getSource**

```
public jsb.core.StopTrigger getSource()
```

Returns the stop trigger that sent out this event when its end condition became true.

Returns:
the stop trigger that sent out this event

Class jsb.core.StopTrigger

```
Object
   |
   +----ParameterizedObject
           |
           +----jsb.core.StopTrigger
```

public abstract class **StopTrigger**
extends ParameterizedObject

A trigger that will cause a `StopEvent` to be thrown once certain conditions are met. This is used to stop the playback of a sequence or a section of a sequence.

Version:
August 1999

Author:
Jacqueline Beaulac (Faculty of Music, McGill University)

See Also:
StopEvent

Variable Index

- **listenerList**
  Holds a list of references to all listeners that have registered themselves with this object.

Constructor Index

- **jsb.core.StopTrigger**()

Method Index

- **activate**()
  Enables the trigger to throw `StopEvents`.
- **addStopListener**(StopListener)
  Adds a listener that should be sent any `StopEvents`.
- **dispatchStopEvent**(StopEvent)
  Iterates through the list of listeners and sends an event to each one.
- **getInstance**(String)
  Returns an instance of the named trigger.
- **removeStopListener**(StopListener)
  Removes a listener so that `StopEvents` are no longer sent to it.

Variables

- **listenerList**

```
protected final java.util.Vector listenerList
```

Holds a list of references to all listeners that have registered themselves with this object. This is declared `protected` so that subclasses of this class can access the list.

Constructors

- **StopTrigger**

```
public StopTrigger()
```

Methods

- **activate**

```
public abstract void activate()
```

Enables the trigger to throw `StopEvents`. This must be called when the trigger is supposed to start checking for its stop condition. The trigger is automatically deactivated once the stop condition is met.

- **addStopListener**

```
public void addStopListener(StopListener aListener)
```

Adds a listener that should be sent any `StopEvents`.

Parameters:
aListener - a StopListener to be added to this trigger

- **dispatchStopEvent**

```
protected final void dispatchStopEvent(StopEvent aStopEvent)
```

Iterates through the list of listeners and sends an event to each one. This is declared `protected` so that subclasses of this class can call it. However, it is also declared `final` so that it cannot be redefined in a subclass.

Parameters:
aStopEvent - the event to be dispatched to all listeners

- **getInstance**

```
public static jsb.core.StopTrigger getInstance(String aClassName)
        throws ClassLoadFailureException
```

Returns an instance of the named trigger. This is a convenience method which does type checking and translates any exception to a `ClassLoadFailureException`.

Parameters:
aClassName - the trigger to be instantiated
Throws: ClassLoadFailureException
if no instance of the named class can be constructed, or if the named class is not a trigger

- **removeStopListener**

```
public void removeStopListener(StopListener aListener)
```

Removes a listener so that `StopEvents` are no longer sent to it.

Parameters:
aListener - a StopListener to be removed from this trigger

Class jsb.core.StopTriggerOp

```
Object
   |
   +----jsb.core.StopTriggerOp
```

public class **StopTriggerOp**
extends Object

A logical operator that relates two `StopTriggers`. This is used when creating a `CompositeStopTrigger`.

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)
See Also:
StopTrigger, CompositeStopTrigger

Variable Index

- **AND_OP**
  Indicates that the composite trigger should be set off only when both of its subtriggers have gone off.
- **OR_OP**
  Indicates that the composite trigger should be set off when either of its two subtriggers goes off.

Constructor Index

- **jsb.core.StopTriggerOp**(String)
  Constructs a binary operator based on a given tag.

Method Index

- **isAndOp**()
  Returns `true` if the composite trigger that uses this operator should be set off only when both of its subtriggers have gone off.
- **isOrOp**()
  Returns `true` if the composite trigger that uses this operator should be set off when either of its two subtriggers goes off.
- **toString**()
  Returns the string representation of the operator.

Variables

- **AND_OP**

```
public static final java.lang.String AND_OP
```

Indicates that the composite trigger should be set off only when both of its subtriggers have gone off.

- **OR_OP**

```
public static final java.lang.String OR_OP
```

Indicates that the composite trigger should be set off when either of its two subtriggers goes off.

Constructors

- **StopTriggerOp**

```
public StopTriggerOp(String aString) throws UndefinedException
```

Constructs a binary operator based on a given tag. If the tag equals `AND_OP`, the operator is an "and" operator. If it equals `OR_OP`, the operator is an "or" operator.

Parameters:
aString - the tag that defines the binary operator
Throws: UndefinedException
if the tag is not equal to either `AND_OP` or `OR_OP`

Methods

- **isAndOp**

```
public boolean isAndOp()
```

Returns `true` if the composite trigger that uses this operator should be set off only when both of its subtriggers have gone off. Returns `false` if this is an "or" operator.

Returns:
`true` if this operator represents a logical "and", `false` otherwise

- **isOrOp**

`public boolean isOrOp()`

Returns `true` if the composite trigger that uses this operator should be set off when either of its two subtriggers goes off. Returns `false` if this is an "and" operator.

Returns:
`true` if this operator represents a logical "or", `false` otherwise

- **toString**

`public java.lang.String toString()`

Returns the string representation of the operator.

Overrides:
toString in class Object

Class jsb.core.ClassLoadFailureException

```
Object
   |
   +----Throwable
           |
           +----Exception
                   |
                   +----jsb.core.ClassLoadFailureException
```

public class **ClassLoadFailureException**
extends Exception

A set of class loading exceptions, each differentiated by its error code.

