MusicKit Tutorials

David A. Jaffe

Edited by

Hans-Christoph Steiner

Leigh M. Smith

MusicKit Tutorials
by David A. Jaffe

Edited by Hans-Christoph Steiner

Edited by Leigh M. Smith

Table of Contents

List of Tables

List of Figures

Editors Preface

This book contains teaching materials for the CCRMA MusicKit class taught by David A. Jaffe in Jan-Feb 1991. This is an intensive class for programmers who already know the C Programming Language. The materials in this book may be distributed and used for teaching purposes. However, any publication is forbidden without the written consent of David A. Jaffe (reachable as <`daj@ccrma.stanford.edu`>).

The students should have access to the MusicKit and SndKit Concepts (http://musickit.sourceforge.net/MusicKitConcepts) Documentation. The students should be running V5.2.2 frameworks or later and should refer to the on-line documentation for the Reference section of the MusicKit and SndKit and the appropriate OpenStep system documentation.

Chapter 1. Class 1 — Object Oriented Programming in Objective-C

1.1. Objective-C

Objective-C is a language that expands the C programming language by incorporating three object-oriented concepts:

- Encapsulation
- Messaging (with dynamic binding)
- Inheritance

These serve to maximize:

- Program modularity
- Program clarity and readability
- Program maintainability

1.2. Encapsulation (motivation)

- Lets you create complex data types.
- Makes code easier to read.
- Makes function calls simpler.

C structures and typedefs already provide this functionality. Hypothetical example:

```
typedef struct {
 double freq;
 int keyNum;
} Note;
```

This struct Note is now a convenient package. To create a new Note, you just call `malloc`.

```
Note *myNote1;
myNote1 = malloc(sizeof(Note));
myNote1->freq = 440.0;
myNote1->keyNum = 69;
play(myNote1);
```

But C structs only go half way. Objective-C introduces the notion of a "Class" that encapsulates both the data *and the functions that operate on them*. This serves to protect the data and localize specialized knowledge of the data, making it harder to introduce bad bugs.

1.3. Messaging (motivation)

- Problem: Different C structs may require different functions to provide similar behavior. Hypothetical example:

  ```
  typedef struct {
       int keyNumber;
  } MIDINote;

  typedef struct {
     double freq;
  } DSPNote;

  MIDINote *aMidiNote;
  DSPNote *aDSPNote;

  /* ... (create and fill in fields of structs) */
  playMIDI(aMidiNote);
  playDSP(aDSPNote);
  ```

- We'd prefer a similar behavior to be represented by a single "message". Hypothetical example:

  ```
  play(aMidiNote);
  play(aDSPNote);
  ```

But this requires the writer of **play** to know every possible kind of Note it would be passed. This violates the principle of programming modularity.

- Objective-C lets each Class define its own **play** "method". The Objective-C run-time system then invokes the correct **play** method. This process is called "messaging".

1.4. Basic Objective-C Terminology

- **Class** - analogous to the **typedef** above; defines a complex data type and functions to operate on that data.
- **Instance of a class** - analogous to the pointer to the `malloc`ed memory above; each instance of a class has its own memory. In this memory, the instance stores the values of its "instance variables".
- **Instance variables** - analogous to the fields of the **struct**. The memory allocated for each instance is used to store that instance's variables.
- **method** - A "method" is a function associated with a class. A class may have any number of methods.
- **message** - A "message" is how a method is invoked. In C, you invoke a function by using its name followed by parens. In Objective-C, you invoke a method by sending a message to an instance, using the following syntax:

  ```
   [myInstance aMessage];
  or      [myInstance aMessageWithArg:arg];
  or      [myInstance aMessageWithArg:arg otherArg:arg2];
  ```

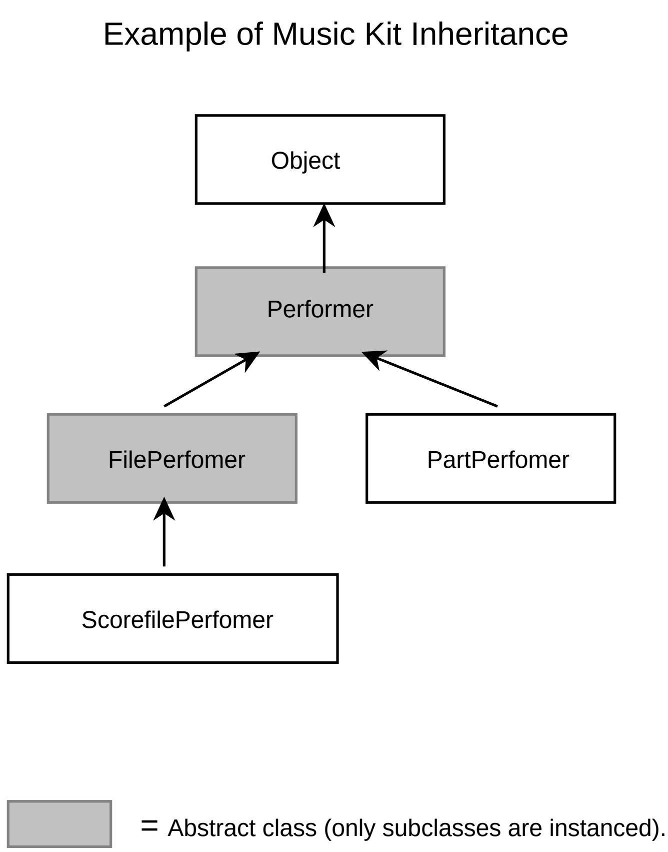
  This causes the Objective-C run-time system to look at the class of myInstance, find the correct method for the given message, and invoke that method.

1.5. Inheritance

- Allows a Class to be "specialized" into different versions. E.g.: `MKPerformer` is specialized to `MKPartPerformer` and `MKScorefilePerformer` in the MusicKit.

- A specialized class ("subclass") need only implement that part of its behavior that is different from its parent class ("superclass").
- A subclass may define instance variables. Each instance of the subclass gets the instance variables defined in both the superclass and the subclass.
- Subclassing may be applied recursively, forming trees of inheritance. All classes inherit from `NSObject`. Therefore, instances of a class are sometimes called "objects".
- Inheritance can get confusing. For this reason, the MusicKit uses it sparingly.
- If both the subclass and the superclass implement the same method, the subclass version takes precedence. However, the subclass can invoke the superclass version of a method as part of its own implementation of that method by sending the message to the special identifier **super**.

Figure 1-1. Example of MusicKit Inheritance



1.6. Various Details

- To create an instance, you need to send a message directly to a class. This invokes a "class method", which is different from an instance method in that it may be sent only to a class.
- To make it possible to send such a message, the Objective-C compiler creates a special "class object".
- You send **alloc** followed by **init** (for reasons that will not be covered here.)

  ```
  MKNote *aNote;

  aNote = [MKNote alloc];
  [aNote init];
  ```

- An object-valued variable (such as "aNote" above) need not have a type. To define an untyped object-valued variable, use the special type **id**. Example: **id** aNote;
- **self**, when used in a method definition, refers to the object that's receiving the message.
- Sometimes "class-wide" behavior is implemented by class methods. For example:

  ```
  [MKConductor startPerformance].
  ```

1.7. Example — Using Objective-C Classes

```
#include <MusicKit/MusicKit.h>

main()
{
      MKNote *aNote;
      MKPart *aPart;
      MKScore *aScore;
      aScore = [MKScore score];     /* In 2.0, use alloc/init */
      aPart = [MKPart new];
      aNote = [MKNote new];
      [aNote setPar: MK_freq toDouble: 440.0];
```

```
        [aNote setTimeTag: 1.0];      /* Play after 1 beat */
        [aNote setDur: 1.0];          /* Duration is 1 beat */
        [aScore addPart: aPart];
        [aPart addNote: aNote];

        /* "info" code may be added here - see below */
        [aScore writeScorefile: @"test.score"];
}
```

This example writes the file "test.score". However, if you want to play that file with the ScorePlayer application, you need to specify which `MKSynthPatch` (DSP instrument) to use. We do this by adding a special `MKNote` called a "`MKPart` info" with a parameter indicating the name of the `MKSynthPatch`:

```
aNote = [MKNote new];                              /* Another MKNote for info
[aNote setPar: MK_synthPatch toString: @"Pluck"];
[aPart setInfo: aNote];
```