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)

Variable Index

- **CANNOT_INSTANTIATE**
  Indicates that it is not possible to create an instance of the class to be loaded.
- **CLASS_NOT_FOUND**
  Indicates that no class definition can be found for the class to be loaded.
- **WRONG_CLASS_TYPE**
  Indicates that the class to be loaded does not derive from a required class, or does not implement a required interface.

Constructor Index

- **jsb.core.ClassLoadFailureException**(int, String)
  Constructs a class loading exception with the given error code and message.
- **jsb.core.ClassLoadFailureException**(int)
  Constructs a class loading exception with the given error code.

Method Index

- **getErrorCode**()
  Returns the error code, which can be used for error recovery purposes.

Variables

- **CANNOT_INSTANTIATE**

```
public static final int CANNOT_INSTANTIATE
```

Indicates that it is not possible to create an instance of the class to be loaded. This could occur because of access restrictions, or because the attempt to create an instance caused an exception.

- **CLASS_NOT_FOUND**

```
public static final int CLASS_NOT_FOUND
```

Indicates that no class definition can be found for the class to be loaded.

- **WRONG_CLASS_TYPE**

```
public static final int WRONG_CLASS_TYPE
```

Indicates that the class to be loaded does not derive from a required class, or does not implement a required interface.

Constructors

- **ClassLoadFailureException**

```
public ClassLoadFailureException(int anErrorCode,
                                 String aMessage)
```

Constructs a class loading exception with the given error code and message.

Parameters:
anErrorCode - an int that defines the type of error
aMessage - a message that can be used to display the error

- **ClassLoadFailureException**

```
public ClassLoadFailureException(int anErrorCode)
```

Constructs a class loading exception with the given error code.

Parameters:
anErrorCode - an int that defines the type of error

Methods

- **getErrorCode**

```
public int getErrorCode()
```

Returns the error code, which can be used for error recovery purposes.

Returns:
an int that defines the type of error

Class jsb.core.UndefinedException

```
Object
   |
   +----Throwable
           |
           +----Exception
                   |
                   +----jsb.core.UndefinedException
```

public class **UndefinedException**
extends Exception

A set of exceptions that may be thrown when `SequenceObjects` are defined. Each indicates that incomplete or erroneous information was provided, such that the object could not be created. The exceptions are differentiated by error code.

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)
See Also:
SequenceObject

Variable Index

- **BAD_PARAM**
  Indicates that an unknown parameter was passed to a `ParameterizedObject`.
- **ILLEGAL_PARAM_VALUE**
  Indicates that an illegal value was passed to a `ParameterizedObject`.
- **NO_CHILDREN**
  Indicates that no children were listed when a `CompositeSequenceObject` was defined.
- **NO_CYCLES**
  Indicates that the number of cycles was not defined for a `CompositeSequenceObject`.
- **NO_DRIVER**
  Indicates that no instance of the driver class was provided.
- **NO_ID**
  Indicates that the ID tag was missing.

- STOPTRIGGER_MISMATCH
  Indicates that the `StopTrigger` specified was inappropriate for the type of `SequenceObject` defined.
- UNKNOWN_TRIGGEROP
  Indicates that an unknown `StopTriggerOp` was used.

Constructor Index

- jsb.core.UndefinedException(int, String)
  Constructs an `UndefinedException` with the given error code and message.
- jsb.core.UndefinedException(int)
  Constructs an `UndefinedException` with the given error code.

Method Index

- getErrorCode()
  Returns the error code, which can be used for error recovery purposes.

Variables

- **BAD_PARAM**

```
public static final int BAD_PARAM
```

Indicates that an unknown parameter was passed to a `ParameterizedObject`.

See Also:
ParameterizedObject

- **ILLEGAL_PARAM_VALUE**

```
public static final int ILLEGAL_PARAM_VALUE
```

Indicates that an illegal value was passed to a `ParameterizedObject`.

See Also:
ParameterizedObject

- **NO_CHILDREN**

```
public static final int NO_CHILDREN
```

Indicates that no children were listed when a `CompositeSequenceObject` was defined.

See Also:
CompositeSequenceObject

- **NO_CYCLES**

```
public static final int NO_CYCLES
```

Indicates that the number of cycles was not defined for a `CompositeSequenceObject`.

See Also:
CompositeSequenceObject

- **NO_DRIVER**

```
public static final int NO_DRIVER
```

Indicates that no instance of the driver class was provided.

See Also:
SequenceDriver

- **NO_ID**

```
public static final int NO_ID
```

Indicates that the ID tag was missing.

- **STOPTRIGGER_MISMATCH**

```
public static final int STOPTRIGGER_MISMATCH
```

Indicates that the `StopTrigger` specified was inappropriate for the type of `SequenceObject` defined.

See Also:
StopTrigger, SequenceObject

- **UNKNOWN_TRIGGEROP**

```
public static final int UNKNOWN_TRIGGEROP
```

Indicates that an unknown `StopTriggerOp` was used.

See Also:
StopTriggerOp

Constructors

- **UndefinedException**

```
public UndefinedException(int anErrorCode,
                          String aMessage)
```

Constructs an `UndefinedException` with the given error code and message.

Parameters:
anErrorCode - an int that defines the type of error
aMessage - a message that can be used to display the error

- **UndefinedException**

```
public UndefinedException(int anErrorCode)
```

Constructs an `UndefinedException` with the given error code.

Parameters:
anErrorCode - an int that defines the type of error

Methods

- **getErrorCode**

```
public int getErrorCode()
```

Returns the error code, which can be used for error recovery purposes.

Returns:
an int that defines the type of error

package jsb.gui

Class Index

- BasicObjectGui
- CommitListener
- CompositeObjectGui
- DataFrame
- ParamsTable
- ParamsTableModel
- SequenceGui
- StopTriggerPanel

Class jsb.gui.BasicObjectGui

```
Object
   |
   +----Component
           |
           +----Container
                   |
                   +----JComponent
                           |
                           +----JInternalFrame
                                   |
                                   +----jsb.gui.BasicObjectGui
```

public class **BasicObjectGui**
extends JInternalFrame

A Swing-compatible frame that can be used for creating, displaying, and editing a `BasicSequenceObject`.

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)
See Also:
BasicSequenceObject

Variable Index

- **driverComboBox**
  The field used to display the currently selected driver.
- **idField**
  The field that is used to display the ID of the sequence object.
- **okButton**
  The button used to confirm that the data is complete and the sequence object should be entered into a sequence.
- **theDriver**
  The currently selected driver.
- **theParamsTable**
  The table used to display the parameters of the current driver.
- **theStopTriggerPanel**
  The panel used to display the stop trigger of the sequence object.

Constructor Index

- **jsb.gui.BasicObjectGui**()
  Constructs a GUI that can be used for creating a `BasicSequenceObject`.
- **jsb.gui.BasicObjectGui**(BasicSequenceObject)
  Constructs a GUI that can be used to display and edit a `BasicSequenceObject`.

Method Index

- **addCommitListener**(CommitListener)
  Adds a listener that will be called when the OK button is pressed.
- **getBasicSequenceObject**()
  Returns the object displayed in this GUI.

Variables

- **driverComboBox**

```
final javax.swing.JComboBox driverComboBox
```

The field used to display the currently selected driver.

- **idField**

```
final javax.swing.JTextField idField
```

The field that is used to display the ID of the sequence object.

- **okButton**

```
final javax.swing.JButton okButton
```

The button used to confirm that the data is complete and the sequence object should be entered into a sequence.

- **theDriver**

```
jsb.core.SequenceDriver theDriver
```

The currently selected driver.

- **theParamsTable**

```
final jsb.gui.ParamsTable theParamsTable
```

The table used to display the parameters of the current driver.

- **theStopTriggerPanel**

```
jsb.gui.StopTriggerPanel theStopTriggerPanel
```

The panel used to display the stop trigger of the sequence object.

Constructors

- **BasicObjectGui**

```
public BasicObjectGui() throws ClassLoadFailureException
```

Constructs a GUI that can be used for creating a `BasicSequenceObject`.

Throws: ClassLoadFailureException
if a driver cannot be loaded

- **BasicObjectGui**

```
public BasicObjectGui(BasicSequenceObject aBasicSequenceObject)
        throws ClassLoadFailureException
```

Constructs a GUI that can be used to display and edit a `BasicSequenceObject`.

Parameters:
aBasicSequenceObject - the object to be edited
Throws: ClassLoadFailureException
if a driver cannot be loaded

Methods

- **addCommitListener**

```
public void addCommitListener(CommitListener aCommitListener)
```

Adds a listener that will be called when the OK button is pressed. The listener should add the object displayed in this GUI to a sequence.

Parameters:
aCommitListener - a listener which will respond when the data is confirmed and should be retrieved

- **getBasicSequenceObject**

```
jsb.core.BasicSequenceObject getBasicSequenceObject()
        throws UndefinedException
```

Returns the object displayed in this GUI.

Returns:
the object displayed
Throws: UndefinedException
if the object is not fully defined, or if some of the data is invalid

Class jsb.gui.CommitListener

```
Object
   |
   +----jsb.gui.CommitListener
```

public class **CommitListener**
extends Object
implements ActionListener

A listener that will retrieve the data from a temporary edit frame and store that data in the main `DataFrame`. It listens for "confirm" events from the temporary frame (such as pressing an OK button).

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)

Constructor Index

- **jsb.gui.CommitListener**(JInternalFrame, DataFrame)
  Constructs a listener which will pull data from a temporary edit frame when requested and store it within a master data storage object.

Method Index

- **actionPerformed**(ActionEvent)

Constructors

- **CommitListener**

```
public CommitListener(JInternalFrame anInternalFrame,
                      DataFrame aMasterFrame)
```

Constructs a listener which will pull data from a temporary edit frame when requested and store it within a master data storage object.

Parameters:
anInternalFrame - a temporary edit frame
aMasterFrame - the master application frame

Methods

- **actionPerformed**

```
public void actionPerformed(ActionEvent anEvent)
```

Class jsb.gui.CompositeObjectGui

```
Object
   |
   +----Component
           |
           +----Container
                   |
                   +----JComponent
                           |
                           +----JInternalFrame
                                   |
                                   +----jsb.gui.CompositeObjectGui
```

public class **CompositeObjectGui**
extends JInternalFrame

A Swing-compatible frame that can be used for creating, displaying, and editing a `CompositeSequenceObject`.

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)
See Also:
CompositeSequenceObject

Variable Index

- **childAddButton**
  The button used to enter a new child ID.
- **childList**
  The list that is used to display the IDs of the child objects.
- **childListData**
  The list of child IDs.
- **childNameField**
  The field used to enter a new child ID.
- **childRemoveButton**
  The button used to remove a child ID from the list.
- **idField**
  The field that is used to display the ID of the sequence object.
- **managerComboBox**
  The field used to display the currently selected manager.