1.8. Example — Defining An Objective-C Class

- You need a header file (.h) and a code file (.m)
- The header file defines the interface to the class, i.e. the instance variables and methods defined by the class. Instance methods begin with "-", class methods begin with "+". Example:

```
/* Here is an example .h file, "CircularList.h".
   All behavior is inherited from List, which defines a List of objects.
   Nowdays we can use NSArray which is more complex than List.
*/

#include <objc/List.h>  /* Superclass interface */

@interface CircularList: List /* List is superclass */
{
    int currentLocation;
}

- next; /* Returns next object in List or nil if none.  */
```

```
@end

/* Here is the corresponding .m file: */
#include "CircularList.h"

@implementation CircularList

- next
{
    int numObjects = [self count];
    if (currentLocation >= numObjects)
        currentLocation = 0;
    return [self objectAt:currentLocation++];
}

@end
```

1.9. OpenStep Software Kits

- Application Kit — Contains `NSButtons`, `NSSliders`, and other user interface classes. Best used with the Interface Builder application.
- Foundation Kit — Contains classes that aid the manipulation of non-user interface data, for example `NSArray`, `NSDictionary`, `NSString`.
- MusicKit — Contains classes such as `MKNote`, `MKMidi`, `MKConductor`, and `MKOrchestra` for doing DSP synthesis, MIDI processing, scheduling, etc. The MusicKit does not provide display capabilities. To do this, you must combine the Music and Application Kits.
- SndKit — Contains the `Snd` class and `SndView`, a class for displaying a `Snd`. Lets you record, playback and display sound data.

Each class in each Kit has a "Class Description" on-line. It describes the class in prose form, and describes in detail the instance variables and the methods. These Class Descriptions are also given out in class.

The Apple Kit documentation for the AppKit (http://www.apple.com) and FoundationKit (http://www.apple.com) is available as is the documentation for the

MusicKit (http://www.musickit.org/Frameworks/MusicKit) and SndKit (http://www.musickit.org/Frameworks/SndKit).

1.10. Interface Builder

- Lets you make a graphic user interface using graphic tools. It takes some getting used to, but once you "get it", it's invaluable. It's best understood by working through an example. But a bit of background preparation may help:
- To use Interface Builder, you create an "Interface Builder Custom Object" and configure a user interface to send messages to an instance of your object.
- There is one big restriction on Custom Objects: Any method that is invoked from the user interface must have exactly one argument and this argument must be the control that sent the message. Example:

  ```
  - setFreq: sender
  {
      freq = [sender doubleValue];
  }
  ```

You can also send messages directly to the interface from within your Custom Object. To do this, you need to know the **id** of the controls in your interface. Interface Builder provides a mechanism called "outlets" to do this. Outlets are simply **id**-valued instance variables defined by your Custom Object. In Interface Builder, you can connect an outlet to a control—then, when your program runs, the instance variable's value will be the control and you can send whatever messages you like to it.

1.11. Assignments - Week 1

1. In the MusicKit Concepts Manual (http://www.musickit.org/MusicKitConcepts), read the sections System Overview (http://www.musickit.org/MusicKitConcepts/systemoverview.html) and Representing Musical Data (http://www.musickit.org/MusicKitConcepts/musicdata.html).

2. `Examples/example1` is a program that writes a one-note scorefile. Copy the directory and modify it to write a series of notes. Then say build it using Project Builder or "make" typed at the shell if using a non-MacOS X system.

Chapter 2. Class 2 — MusicKit Representation Classes

2.1. Review: Classes in the MusicKit

- Representation classes (7)

  `MKNote`, `MKPart`, `MKScore`, etc.

- Performance classes (16)

  `MKConductor`, `MKPerformer`, `MKInstrument`, etc.

- DSP Synthesis classes (4)

  `MKOrchestra`, `MKUnitGenerator`, `MKSynthPatch`, `MKSynthData`

2.2. MusicKit Representation Classes

- Musical events are represented by `MKNote` instances.
- `MKNote`s are grouped into `MKPart`s. Each `MKPart` corresponds to a *like manner of realization* during performance. E.g. all notes in a `MKPart` are synthesized with the same synthesis technique or on the same MIDI channel.
- `MKPart`s are grouped into `MKScore`s. A `MKScore` may be written out as an ASCII note list called a “Scorefile”. ScoreFile is actually a simple language. There is also a binary format of the scorefile (in release 2.0). `MKScore`s can also read/write Standard MIDI files. Finally, `MKScore`s, as well as all MusicKit objects, can be “archived” using an `NSArchiver`, as defined in the Application Kit.

- `MKEnvelope` and `MKWaveTable` data are stored in `MKEnvelope` and `MKWaveTable` objects, respectively. There are two subclasses of `MKWaveTables`, `MKPartials` (frequency domain representation) and `MKSamples` (time domain representation).
- A `MKNote` can only belong to one `MKPart` and a `MKPart` can only belong to one `MKScore`. However, `MKEnvelopes` and `MKWaveTables` may be referenced by any number of `MKNotes`.
- `MKTuningSystem` is a class that represents a mapping of the 128 MIDI keys to a set of frequencies. These frequencies need not be increasing.

2.3. The `MKNote` Class

- A `MKNote` consists of:
  - a `noteType` and a `noteTag`
  - a set of parameters
  - an optional `timeTag` and duration

- There are 5 types of `MKNotes`, represented by the *noteType:*
  - `noteOn` - start of a musical phrase or rearticulation
  - `noteOff` - end of a musical phrase
  - `noteDur` - a `noteOn` with a duration
  - `noteUpdate` - update to a running musical phrase(s)
  - `mute` - none of the above

  (in an MusicKit program, a prefix is required, as in `MK_noteOn`)

- The `noteTag` groups a series of `noteOns` and `noteUpdates` with a single `noteOff`. This is called a *phrase*.

  `noteTag` is essential for `noteOn` and `noteOff`

  `noteTag` is optional for `noteDur` and `noteUpdate`

  `noteTag` is not used for `mute`

- A `noteUpdate` without a `noteTag` applies to all running patches and is "sticky".
- A `noteDur` represents a `noteOn`/`noteOff` pair. If another `noteOn` with the same `noteTag` appears before the duration is expended, the implied `noteOff` is canceled.
- The `timeTag` refers to the location of the `MKNote` in a `MKPart` and is only used in that context. Its value is in beats.

2.4. `MKNote` Parameters

- Parameters consist of an integer identifier and a value.

Table 2-1. Examples of `MKNote` Parameters

Examples:	identifier	value
	`MK_freq`	440
	`MK_amp`	0.4
	`MK_waveform`	"SA"

- The MusicKit defines a number of parameters. These begin with the "MK_" prefix. In addition, you can define your own with [`MKNote` parTagForName: @"myParameter"]
- Parameter values may be one of the following types:
  - int
  - double
  - `NSString`
  - `MKEnvelope`
  - `MKWaveTable`
  - any object (e.g. a param's value could be a `MKScore`)

- The object that realizes the `MKNote` determines how to interpret the parameters. Any parameters it doesn't care about are ignored. This makes reorchestration easy.

- The `MKNote` class does automatic type conversion where possible. Thus, the consumer of a `MKNote` parameter need not concern himself with how the parameter was set.

2.5. `MKEnvelope` Class

- `MKEnvelopes` are (*x*,*y*,*z*) triplets:

  x — time in seconds. The first *x* value is usually 0.

  y — value

  z — smoothing value (rarely used)

- An envelope may have a “stickpoint”. The envelopes stops at the stickpoint until the `noteOff` or the end of its duration.
- Example scorefile: [(0,0)(.1,1)(2.1,.5) | (2.7,.1)(3,0)];
- Same example in Objective-C

```
MKEnvelope *env;

double times[] = {0, 0.1, 2.0, 2.7, 3.0};
double values[] = {0, 1, 0.5, 0.1, 0};
env = [MKEnvelope new];              /* or alloc/init in 2.0 */
[env setPointCount:5 xArray:times yArray:values];
[env setStickPoint:2];               /* zero-based          */
```

- Some `MKSynthPatches` (software DSP instruments) also support attack and release parameters. If present, they override the times in the `MKEnvelope`. E.g. if attack is 0.1 in example above, the envelope times becomes {0,.005,0.1...}
- Scaling parameters are also common. E.g. freq1 for value when frequency envelope is 1 and **freq0** for value when frequency envelope is 0.
- The EnvelopeEd program (included in the MusicKit distribution) helps design envelopes.