- **okButton**
  The button used to confirm that the data is complete and the sequence object should be entered into a sequence.
- **theManager**
  The currently selected manager.
- **theParamsTable**
  The table used to display the parameters of the current manager.
- **theStopTriggerPanel**
  The panel used to display the stop trigger of the sequence object.

Constructor Index

- **jsb.gui.CompositeObjectGui**()
  Constructs a GUI that can be used for creating a `CompositeSequenceObject`.
- **jsb.gui.CompositeObjectGui**(CompositeSequenceObject)
  Constructs a GUI that can be used to display and edit a `CompositeSequenceObject`.

Method Index

- **addCommitListener**(CommitListener)
  Adds a listener that will be called when the OK button is pressed.
- **getCompositeSequenceObject**()
  Returns the object displayed in this GUI.

Variables

- **childAddButton**

```
javax.swing.JButton childAddButton
```

The button used to enter a new child ID.

- **childList**

```
javax.swing.JList childList
```

The list that is used to display the IDs of the child objects.

- **childListData**

```
java.util.Vector childListData
```

The list of child IDs.

- **childNameField**

```
final javax.swing.JTextField childNameField
```

The field used to enter a new child ID.

- **childRemoveButton**

`javax.swing.JButton childRemoveButton`

The button used to remove a child ID from the list.

- **idField**

`javax.swing.JTextField idField`

The field that is used to display the ID of the sequence object.

- **managerComboBox**

`final javax.swing.JComboBox managerComboBox`

The field used to display the currently selected manager.

- **okButton**

`javax.swing.JButton okButton`

The button used to confirm that the data is complete and the sequence object should be entered into a sequence.

- **theManager**

`jsb.core.SetManager theManager`

The currently selected manager.

- **theParamsTable**

`jsb.gui.ParamsTable theParamsTable`

The table used to display the parameters of the current manager.

- **theStopTriggerPanel**

`jsb.gui.StopTriggerPanel theStopTriggerPanel`

The panel used to display the stop trigger of the sequence object.

Constructors

- **CompositeObjectGui**

`public CompositeObjectGui() throws ClassLoadFailureException`

Constructs a GUI that can be used for creating a `CompositeSequenceObject`.

Throws: ClassLoadFailureException
if a manager cannot be loaded

- **CompositeObjectGui**

```
public CompositeObjectGui
        (CompositeSequenceObject aCompositeSequenceObject)
        throws ClassLoadFailureException
```

Constructs a GUI that can be used to display and edit a `CompositeSequenceObject`.

Parameters:
aCompositeSequenceObject - the object to be edited
Throws: ClassLoadFailureException
if a manager cannot be loaded

Methods

- **addCommitListener**

```
public void addCommitListener(CommitListener aCommitListener)
```

Adds a listener that will be called when the OK button is pressed. The listener should add the object displayed in this GUI to a sequence.

Parameters:
aCommitListener - a listener which will respond when the data is confirmed and should be retrieved

- **getCompositeSequenceObject**

```
jsb.core.CompositeSequenceObject getCompositeSequenceObject()
        throws UndefinedException
```

Returns the object displayed in this GUI.

Returns:
the object displayed
Throws: UndefinedException
if the object is not fully defined, or if some of the data is invalid

Class jsb.gui.DataFrame

```
Object
   |
   +----Component
           |
           +----Container
                   |
                   +----Window
                           |
                           +----Frame
                                   |
                                   +----JFrame
                                           |
                                           +----jsb.gui.DataFrame
```

public class **DataFrame**
extends JFrame

Extends `JFrame` with an object-storing mechanism. If a menu is provided by an outside class, the data that is stored within this class can be manipulated using the menu.

This class is currently used only for `SequenceObjects` and the `JInternalFrames` associated with them. However, this could easily be modified to be used as a general-purpose class.

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)

Constructor Index

- **jsb.gui.DataFrame**()
  Constructs a window that can contain GUI sub-elements.

Method Index

- **elements**()
  Returns a list of all of the sequence objects, used for iteration.
- **get**(Object)
  Returns the sequence object associated with the given key.
- **getAllElements**()
  Returns the full data table.

- keys()
  Returns a list of all of the data keys that may be used for iteration.
- put(Object, Object, JInternalFrame)
  Stores the given objects, associates them with the given key, and returns the sequence object previously associated with that key (if any).
- remove(Object)
  Removes the objects associated with the given key, and returns the `SequenceObject` that was associated with the key (if any).
- setObjectMenu(JMenu)
  Sets the object menu.

Constructors

- **DataFrame**

```
public DataFrame()
```

Constructs a window that can contain GUI sub-elements.

The current implementation is based on a JDesktopPane.

Methods

- **elements**

```
java.util.Enumeration elements()
```

Returns a list of all of the sequence objects, used for iteration.

Returns:
an enumerated list of the `SequenceObjects` stored

- **get**

```
java.lang.Object get(Object aKey)
```

Returns the sequence object associated with the given key.

Parameters:
aKey - an ID that identifies a particular object
Returns:
the `SequenceObject` associated with the ID

- **getAllElements**

```
java.util.Hashtable getAllElements()
```

Returns the full data table.

Returns:
a table of the `SequenceObjects` stored

- **keys**

```
java.util.Enumeration keys()
```

Returns a list of all of the data keys that may be used for iteration.

Returns:
an enumerated list of the keys of the objects stored

- **put**

```
java.lang.Object put(Object aKey,
                     Object aValue,
                     JInternalFrame aGui)
```

Stores the given objects, associates them with the given key, and returns the sequence object previously associated with that key (if any).

Parameters:
aKey - an ID that will be used to identify a particular `SequenceObject` and its GUI representation
aValue - the `SequenceObject` associated with the ID
aGui - the `JInternalFrame` associated with the ID
Returns:
the `SequenceObject` previously associated with the ID, or `null` if the ID was not associated with any object

- **remove**

```
java.lang.Object remove(Object key)
```

Removes the objects associated with the given key, and returns the `SequenceObject` that was associated with the key (if any).

Parameters:
aKey - an ID that will be identifies a particular `SequenceObject` and its GUI representation
Returns:
the `SequenceObject` associated with aKey, or `null` if aKey was not associated with any object

- **setObjectMenu**

```
public void setObjectMenu(JMenu anObjectMenu)
```

Sets the object menu.

Parameters:
anObjectMenu - a Swing-compatible menu that shows the objects available, provides a way to access them, and may also provide ways to manipulate them

Class jsb.gui.ParamsTable

```
Object
   |
   +----Component
           |
           +----Container
                   |
                   +----JComponent
                           |
                           +----JTable
                                   |
                                   +----jsb.gui.ParamsTable
```

class **ParamsTable**
extends JTable

A panel that displays a set of parameters and their values in a tabular format.

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)

Note: *This class is not public and therefore cannot be used outside this package.*

Constructor Index

- **jsb.gui.ParamsTable**()
  Constructs a table that can be used to display parameters and their associated values.

Method Index

- **getValues**()
  Returns a table that contains all the parameters currently displayed and their associated values.
- **setValues**(Hashtable)
  Sets the parameters to be displayed and their values.

Constructors

- **ParamsTable**

```
public ParamsTable()
```

Constructs a table that can be used to display parameters and their associated values.

Methods

- **getValues**

```
public java.util.Hashtable getValues()
```

Returns a table that contains all the parameters currently displayed and their associated values.

Returns:
a table of parameters and their current values

- **setValues**

```
public void setValues(Hashtable aValueTable)
```

Sets the parameters to be displayed and their values.

Parameters:
aValueTable - a table that contains the params to be displayed and their current values

Class jsb.gui.ParamsTableModel

```
Object
   |
   +----AbstractTableModel
           |
           +----jsb.gui.ParamsTableModel
```

class **ParamsTableModel**
extends AbstractTableModel

Note: *This class is not public and therefore cannot be used outside this package.*

Constructor Index

- **jsb.gui.ParamsTableModel**()

Method Index

- **getColumnCount**()
- **getColumnName**(int)
- **getRowCount**()
- **getValueAt**(int, int)
- **getValues**()
  Returns a table that contains all the displayed parameters and their associated values.
- **isCellEditable**(int, int)
- **setValueAt**(Object, int, int)
- **setValues**(Hashtable)
  Initializes the table with a set of parameters and their values.

Constructors

- **ParamsTableModel**

```
ParamsTableModel()
```

Methods

- **getColumnCount**

```
public int getColumnCount()
```

Overrides:
getColumnCount in class AbstractTableModel

- **getColumnName**

```
public java.lang.String getColumnName(int column)
```

Overrides:
getColumnName in class AbstractTableModel

- **getRowCount**

```
public int getRowCount()
```

Overrides:
getRowCount in class AbstractTableModel

- **getValueAt**

```
public java.lang.Object getValueAt(int row,
                                   int column)
```

Overrides:
getValueAt in class AbstractTableModel

- **getValues**

```
public java.util.Hashtable getValues()
```

Returns a table that contains all the displayed parameters and their associated values.

Returns:
a table of params and their values

- **isCellEditable**

```
public boolean isCellEditable(int row,
                              int column)
```

Overrides:
isCellEditable in class AbstractTableModel

- **setValueAt**

```
public void setValueAt(Object value,
                       int row,
                       int column)
```

Overrides:
setValueAt in class AbstractTableModel

- **setValues**

```
public void setValues(Hashtable aParamsTable)
```

Initializes the table with a set of parameters and their values.

Parameters:
aParamsTable - a table of params and their values

Class jsb.gui.SequenceGui

```
Object
   |
   +----jsb.gui.SequenceGui
```

public class **SequenceGui**
extends Object
implements SequenceFormat

A GUI format for representing sequences.

Version:
August 1999

Author:
Jacqueline Beaulac (Faculty of Music, McGill University)

Constructor Index

- **jsb.gui.SequenceGui**()
  Constructs a new GUI using a plain vanilla `DataFrame`.

Method Index

- **addBasicSequenceObject**(BasicSequenceObject)
  Adds a `BasicSequenceObject` to this GUI representation of a sequence.
- **addCompositeSequenceObject**(CompositeSequenceObject)
  Adds a `CompositeSequenceObject` to this GUI representation of a sequence.
- **addElement**(SequenceObject)
  Adds a section to this GUI representation of a sequence.
- **elements**()
  Returns an enumerated list of all the sections.
- **getAllElements**()
  Returns a table of all the sections, keyed by ID.
- **getFormattedSequence**()
  Returns the constructed sequence in GUI format.
- **removeElement**(Object)
  Removes a section from this GUI representation of a sequence.
- **setFormattedSequence**(Object)
  Sets the main GUI window.

Constructors

- **SequenceGui**

```
public SequenceGui()
```

Constructs a new GUI using a plain vanilla `DataFrame`.