2.6. `MKWaveTable` Class

- `MKWaveTable` class can supply data as **DSPDatum** or double.
- `MKPartials` objects (frequency domain `MKWaveTables`) are set in a similar manner to `MKEnvelopes`, where (*x*,*y*,*z*) are harmonic number, relative amplitude, and phase in degrees.
- `MKSamples` objects (time domain `MKWaveTables`) are set by supplying a `MKSound` object or **soundfile**. Currently, the `MKSound's` length must be a multiple of 2, and the sound must be 16 bit mono.
- The WaveformEditor program (ccrma ftp) helps design waveforms.

Figure 2-1. EnvelopeEd

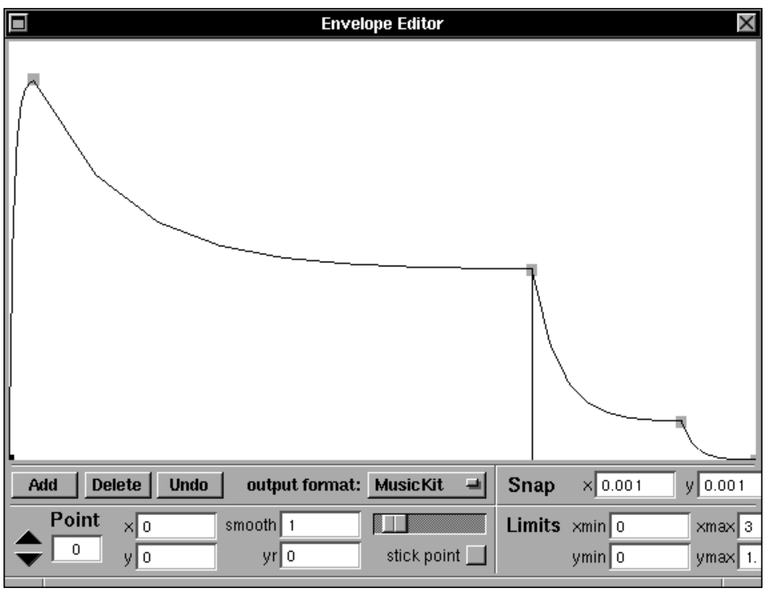
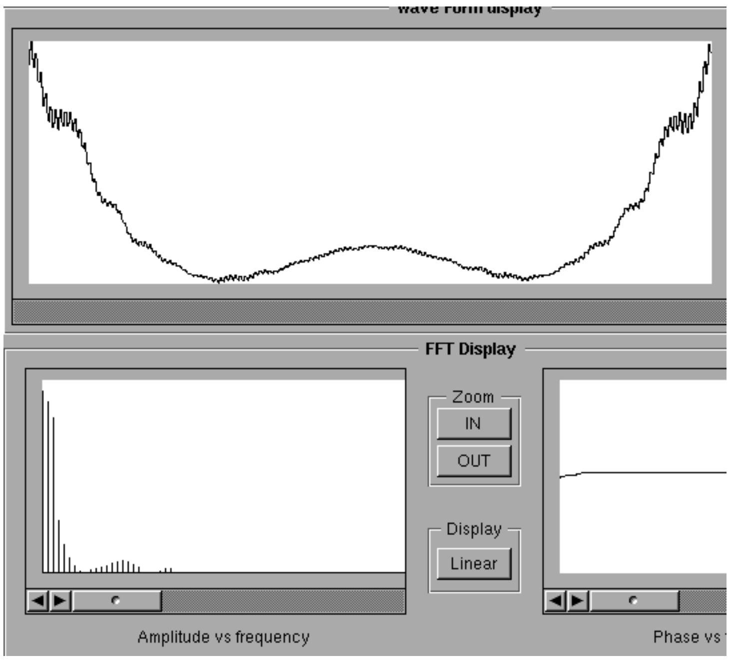


Figure 2-2. Wave Form Display



2.7. Example (review)

```
#import <MusicKit/MusicKit.h>
#import <MKSynthPatches/MKSynthPatches.h>
MKNote *aNote;
MKPart *aPart;
MKScore *aScore;
MKEnvelope *env;

double times[] = {0,0.1,2.1,2.7,3.0};
double values[] = {0,1,0.5,0.1,0};
aScore = [[MKScore alloc] init];
aPart = [[MKPart alloc] init];
aNote = [[MKNote alloc] init];
env = [[MKEnvelope alloc] init];
[env setPointCount: 5 xArray: times yArray: values];
[env setStickPoint: 2];
[aNote setPar: MK_ampEnv toEnvelope: env];
[aNote setPar: MK_freq toDouble: 440.0];
[aNote setTimeTag: 1.0];           /* Play after 1 beat */
[aNote setDur: 1.0];               /* Duration is 1 beat */
```

```
[aScore addPart: aPart];
[aPart addNote: aNote];
aNote = [[MKNote alloc] init];      /* Another Note for info */
[aNote setPar: MK_synthPatch toString: @"Wave1i"];
[aPart setInfo: aNote];
[aScore writeScorefile: @"test.score"];
system("playscore test.score");    /* We'll show how to do this in the progr
```

2.8. Assignment - Week 2

1. Do Interface Builder example (MusicKitClass/example2.wn).
2. Create an Interface Builder program that creates a `MKScore` algorithmically (based on user input), writes a scorefile, and invokes **playscore** by:

   ```
   system("playscore test.score");
   ```

   Use `Examples/example3` as an example. In a few weeks, we'll show you how to play the score directly from Objective-C.

Chapter 3. Class 3 — Performance Classes

3.1. Review: Classes in the MusicKit

Representation classes (7)

`MKNote`, `MKPart`, `MKScore`, etc.

Performance classes (16)

`MKConductor`, `MKPerformer`, `MKInstrument`, etc.

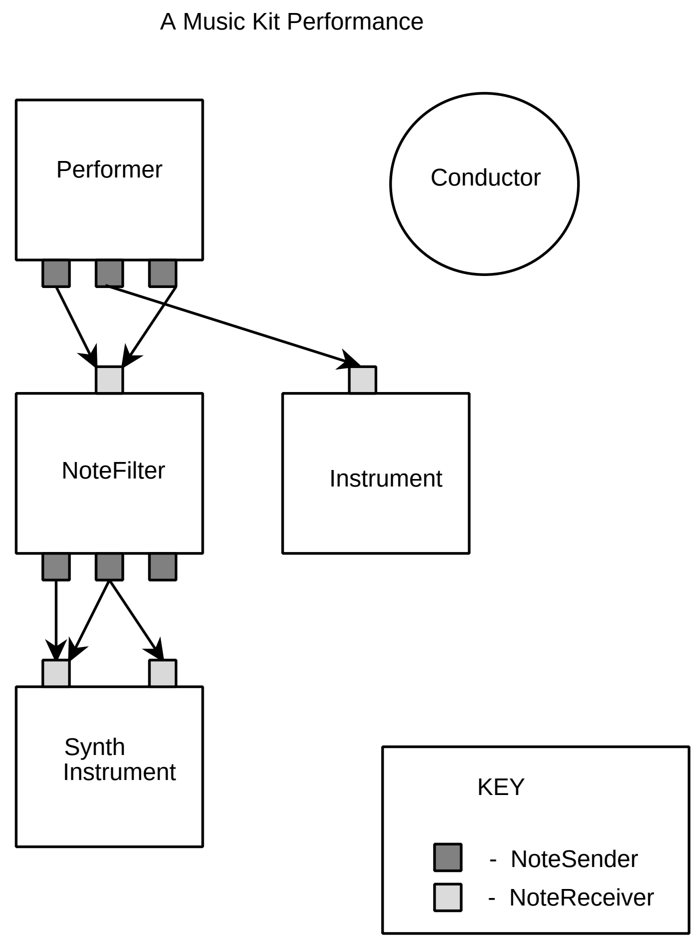
DSP Synthesis classes (4)

`MKOrchestra`, `MKUnitGenerator`, `MKSynthPatch`, `MKSynthData`

3.2. MusicKit Performance Classes

- The `MKConductor` class provides scheduling capability.
- The `MKInstrument` class (abstract) realizes `MKNote`s in some manner. E.g. `MKSynthInstrument`realizes `MKNote`s on the DSP.
- The `MKPerformer` class (abstract) dispatches a time-ordered stream of `MKNote`s. For example to perform a `MKScore`, you use a `MKPartPerformer` for each `MKPart` in the `MKScore`.