See Also:
DataFrame

Methods

- **addBasicSequenceObject**

```
void addBasicSequenceObject
        (BasicSequenceObject aBasicSequenceObject)
        throws ClassLoadFailureException
```

Adds a `BasicSequenceObject` to this GUI representation of a sequence. This helper method is used by the `addElement` method.

Parameters:
aBasicSequenceObject - a basic object to be added to the current sequence
See Also:
addElement, BasicObjectGui

- **addCompositeSequenceObject**

```
void addCompositeSequenceObject
        (CompositeSequenceObject aCompositeSequenceObject)
        throws ClassLoadFailureException
```

Adds a `CompositeSequenceObject` to this GUI representation of a sequence. This helper method is used by the `addElement` method.

Parameters:
aCompositeSequenceObject - a composite object to be added to the current sequence
See Also:
addElement, CompositeObjectGui

- **addElement**

```
public void addElement(SequenceObject aSequenceObject)
        throws ClassLoadFailureException
```

Adds a section to this GUI representation of a sequence. This method delegates to helper methods for each implementation of `SequenceObject` supported in this format.

If new implementations of `SequenceObject` are to be supported by a subclass, a new helper method can be added for each new implementation. This method can then be overriden to delegate to the new helper methods.

Parameters:
aSequenceObject - a section to be added to the current sequence

- **elements**

```
public java.util.Enumeration elements()
```

Returns an enumerated list of all the sections.

Returns:
a list of all sections in the sequence

- **getAllElements**

```
public java.util.Hashtable getAllElements()
```

Returns a table of all the sections, keyed by ID.

Returns:
a table of all sections

- **getFormattedSequence**

```
public java.lang.Object getFormattedSequence()
```

Returns the constructed sequence in GUI format.

Returns:
the GUI representation of this sequence

- **removeElement**

```
public jsb.core.SequenceObject removeElement(Object aKey)
```

Removes a section from this GUI representation of a sequence.

Parameters:
aKey - a ID to be used to identify the section to be removed from the current sequence

- **setFormattedSequence**

```
public void setFormattedSequence(Object aSequence)
```

Sets the main GUI window.

Parameters:
aSequence - a `DataFrame` to be used as the main window and data holder for this sequence

See Also:
DataFrame

Class jsb.gui.StopTriggerPanel

```
Object
   |
   +----Component
           |
           +----Container
                   |
                   +----JComponent
                           |
                           +----JPanel
                                   |
                                   +----jsb.gui.StopTriggerPanel
```

public class **StopTriggerPanel**
extends JPanel

A Swing-compatible GUI panel that can be used for creating, displaying, and editing a `StopTrigger`.

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)
See Also:
StopTrigger

Constructor Index

- **jsb.gui.StopTriggerPanel**()
  Constructs a panel which may be used to create a new `StopTrigger`.

Method Index

- **getValue**()
  Returns the trigger which is being edited.
- **setValue**(StopTrigger)
  Sets the trigger data which should be shown and edited in the panel.

Constructors

- **StopTriggerPanel**

`public StopTriggerPanel() throws ClassLoadFailureException`

Constructs a panel which may be used to create a new `StopTrigger`.

Throws: ClassLoadFailureException
if any of the classes in `triggerChoiceList` cannot be loaded

Methods

- **getValue**

`public jsb.core.StopTrigger getValue() throws UndefinedException`

Returns the trigger which is being edited.

Returns:
the stop trigger currently shown

- **setValue**

`public void setValue(StopTrigger aStopTrigger)`

Sets the trigger data which should be shown and edited in the panel.

Parameters:
aStopTrigger - the trigger to be edited

package jsb.text

Class Index

- SequenceText

Exception Index

- BadParseException

Class jsb.text.SequenceText

```
Object
   |
   +----jsb.text.SequenceText
```

public class **SequenceText**
extends Object
implements SequenceFormat

A text format for representing sequences. If new implementations of `SequenceObject` are provided, this class can be extended to support them. (All of the internal methods used in the class are declared `protected` rather than `private` in order to allow easier subclassing.)

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)
See Also:
SequenceObject

Constructor Index

- **jsb.text.SequenceText**()

Method Index

- **addBasicSequenceObject**(StringBuffer, BasicSequenceObject)
  Adds a `BasicSequenceObject` to this text representation of a sequence.
- **addCompositeSequenceObject**(StringBuffer, CompositeSequenceObject)
  Adds a `CompositeSequenceObject` to this text representation of an sequence.
- **addElement**(SequenceObject)
  Adds a section to this text representation of a sequence.
- **addParams**(StringBuffer, Hashtable)
  Adds a set of parameters to this text representation of a sequence.
- **addStopTrigger**(StringBuffer, StopTrigger)
  Adds a stop trigger to this text representation of a sequence.
- **elements**()
  Returns an enumeration of all the sections.

- firstParse(Object)
  Parses the contents of a stream into individual string tokens, removing C-style comments.
- getAllElements()
  Returns a table of all the sections, keyed by ID.
- getFormattedSequence()
  Returns the constructed sequence in text format.
- getNextToken(Enumeration, int)
  Returns the next token in the given `Enumeration`, or throws the given exception.
- nextBasicSequenceObject(Enumeration)
  Parses a `BasicSequenceObject` out of the text representation.
- nextCompositeSequenceObject(Enumeration, String)
  Parses a `CompositeSequenceObject` out of the text representation.
- nextParams(Enumeration)
  Parses a set of parameters out of the text representation.
- nextStopTrigger(Enumeration)
  Parses a `StopTrigger` out of the text representation.
- removeElement(Object)
  Removes a section from this text representation of a sequence, based on the ID of the section.
- setFormattedSequence(Object)
  Sets the current sequence and parses it from text format.

Constructors

- **SequenceText**

```
public SequenceText()
```

Methods

- **addBasicSequenceObject**

```
protected void addBasicSequenceObject
        (StringBuffer textRepresentation,
         BasicSequenceObject aBasicSequenceObject)
```

Adds a `BasicSequenceObject` to this text representation of a sequence. This helper method is used by `getFormattedSequence()`.

Parameters:
textRepresentation - the current text holder
aBasicSequenceObject - a basic object to be added to the current sequence
See Also:
getFormattedSequence

- **addCompositeSequenceObject**

```
protected void addCompositeSequenceObject
        (StringBuffer textRepresentation,
         CompositeSequenceObject aCompositeSequenceObject)
```

Adds a `CompositeSequenceObject` to this text representation of an sequence. This helper method is used by `getFormattedSequence()`.

Parameters:
textRepresentation - the current text holder
aCompositeSequenceObject - a composite object to be added to the current sequence

See Also:
getFormattedSequence

- **addElement**

```
public void addElement(SequenceObject aSequenceObject)
```

Adds a section to this text representation of a sequence. This method delegates to helper methods for each implementation of `SequenceObject` supported in this format.

If new implementations of `SequenceObject` are to be supported by a subclass, a new helper method can be added for each new implementation. This method can then be overriden to delegate to the new helper methods.

Parameters:
aSequenceObject - an object to be added to the current sequence

- **addParams**

```
protected void addParams(StringBuffer textRepresentation,
                         Hashtable aParamTable)
```

Adds a set of parameters to this text representation of a sequence.

Parameters:
textRepresentation - the current text holder
aParamTable - a table containing a set of parameters and their values, to be added to the current sequence

- **addStopTrigger**

```
protected void addStopTrigger(StringBuffer textRepresentation,
                              StopTrigger aStopTrigger)
```

Adds a stop trigger to this text representation of a sequence.

Parameters:
textRepresentation - the current text holder
aStopTrigger - a stop trigger to be added to the current sequence

- **elements**

```
public java.util.Enumeration elements()
```

Returns an enumeration of all the sections.

Returns:
an enumerated list of all `SequenceObjects` in this sequence

- **firstParse**

```
protected java.util.Enumeration firstParse(Object aSequence)
        throws BadParseException
```

Parses the contents of a stream into individual string tokens, removing C-style comments.

Parameters:
aSequence - a reader from which the text tokens will be read
Throws: BadParseException
if an I/O error occurs while reading the stream

- **getAllElements**

```
public java.util.Hashtable getAllElements()
```

Returns a table of all the sections, keyed by ID.

Returns:
a table of all sections

- **getFormattedSequence**

```
public java.lang.Object getFormattedSequence()
```

Returns the constructed sequence in text format. This method delegates to helper methods for each implementation of `SequenceObject` supported in this format.

If new implementations of `SequenceObject` are to be supported by a subclass, a new helper method can be added for each new implementation. This method can then be overriden to delegate to the new helper methods.

Returns:
a `StringBuffer` that contains the text representation of this sequence

- **getNextToken**

```
protected java.lang.String getNextToken(Enumeration textTokens,
                                        int errorCode)
        throws BadParseException
```

Returns the next token in the given `Enumeration`, or throws the given exception. This is a convenience method, since this operation is made many times within the parsing methods.

Parameters:
textTokens - an enumerated list of strings
errorCode - an int that indicates the type of `BadParseException` to be thrown if the end of the list of tokens has been reached
Throws: BadParseException
if no more tokens in the enumeration

- **nextBasicSequenceObject**

```
protected jsb.core.BasicSequenceObject nextBasicSequenceObject
        (Enumeration textTokens)
        throws BadParseException, UndefinedException,
               ClassLoadFailureException
```

Parses a `BasicSequenceObject` out of the text representation. This helper method is used by `setFormattedSequence()`.

Returns:
the next basic object defined in the text
Throws: BadParseException
if a parsing error occurs
Throws: UndefinedException
if some values are missing
Throws: ClassLoadFailureException
if the driver class is invalid
See Also:
setFormattedSequence, BasicSequenceObject

- **nextCompositeSequenceObject**

```
protected jsb.core.CompositeSequenceObject
        nextCompositeSequenceObject(Enumeration textTokens,
                                    String currToken)
        throws BadParseException, UndefinedException,
               ClassLoadFailureException
```

Parses a `CompositeSequenceObject` out of the text representation. This helper method is used by `setFormattedSequence()`.

Returns:
the next composite object defined in the text
Throws: BadParseException
if a parsing error occurs
Throws: UndefinedException
if some values are missing
Throws: ClassLoadFailureException
if the driver class is invalid
See Also:
setFormattedSequence, CompositeSequenceObject

- **nextParams**

```
protected java.util.Hashtable nextParams(Enumeration textTokens)
        throws BadParseException
```

Parses a set of parameters out of the text representation.

Returns:
a table of parameters and their associated values
Throws: BadParseException
if a parsing error occurs

- **nextStopTrigger**

```
protected jsb.core.StopTrigger nextStopTrigger
        (Enumeration textTokens)
        throws BadParseException, UndefinedException,
               ClassLoadFailureException
```

Parses a `StopTrigger` out of the text representation.