Figure 3-1. A MusicKit Performance



3.3. The `MKConductor` Class

- The `MKConductor` class is the primary performance class.
- Allows you to schedule an Objective-C message to be sent in the future. Example:

```
[aConductor sel: @selector(hello:)
             to: anObject
         atTime: 3.0
       argCount: 1, anotherObject];
```

At time 3.0, aConductor will send:

```
[anObject hello: anotherObject];
```

- A MusicKit performance requires a `MKConductor`. You need not create a `MKConductor` explicitly. A “defaultConductor” is created for you and is obtained by:

  ```
  [MKConductor defaultConductor];
  ```

- Multiple `MKConductors` may be used. Each may have its own tempo and may be paused/resumed independently. However, the entire performance is controlled by the `MKConductor` class. E.g., to start a performance, you send:

  ```
  [MKConductor startPerformance];
  ```

3.4. `MKConductor` Class Settings

+ setClocked:

YES (clocked) — messages sent at the proper time Use this mode when you want to interact with the performance. This is the default. Example: ScorePlayer

NO (unclocked) — messages sent in time order, but ASAP. Use this mode when no interaction is required. Example: **playscore**

+ setFinishWhenEmpty:

YES — **[MKConductor finishPerformance]** is automatically triggered when the `MKConductor` has no more scheduled messages. This is the default. Example: ScorePlayer.

NO — The performance continues until the Application sends *[Conductor finishPerformance]*. Example: Ensemble.

MKSetDeltaT(double val**)** sets “scheduler advance” over MIDI and DSP. The larger the argument, the more dependable the performance and the greater the latency. E.g. **MKSetDeltaT**(0.1) sets a “delta time” of 100 ms.

3.5. The `MKInstrument` Class

- An abstract class that realizes `MKNotes` in a manner defined by the subclass. `MKInstruments` are passive, they respond to `MKNotes` sent to them by the user interface or a `MKPerformer`.
- The subclass defines its means of realization by implementing **realizeNote:fromNoteReceiver:**.
-

Table 3-1. `MKInstrument` subclasses provided by the MusicKit

Class	Means of realization
`MKPartRecorder`	adds `MKNotes` to a `MKPart`.
`MKSynthInstrument`	realizes `MKNotes` on DSP.
`MKFileWriter`	(abstract) writes `MKNotes` to a file.
`MKScorefileWriter`	writes `MKNotes` to a scorefile.

-

Table 3-2. Pseudo-Instrument classes provided by MusicKit

Class	Means of realization
`MKMidi`	sends `MKNotes` to MIDI via serial port
`MKScoreRecorder`	manages set of `MKPartRecorders`

- `MKNoteFilter` is a special (abstract) subclass of Instrument that processes `MKNotes` it will be described later.
- `MKInstruments` receive `MKNotes` via their “inputs”, which are small objects called

`MKNoteReceivers`. You can send `MKNotes` directly to a `MKNoteReceiver` or use a `MKPerformer` to dispatch the `MKNotes` (e.g. when playing a `MKScore`).

Table 3-3. `MKNoteReceivers` provided by `MKInstrument` subclasses

Class	Number of `MKNoteReceivers`
`MKSynthInstrument`	1
`MKPartRecorder`	1
`MKScoreRecorder`	1 per Part in the Score
`MKScorefileWriter`	1 per `MKPart` in the scorefile
`MKMidi`	1 per MIDI channel + 1 extra

- You can tell an `MKInstrument` to realize a `MKNote` by sending **receiveNote:** to one of its `MKNoteReceivers`. You can obtain the `MKNoteReceiver` in various ways. To get its first `MKNoteReceiver`, send **noteReceiver** to the `MKInstrument`.
- When sending `MKNotes` *directly* to an `MKInstrument`'s `MKNoteReceiver`, you must update time. Afterwards (if using the DSP) you must make sure that the DSP command buffers are properly emptied. Example:

```
[MKConductor lockPerformance];
[[anInstrument noteReceiver] receiveNote: aNote];
[MKConductor unlockPerformance];
```

3.6. The `MKPerformer` Class

- An abstract class that dispatches `MKNotes` in a time-ordered fashion. `MKPerformers` are active; they are `MKNote` dispatchers.
- Subclass implements **perform**, invoked periodically by its `MKConductor`, as determined by the instance var *nextPerform*, reset within **perform** to specify when next `MKNote` is to occur.
-

Table 3-4. `MKPerformer` subclasses provided by the MusicKit

Class	Means of performance
`MKPartPerformer`	performs `MKNotes` from a `MKPart`.
`MKFilePerformer`	(abstract) performs `MKNotes` from a file.
`MKScorefilePerformer`	performs `MKNotes` from a scorefile.
`MKScorePerformer`	manages a set of `MKPartPerformers`
`MKMidi`	(abstract) performs `MKNotes` it receives via MIDI

- `MKPerformers` send `MKNotes` via their "outputs", small objects called `MKNoteSenders`. A `MKPerformer` sends a `MKNote` to one of its `MKNoteSenders`: [aNoteSender **sendNote:** aNote];

Table 3-5. `MKNoteSenders` provided by `MKPerformer` subclasses.

Class	Number of `MKNoteSenders`
`MKPartPerformer`	1
`MKScorePerformer`	1 per `MKPart` in the `MKScore`
`MKScorefilePerformer`	1 per `MKPart` in the scorefile
`MKMidi`	1 per MIDI channel + 1 extra

- `MKPerformers` may be paused, resumed, delayed, and created dynamically. Similar to Pla "voices" or Common Music "parts".

3.7. Connecting `MKPerformers` to `MKInstruments`

- To connect a `MKPerformer` to an `MKInstrument`, you send the message **connect:** to a `MKNoteSender` of a `MKPerformer` with a `MKNoteReceiver` of an `MKInstrument` as an argument. Example:

```
[[aPerf noteSender] connect: [anIns noteReceiver]];
```

Or, equivalently:

```
[[anIns noteReceiver] connect: [aPerf noteSender]];
```

- Any number of `MKNoteReceivers` may be connected to a `MKNoteSender` and vica versa. Any number of `MKPerformers` and `MKInstruments` may be involved in a single performance. Any number of `MKPerformers` may be governed by one `MKConductor`

3.8. The `MKNoteFilter` Class

- A `MKNoteFilter` (subclass of `MKInstrument`) is an abstract class that processes `MKNotes` in some manner.
- `MKNoteFilter` inherits the `MKNote`-receiving behavior of `MKInstrument`. It also supports the `MKNote`-sending behavior of `MKPerformer`.
- Like any other `MKInstrument`, `MKNoteFilters` implement **realizeNote:fromNoteReceiver:** to process `MKNotes` it receives. Example: MidiEcho.
- Rules:

  1. Copy `MKNote` on write.

  (Or return `MKNote` to original condition)

  2. Copy `MKNote` on store.

- Ensemble is an Application based on `MKNoteFilters`.

3.9. The `MKMidi` Class

- `MKMidi` is a pseudo-`MKPerformer` in that it can't predict when the next `MKNote` will occur. However, it may be treated as any other `MKPerformer`.

- There may be two instances, one for each serial port. Thus, 32 MIDI channels are possible.
- `MKMidi` is a direct connection to the MIDI Device Driver. Similarly, `MKOrchestra` is a direct connection to the DSP. Both implement the following protocol:

Table 3-6. Pseudo-`MKPerformer` Performance Protocol

open	claims device
run	starts device clock
stop	stops device clock
close	releases device after waiting
abort	releases device without waiting

- To use `MKMidi` (or `MKOrchestra`), you must send **run** when you send **startPerformance** to the `MKConductor`. Example:

```
[aMidi run];
[MKOrchestra run];
[MKConductor startPerformance];
```

3.10. Summary of Performance Classes

`MKConductor`

- `MKPerformer`, `MKNoteFilter` & `MKInstrument`
- `MKPartPerformer` & `MKPartRecorder`
- `MKScorePerformer` & `MKScoreRecorder`
- `MKFilePerformer` & `MKFileWriter`
- `MKScorefilePerformer` & `MKScorefileWriter`
- `MKNoteSender` & `MKNoteReceiver`
- `MKSynthInstrument`
- `MKMidi`

3.11. Assignment - Week 3

Copy and modify `Examples/MusicKit/MidiEcho` to do some other type of `MKNoteFilter` processing on MIDI data.

The following is an example `MKNoteFilter`:

```
/* This class is a MKNoteFilter that generates echoes and sends them to
   its successive MKNoteSenders. In MyApp, we connect the MKNoteSenders to
   the MKNoteReceivers of MKMidi, thus producing MIDI echoes on successive
   MIDI channels. To use this app, you need to have a MIDI synthesizer that
   can receive on multiple channels, such as the Yamaha SY77 or FB01. */

#import <MusicKit/MusicKit.h>
#import "EchoFilter.h"
#define NUMCHANS 8  /* My MIDI Synthesizer handles 8 channels. */

@implementation EchoFilter : MKNoteFilter
  /* A simple note filter that does MIDI echo */
{
    double delay;      /* delay between echos, in seconds */
}

- init
  /* Called automatically when an instance is created. */
{    int i;

     [super init];
     delay = .1;
     for (i = 0; i <= NUMCHANS; i++)  /* 1 for each channel plus 'sys' messa
         [self addNoteSender: [[MKNoteSender alloc] init]];
     [self addNoteReceiver: [[MKNoteReceiver alloc] init]];
     return self;
 }

- setDelay: (double)delayArg
  /* change the amount of delay (in seconds) between echoes */
{
    delay = delayArg;
    return self;
}

- connectAcross: anInstOrNoteFilter
  /* Just connects successive MKNoteSenders of the receivers to successive
     MKNoteReceivers of anInstOrNoteFilter. */
```

```
{
    NSArray *pList = [self noteSenders];
    NSArray *iList = [anInstOrNoteFilter noteReceivers];
    int i,siz;
    int pSiz = [pList count];
    int iSiz = [iList count];
    siz = (pSiz > iSiz) ? iSiz : pSiz;   /* Take min length */
    for (i = 0; i < siz; i++)            /* Connect them up */
        [[pList objectAtIndex: i] connect: [iList objectAtIndex: i]];
    return self;
}

#define NOTESENDER(_i) [noteSenders objectAtIndex: _i]

- realizeNote: aNote fromNoteReceiver: aNoteReceiver
  /* Here's where the work is done. */
{
    /* This relies on the knowledge that the MKMidi object sorts its incomin
       notes by channel as well as by noteTag. Thus, duplicating a note with
       a particular noteTag on several channels works ok. In general, this
       MKNoteFilter assumes each output (MKNoteSender) is assigned a unique
       connectio (MKNoteReceiver). */

    int i;
    double curDly;
    int velocity, noteType;
    id newNote;

    noteType = [aNote noteType];
    if (noteType == MK_mute) {
        [NOTESENDER(0) sendNote: aNote];          /* Just forward these */
        return self;
    }
    curDly = 0;
    [NOTESENDER(1) sendNote: aNote];             /* Send current note */
    velocity = [aNote parAsInt: MK_velocity];    /* Grab velocity */
    for (i = 2; i <= NUMCHANS; i++) {         /* Make echoes */
        curDly += delay;
        newNote = [aNote copy];                  /* Need to copy notes here
        if (noteType == MK_noteOn)               /* Decrement echo velocity
            [newNote setPar: MK_velocity toInt: velocity -= 15];
        /* Schedule it for later */
        [NOTESENDER(i) sendAndFreeNote: newNote withDelay: curDly];
    }
```

```
    return self;
}

@end
```

Chapter 4. Class 4 — DSP Synthesis Classes

4.1. Review: Classes in the MusicKit

Representation classes (7)

`MKNote`, `MKPart`, `MKScore`, etc.

Performance classes (16)

`MKConductor`, `MKPerformer`, `MKInstrument`, etc.

DSP Synthesis classes (4)

`MKOrchestra`, `MKUnitGenerator`, `MKSynthPatch`, `MKSynthData`

(also `MKSynthInstrument`)

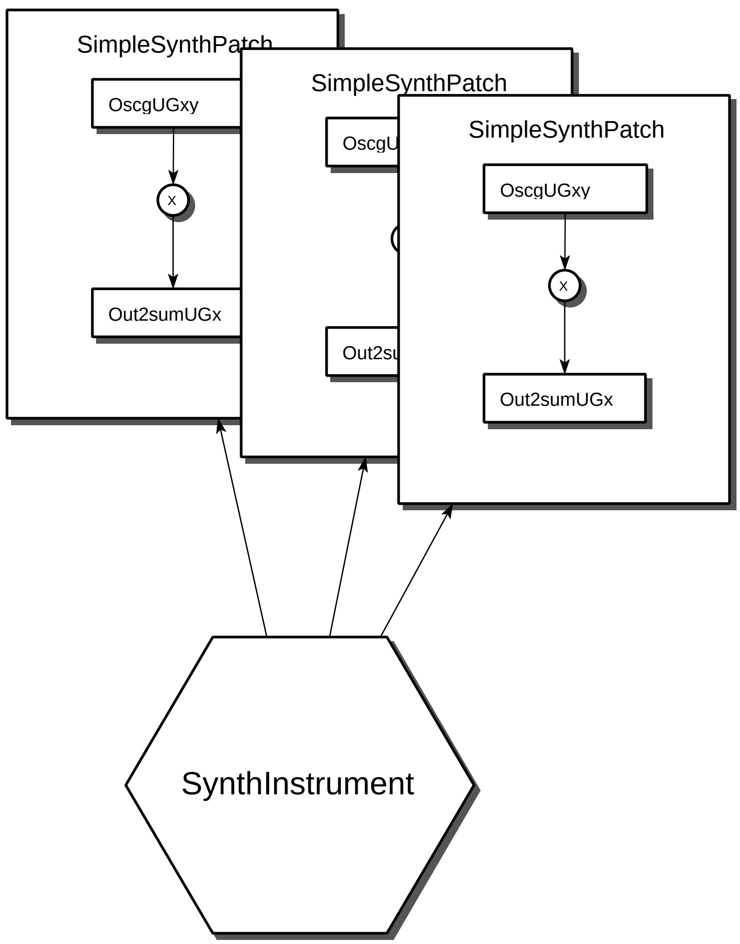
4.2. MusicKit Synthesis Classes

- The `MKOrchestra` class manages the DSP as a whole.
- The `MKUnitGenerator` class (abstract) represents a DSP processing or generating module, such as an oscillator or a filter.
- The `MKSynthData` class represents a piece of DSP memory. A special type of `MKSynthData` called a "patchpoint" is used to connect `MKUnitGenerators`.
- The `MKSynthPatch` class (abstract) contains a list of `MKUnitGenerators` that make up a single sound-producing entity. To produce a chord, multiple instances of a `MKSynthPatch` subclass are required.
- The `MKSynthInstrument` class manages a set of `MKSynthPatches` (voice allocation).

We'll proceed as follows:

1. Look at the system from a high level, focusing on the `MKSynthInstrument` and `MKOrchestra` classes.
2. Look in detail at the `MKOrchestra`, `MKUnitGenerator` and `MKSynthData` classes.
3. 3. Look at the `MKSynthPatch` class. (next time)

Figure 4-1. `MKSynthInstrument`



4.3. A Simple Common Example

- The easiest way to do DSP synthesis is to use one of the `MKSynthPatches` in the `MKSynthPatch` Library. These are general and implement standard synthesis techniques.