Returns:
a stop trigger
Throws: BadParseException
if a parsing error occurs
Throws: UndefinedException
if some values are missing
Throws: ClassLoadFailureException
if the driver class is invalid

- **removeElement**

```
public jsb.core.SequenceObject removeElement(Object aKey)
```

Removes a section from this text representation of a sequence, based on the ID of the section.

Parameters:
aKey - the ID of the `SequenceObject` to be removed from the current sequence
Returns:
the `SequenceObject` that has just been removed

- **setFormattedSequence**

```
public void setFormattedSequence(Object aSequence)
        throws BadParseException, UndefinedException,
               ClassLoadFailureException
```

Sets the current sequence and parses it from text format. This method delegates to helper methods for each implementation of `SequenceObject` supported in this format.

If new implementations of `SequenceObject` are to be supported by a subclass, a new helper method can be added for each new implementation. This method can then be overriden to delegate to the new helper methods.

Parameters:
aSequence - an InputStreamReader from which the text script will be read
Throws: BadParseException
if a parsing error occurs
Throws: UndefinedException
if a parsed object is badly defined
Throws: ClassLoadFailureException
if a class used in a parsed object cannot be loaded

Class jsb.text.BadParseException

```
Object
   |
   +----Throwable
           |
           +----Exception
                   |
                   +----jsb.text.BadParseException
```

public class **BadParseException**
extends Exception

A set of text parsing exceptions, each differentiated by its error code.

Version:
August 1999
Author:
Jacqueline Beaulac (Faculty of Music, McGill University)

Variable Index

- **DUPLICATE_DEF**
  Indicates that an element was defined twice in the same block.
- **MISSING_DRIVER**
  Indicates that the name of a driver class was missing.
- **MISSING_ID**
  Indicates that ID information was missing.
- **MISSING_STOPTRIGGER**
  Indicates that a stop trigger was missing.
- **MISSING_TRIGGER_PARAMS**
  Indicates that the parameters for a stop trigger were missing.
- **STREAM_ERROR**
  Indicates that an error occured when reading a text stream.
- **UNCLOSED_PARAMS**
  Indicates that the closing bracket for a set of parameters was missing.
- **UNCLOSED_SET**
  Indicates that the closing bracket for a set of child IDs was missing.
- **UNCLOSED_STOPTRIGGER**
  Indicates that the closing bracket for a stop trigger was missing.
- **UNENDED_OBJECT**
  Indicates that the closing symbol for an object was missing.

- UNKNOWN_TAG
  Indicates that a word that does not correspond to any known keyword was used.
- UNMATCHED_PARAM
  Indicates that the value of a parameter was missing.

Constructor Index

- **jsb.text.BadParseException**(int, String)
  Constructs a parsing exception with the given error code and message.
- **jsb.text.BadParseException**(int)
  Constructs a parsing exception with the given error code.

Method Index

- **getErrorCode**()
  Returns the error code, which can be used for error recovery purposes.

Variables

- **DUPLICATE_DEF**

```
public static final int DUPLICATE_DEF
```

Indicates that an element was defined twice in the same block.

- **MISSING_DRIVER**

```
public static final int MISSING_DRIVER
```

Indicates that the name of a driver class was missing.

See Also:
SequenceDriver

- **MISSING_ID**

```
public static final int MISSING_ID
```

Indicates that ID information was missing.

- **MISSING_STOPTRIGGER**

```
public static final int MISSING_STOPTRIGGER
```

Indicates that a stop trigger was missing.

See Also:
StopTrigger

- **MISSING_TRIGGER_PARAMS**

```
public static final int MISSING_TRIGGER_PARAMS
```

Indicates that the parameters for a stop trigger were missing.

See Also:
StopTrigger

- **STREAM_ERROR**

```
public static final int STREAM_ERROR
```

Indicates that an error occured when reading a text stream.

- **UNCLOSED_PARAMS**

```
public static final int UNCLOSED_PARAMS
```

Indicates that the closing bracket for a set of parameters was missing.

See Also:
ParameterizedObject

- **UNCLOSED_SET**

```
public static final int UNCLOSED_SET
```

Indicates that the closing bracket for a set of child IDs was missing.

See Also:
CompositeSequenceObject

- **UNCLOSED_STOPTRIGGER**

```
public static final int UNCLOSED_STOPTRIGGER
```

Indicates that the closing bracket for a stop trigger was missing.

See Also:
StopTrigger

- **UNENDED_OBJECT**

```
public static final int UNENDED_OBJECT
```

Indicates that the closing symbol for an object was missing.

- **UNKNOWN_TAG**

```
public static final int UNKNOWN_TAG
```

Indicates that a word that does not correspond to any known keyword was used.

- **UNMATCHED_PARAM**

```
public static final int UNMATCHED_PARAM
```

Indicates that the value of a parameter was missing.

See Also:
ParameterizedObject

Constructors

- **BadParseException**

```
public BadParseException(int anErrorCode,
                         String aMessage)
```

Constructs a parsing exception with the given error code and message.

Parameters:
anErrorCode - an int that defines the type of error
aMessage - a message that can be used to display the error

- **BadParseException**

```
public BadParseException(int anErrorCode)
```

Constructs a parsing exception with the given error code.

Parameters:
anErrorCode - an int that defines the type of error

Methods

- **getErrorCode**

```
public int getErrorCode()
```

Returns the error code, which can be used for error recovery purposes.

Returns:
an int that defines the type of error