- `MKOrchestra` uses the same protocol as `MKMidi`: (**open, run, stop, close, abort**). First, you create and open the `MKOrchestra`.
- Then you create a `MKSynthInstrument` and set its `MKSynthPatch` class (and, optionally, synthPatchCount). Finally, you start the performance and run the `MKOrchestra`:

```
MKSynthInstrument *synthIns;
MKOrchestra *orch = [MKOrchestra newOnDSP: 0];
synthIns = [[MKSynthInstrument alloc] init];
[orch open];
[synthIns setSynthPatchClass: [Pluck class]];
[orch run];
[MKConductor startPerformance];
```

- You can then send `MKNotes` (as explained last class), from your user interface, `MKMidi`, or a `MKPerformer`. E.g.:

```
MKNote *aNote = [[MKNote alloc] init];
[aNote setDur: 1.0];
[MKConductor lockPerformance];
[[synthIns noteReceiver] receiveNote: aNote];
[MKConductor unlockPerformance];
```

4.4. The `MKOrchestra` Class

- Manages control of DSP:

  new or **newOnDSP:**, **open**, **run**, **stop**, **close**, **abort**

- Manages allocation of DSP resources:

  allocUnitGenerator:,

  allocSynthData:,

allocSynthPatch:, etc.

- Class object manages a collection of DSPs:

  + open, **+ run**, **+ allocSynthPatch:**, etc.

- All allocation of DSP resources is done through the `MKOrchestra`. You don't send **alloc** directly to a `MKUnitGenerator` or `MKSynthPatch`.
- You only need to specify allocation requests directly to the `MKOrchestra` when working at a low level. If you use a `MKSynthInstrument`, it takes care of the allocation for you (as in the previous example.) Similarly, if you make your own `MKSynthPatch`, the actual allocation of `MKUnitGenerators` from the `MKOrchestra` is done behind the scenes.

4.5. `MKOrchestra` Settings

+setTimed:

YES (timed) — DSP keeps its own clock running for precise timing. Good for playing scores and when envelope timing is crucial.

NO (untimed) — DSP executes messages as soon as they are received.

+setFastResponse: (before **open**)

YES — Use small sound-out buffers to minimize latency.

NO — Use larger sound-out buffers +more efficient from the system's point of view and gives the DSP more of a cushion.

+setOutputSoundfile: (before **open**)

Sets the name of a file to which samples are written. DACs are not used in this mode.

+setOutputCommandsFile: (before **open**)

Sets the name of a file to which DSP commands are written. DACs are used in this mode.

+setSamplingRate: (before **open**)

Sets the sampling rate to 44100 or 22050.

4.6. The `MKSynthInstrument` Class

- An `MKInstrument` subclass that realizes `MKNotes` on the DSP.
- You specify which `MKSynthPatch` subclass to use with **setSynthPatchClass:**.
- Allocates patches based on `noteTags` of incoming `MKNotes`. Allocation can be done from a global or a local pool. If you send **setSynthPatchCount:**, the pool is local (`MK_MANUALALLOC`) and contains the number of patches specified. Otherwise, pool is global (`MK_AUTOALLOC`).
- Supports automatic preemption of the oldest running patch. You can subclass `MKSynthInstrument` and override one method to provide an alternative preemption strategy.
- Advantage of automatic mode is that there's never any wasted of patches.
- Advantage of manual mode is that important musical parts can be given precedence. (E.g. you can get around a screw case such as overlapping bass-line notes causing a disappearing melody.)
- In Scorefiles, the `MKSynthPatch` is specified in the part info's **synthPatch:** parameter. Manual mode is specified in the part info's **synthPatchCount:** parameter. Example:

```
part p1;         /* Scorefile excerpt*/
p1 synthPatch: "Pluck" synthPatchCount: 2;
```

4.7. Intro to `MKUnitGenerators`

- `MKUnitGenerator` is abstract. It is an Objective-C class that represents a DSP module. The MusicKit supplies a library of `MKUnitGenerator` subclases. Each has the letters UG in its name. The library is sufficient for most common uses.
- To be fast, the DSP uses parallel memory spaces X, Y, P. To get the most possible voices in real time, it is necessary to concern ourselves with memory spaces. The MusicKit has the best benchmarks for 56001 usage we have seen.
- Each `MKUnitGenerator` has some number of inputs and outputs. For each configuration, a `MKUnitGenerator` subclass exists. Example:
  - `OnepoleUG` ("master class") has 1 input and 1 output. Therefore, it has 4 subclasses ("leaf classes"):

    `OnepoleUGxx`, `OnepoleUGxy`,

    `OnepoleUGyx`, `OnepoleUGyy`

  - `OnepoleUGxy` writes its output to X memory and reads its input from Y memory. For starters, you can just use all x memory and worry about optimization later.

- When creating your own `MKUnitGenerator`, you only have to write the DSP code and run the command-line program **dspwrap**, which automatically writes all the classes for you. You *never* have to edit the leaf classes. You may *optionally* edit the master class.

4.8. The `MKUnitGenerator` Library

Table 4-1. `MKUnitGenerators`

Filters:	`Allpass1`, `Onepole`, `Onezero`
Oscillators:	`Oscg`, `Oscgaf`, `Oscgafi`
Scale, mix:	`Add2`, `Mul1add2`, `Mul2`, `Interp`, `Mul2`, `Scl1add2`,`Scl2add2` , `Constant`
Noise:	`Unoise`, `Snoise`

Delay:	`Delay`
Timed switch:	`Dswitcht`, `Dswitch`
Output:	`Out1a`, `Out1b`, `Out2sum`

- Header files for all the `MKUnitGenerators` are referenced from:

  ```
  #import <MKUnitGenerators/MKUnitGenerators.h>
  ```

- DSP source code is provided for all the unit generators on **/usr/lib/dsp/ugsrc/*** . You can copy the DSP source code to a unit generator and modify it to create a new unit generator. You can then run it through **dspwrap** to produce the classes. This is considered "advanced", since it requires knowledge of 56001 assembly and will not be covered in this class.

4.9. The `MKUnitGenerator` Class

- You can allocate a `MKUnitGenerator` from an open `MKOrchestra`.

  ```
  [orch allocUnitGenerator: [Out2sumUGx class]];
  ```

- An allocated `MKUnitGenerator` is, by definition, running on the DSP. You can deallocate a `MKUnitGenerator` by sending:

  ```
  [aUnitGenerator release];
  ```

- `MKUnitGenerators` are in one of three possible states:

  Table 4-2. `MKUnitGenerator` States

MK_idle	Disconnected, not usable.
MK_running	Running
MK_finishing	Envelope release

- To set these states you send the following standard messages:

  idle, **run**, **finish**

- These invoke the following methods, which you may implement if you make your own `MKUnitGenerator` class:

  idleSelf, **runSelf**, **finishSelf**

- The return value of **finish** (and **finishSelf**) is a double that indicates the time until the `MKUnitGenerator` will be finished.
- In addition to these standard methods, individual `MKUnitGenerator` classes implement methods particular to their operation. Common methods include **setInput:** and **setOutput:**. E.g. oscillators implement **setFreq:**.

4.10. Connecting `MKUnitGenerators`

To connect two `MKUnitGenerators`, you use a "patchpoint", a kind of `MKSynthData`, which you can allocate from the `MKOrchestra`. You must be sure to specify the memory space corresponding to the memory space of the input/output that the `MKUnitGenerators` will be reading/writing. Example:

```
MKSynthData *pp;
MKUnitGenerator *osc,*out;
MKOrchestra *orch = [MKOrchestra new];
[orch open];
pp = [orch allocPatchpoint: MK_xPatch];
osc = [orch allocUnitGenerator: [OscgUGxy class]];
out = [orch allocUnitGenerator: [Out1aUGx class]];
[osc setOutput: pp];
[out setInput: pp];
[osc setFreq: 440];
[osc setAmp: 1.0];
[osc setTableToSineROM];
[orch run];
[osc run];
[out run];
/* You now hear a full-amplitude sine wave at 440 hz */
```

Patchpoints may be reused, if you're careful about the order in which `MKUnitGenerators` run. (More on this later.)

4.11. The `MKSynthData` Class

- In addition to patchpoints, you may need other DSP memory. For example, you may want to load a wave table. To do this, you allocate a `MKSynthData` object.
- To allocate a `MKSynthData`, you specify the length and the space:

  ```
  MKSynthData *sd  = [orch allocSynthData: MK_xData length: 256];
  ```

- To load the `MKSynthData` with an array:

  ```
  DSPDatum someData[256] = {0, 1, 2, 3, ...};
  [sd setData: someData];
  ```

- To load the `MKSynthData` with a constant:

  ```
  [sd setToConstant: 1];
  ```

- Since patchpoints are actually `MKSynthData`, you can use these methods for them as well.
- For convenience, `MKWaveTables` have a **dataDSPLength:** method:

  ```
  [sd setData: [aWaveTable dataDSPLength: 256]];
  ```

4.12. Simple Example of a Collection of `MKUnitGenerators`, Operated from a User Interface

```
#import <MusicKit/MusicKit.h>
#import <MKUnitGenerators/MKUnitGenerators.h>
#import "MyCustomObject.h"

@implementation MyCustomObject

MKSynthData *pp;
MKUnitGenerator *osc,*out;

+ init
{
 MKOrchestra *orch = [MKOrchestra new];
 [MKUnitGenerator enableErrorChecking: YES];
 [orch open];
 pp = [orch allocPatchpoint: MK_xData];
 osc = [orch allocUnitGenerator: [OscgUGxy class]];
 out = [orch allocUnitGenerator: [Out2sumUGx class]];
 [osc setOutput: pp];
 [out setInput: pp];
 [osc setFreq: 440];
 [osc setAmp: 0.1];
 [osc setTableToSineROM];
 [osc run];
 [out run];
 [orch run];
 [MKConductor startPerformance];
}

+ setFreqFrom: sender
{
 [MKConductor lockPerformance];
 [osc setFreq: [sender doubleValue]];
 [MKConductor unlockPerformance];
}

+ setBearingFrom: sender
{
 [MKConductor lockPerformance];
```

```
 [out setBearing: [sender doubleValue]];
 [MKConductor unlockPerformance];
}

+ setAmplitudeFrom: sender
{
 [MKConductor lockPerformance];
 [osc setAmp: [sender doubleValue]];
 [MKConductor unlockPerformance];
}

@end
```

4.13. Assignment - Week 4

Modify `Examples/example4` to make a different sound. Try using some other `MKUnitGenerators`.

Read the documentation on DSP synthesis in the MusicKit Concepts Manual (http://www.musickit.org/MusicKitConcepts). Next week we'll cover `MKSynthPatches`.

Chapter 5. Class 5 — `MKSynthPatches`

5.1. Review: Classes in the MusicKit

Representation classes (7)

`MKNote`, `MKPart`, `MKScore`, etc.

Performance classes (16)

`MKConductor`, `MKPerformer`, `MKInstrument`, etc.

DSP Synthesis classes (5)

`MKOrchestra`, `MKUnitGenerator`, `MKSynthPatch`, `MKSynthData`, `MKPatchTemplate` (also `MKSynthInstrument`)

5.2. MusicKit Synthesis Classes (review)

- The `MKOrchestra` class manages the DSP as a whole.
- The `MKUnitGenerator` class (abstract) represents a DSP processing or generating module, such as an oscillator or a filter.
- The `MKSynthData` class represents a piece of DSP memory. A special type of `MKSynthData` called a “patchpoint” is used to connect `MKUnitGenerators`.
- The `MKSynthPatch` class (abstract) contains a list of `MKUnitGenerators` that make up a single sound-producing entity. To produce a chord, multiple instances of a `MKSynthPatch` subclass are required.
- The `MKSynthInstrument` class manages a set of `MKSynthPatches` (voice allocation).
- The `MKPatchTemplate` class is an auxiliary class used to define the `MKUnitGenerators` that make up a `MKSynthPatch`.

5.3. The `MKSynthPatch` Class

- Abstract class. You never directly instantiate an instance of the `MKSynthPatch` class. You instantiate its subclasses.
- Each subclass represents a particular synthesis technique. E.g. frequency modulation synthesis, additive synthesis, etc.
- An instance is a single sound-producing entity. Can not ordinarily produce chords.
- A collections of `MKSynthPatch` instances of a particular class are most conveniently managed by a `MKSynthInstrument`. Multiple collections of instances of different classes may be managed by multiple `MKSynthInstruments`.
- Alternatively you can allocate and manage a collection of `MKSynthPatch`es yourself:

  ```
  id  sp = [orch allocSynthPatch: [Pluck class]];
  ```

- The MusicKit provides a library of `MKSynthPatch`es.

5.4. The MusicKit SynthPatch Library Classes

Table 5-1. WaveTable synthesis:

Wave1	“1” stands for one oscillator
Wave1i	“i” stands for interpolating oscillator
Wave1v	“v” stands for vibrato (random,periodic)
Wave1vi	
DBWave1vi	“DB” stands for “data base of timbres”
DBWave2vi	“2” stands for two oscillators

Table 5-2. Frequency Modulation synthesis:

Fm1	“1” stands for one modulator
Fm1i	“i” stands for interpolating carrier

Fm1v	“v” stands for vibrato (random,periodic)
Fm1vi	
DBFm1vi	“DB” stands for “data base of timbres”
Fm2pvi	“2p” stands for 2 modulators in parallel
Fm2cvi	“2c” stands for 2 modulators in cascade
Fm2pnvi	“n” stands for a noise modulator
Fm2cnvi	

Table 5-3. Plucked string synthesis:

Pluck	Karplus/Strong/Jaffe/Smith plucked string simulation

All Wave and Fm `MKSynthPatch`es have separate envelopes with arbitrarily many points on amplitude, frequency, and the various FM indecies. Vibrato may run at audio rates. Both carrier and modulators may have any periodic waveform.

5.5. The `MKSynthPatch` Library Timbre Data Base

You specify a “timbre” as a string to the **waveform** parameter. For the `DBFm1vi`, you can also specify the modulating wave as a timbre.

Each “timbre” represents a family of `MKWaveTables`, one for each frequency range. This is very similar to how samplers work. By changing waveforms as the pitch changes, the “munchkin” effect is avoided. Also, the waveforms are band-limited, preventing aliasing.

List of timbres, derived from analysis of recorded data, includes:

soprano, tenor and bass voices singing various vowels woodwind instrments such as clarinet, oboe and sax. stringed instruments such as violin and cello

piano

various electronic waveforms such as square wave

Interpolation from one timbre to another is supported in some of the `MKSynthPatch`es.

In release 3.0 the data base is user-extendable.

5.6. Making Your Own `MKSynthPatch` Class

A `MKSynthPatch` subclass consists of three fundamental parts:

1. A specification of *a collection of `MKUnitGenerators` Classes* instances of which comprise each `MKSynthPatch` instance. This is done using an auxiliary object called a "`MKPatchTemplate`." A single `MKSynthPatch` class may supply various `MKPatchTemplates` representing various "flavors" of the `MKSynthPatch`. For example, there may be an additive synthesis `MKSynthPatch` with an 8-oscillator flavor and a 16-oscillator flavor. This is done by supplying the class method:

   ```
   + patchTemplateFor:
   ```

2. A description of the *interconnections* of the `MKUnitGenerator` instances. This may be done in the `MKPatchTemplate` or in the `MKSynthPatch` **init** instance method.
3. A description of the *behavior* of the `MKSynthPatch` when sent notes. This is done by supplying the instance methods:

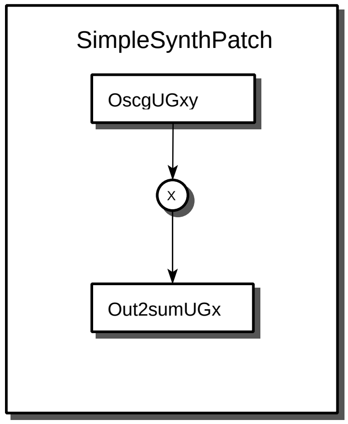
   noteOnSelf:

   noteOffSelf:

   noteUpdateSelf:

   noteEnd

Figure 5-1. A Simple Synth Patch



+patchTemplateFor:aNote { . . . }

-init { . . . }

-noteOnSelf:aNote { . . . }

-noteOffSelf:aNote { . . . }

-noteEnd { . . . }

5.7. Specifying a Collection of `MKUnitGenerators`

To specify the collection, the `MKSynthPatch` subclass implements a single class method:

```
+ patchTemplateFor: aNote
```

This method creates the `MKPatchTemplate` used to represent the connections. The `MKNote` passed to the method may be used to choose between various “flavors”.

The `MKPatchTemplate` consists of a list of the `MKUnitGenerator` *classes* and `MKSynthData` *requests* needed to build an instance of the `MKSynthPatch`.

The `MKOrchestra` uses the `MKPatchTemplate` to build an instance of the `MKSynthPatch`. For each entry in the `MKPatchTemplate`, it allocates an appropriate

`MKUnitGenerator` or `MKSynthData` *instance*. The collection of `MKUnitGenerators` appears in the `MKSynthPatch` instance as a `List` object in the instance variable *synthElements*. The instance can retrieve a particular `MKUnitGenerator` or `MKSynthData` instance by sending itself the message:

```
+ synthElementAt: (int) index
```

The `MKUnitGenerators` appear in the order they were specified in the `MKPatchTemplate`. For convenience, the `MKPatchTemplate` specification methods return the integer used to later access the particular element. By convention a `MKSynthPatch` stores this integer in a static int variable.

`MKUnitGenerators` may be specified as ordered or unordered. By default, they are ordered. Note that you must specify the particular `MKUnitGenerator` leaf class.

`MKSynthData` are specified by supplying a memory space and a length.

Example for simple `MKSynthPatch`:

```
static int  sc, patchPoint, out;

+ patchTemplateFor: aNote
/* We ignore aNote in this simple MKSynthPatch */
{ static PatchTemplate  *t = nil;

      if (!t)  {            /* Only create template the first time. */
                t = [[PatchTemplate alloc]  init];
                osc = [t addUnitGenerator: [OscgUGxy class]];
                patchPoint = [t addPatchpoint: MK_xPatch ];
   out = [t addUnitGenerator: [Out2sumUGx class]];
      }
      return t;
}
```

Alternative to `MKPatchTemplate:`. Just allocate directly from `MKOrchestra` in **-init** method. The advantage of using a `MKPatchTemplate` is that the patch is stored as data and aborting on allocation failure is handled automatically.

5.8. Specifying the Connections

Two `MKUnitGenerator` instances communicate via a patchPoint. The patchPoint's memory space must match the space of the input or output to which it is connected. Connections are made by sending an appropriate message to the `MKUnitGenerators` with the patchPoint as an argument.

There are two ways to specify the connections:

1. 1. In the **init** method.
2. 2. In the `MKPatchTemplate` itself.

It is a bit easier to specify the connections in the **init** method. The only advantage of using the `MKPatchTemplate` is that it allows `MKSynthPatch`es to be more easily edited using a patch editor, since the connections can be stored as data using the NXTypedStream mechanism. For now, we will address only the **init** approach.

Let's continue our example. To make it easy to read, let's define some macros:

```
#define  OSC [self synthElementAt: osc]
#define  OUT [self synthElementAt: out]
#define  PATCHPOINT [self synthElementAt: patchpoint]

+ init
{  [OSC setOutput:PATCHPOINT];
   [OUT setInput:PATCHPOINT];
  return self;
}
```

5.9. Specifying the Performance Behavior

Behavior is defined by supplying the instance methods:

noteOnSelf:

noteUpdateSelf:

noteOffSelf:

noteEndSelf

These are invoked as follows:

1. When a `noteOn` or `noteDur` arrives, the **noteOn:** message is sent. This invokes **noteOnSelf:**
2. When a `noteUpdate` arrives, the **noteUpdate:** message is sent. This invokes **noteUpdateSelf:**
3. When a `noteOff` arrives or the end of the duration occurs, the **noteOff:** message is sent. This invokes **noteOffSelf:** **noteOffSelf:** returns the time required to finish, in seconds. This is ordinarily the time for the amplitude envelope to finish its release portion.
4. When the phrase is really finished (the release portion is finished) , the **noteEnd** message is sent. This invokes **noteEndSelf**

Like a `MKUnitGenerator`, a `MKSynthPatch` may be in one of three states:

Table 5-4. `MKSynthPatch` States

`MK_idle`	Not producing sound.
`MK_running`	Running.
`MK_finishing`	`MKEnvelope` release.

- A `MKSynthPatch` is in the idle state when it is first created or after it has received `noteEnd`
- A `MKSynthPatch` is in the running state when it has received a `noteOn` or `noteDur`.
- A `MKSynthPatch` is in the finishing state when it has received a `noteOff` or its duration has elapsed.
- *The only requirement for the behavior of a `MKSynthPatch` is that it be left "safe" and "quiet" when idle.*
- An easy way to make a `MKSynthPatch` quiet is to set its amplitude to 0.

5.10. Specifying the Performance Behavior + Example

For our simple example we make several assumptions:

1. We ignore noteUpdates for now.

2. We assume that every parameter about which we care is present in every note.

Since our simple example has no envelopes, we need not implement the **noteOffSelf:** method. We can just use the default version that returns 0.

So we need to provide only two methods, **noteOnSelf:** and **noteEndSelf**.

```
+ noteOnSelf: aNote
{
  [OSC setFreq: [aNote freq]];
  [OSC setAmp: [aNote parAsDouble: MK_amp]];
  [OUT setBearing: [aNote parAsDouble: MK_bearing]];
  [synthElements makeObjectsPerform: @selector(run)];
  return self;
}

+ noteEndSelf
{
  [OSC setAmp: 0.0];
  return self;
}
```

5.11. Complete Example

```
#import <MusicKit/MusicKit.h>
#import <MKUnitGenerators/MKUnitGenerators.h>

@implementation MySynthPatch : MKSynthPatch { }

static int osc, patchPoint, out;  /* Used as indexes into synthElements arra
+ patchTemplateFor: aNote
{
 static PatchTemplate *t = nil;

 if (!t)  {             /* Only create template the first time. */
         t = [[PatchTemplate alloc] init];
     osc = [t addUnitGenerator: [OscgUGxy class]];
    patchPoint = [t addPatchpoint: MK_xPatch];
    out = [t addUnitGenerator: [Out2sumUGx class]];
 }
 return t;
}
```

```
#define OSC [self synthElementAt: osc]
#define OUT [self synthElementAt: out]
#define PATCHPOINT [self synthElementAt: patchpoint]

+ init /* Sent once when object is created */
{
 [OSC setOutput: PATCHPOINT];
   [OUT setInput: PATCHPOINT];
 return self;
}

+noteOnSelf: aNote
{
 [OSC setFreq: [aNote freq]];
   [OSC setAmp: [aNote parAsDouble: MK_amp]];
  [OUT setBearing: [aNote parAsDouble: MK_bearing]];
 [synthElements makeObjectsPerform: @selector(run)];
 return self;
}

+ noteEndSelf
{
 [OSC setAmp: 0.0];
 return self;
}
```

5.12. Fancier SynthPatches

To support `noteUpdates`, you merely supply a **noteUpdateSelf:** method. It is up to you what parameters you want to allow to change in a `noteUpdate`. Example:

```
+ noteUpdateSelf: aNote
{
 [OSC setFreq: [aNote freq]];
  [OSC setAmp: [aNote parAsDouble: MK_amp]];
   [OUT setBearing: [aNote parAsDouble: MK_bearing]];
   return self;
}
```

To relax the restriction that all MKNotes need to have every parameter present, you can set the parameter in **noteOnSelf:** and **noteUpdateSelf:** only when it is present, store its value in an instance variable, and set it back to a default value in noteEnd. E.g., if there is an instance variable *freq*.

```
+ noteOnSelf: aNote
{
 if ([aNote isParPresent: MK_freq])
     freq = [aNote freq];
  [OSC setFreq: freq];
  . . .
}

+ noteEndSelf
{
  freq = 440;
  [OSC setAmp: 0.0];
  return self;
}
```

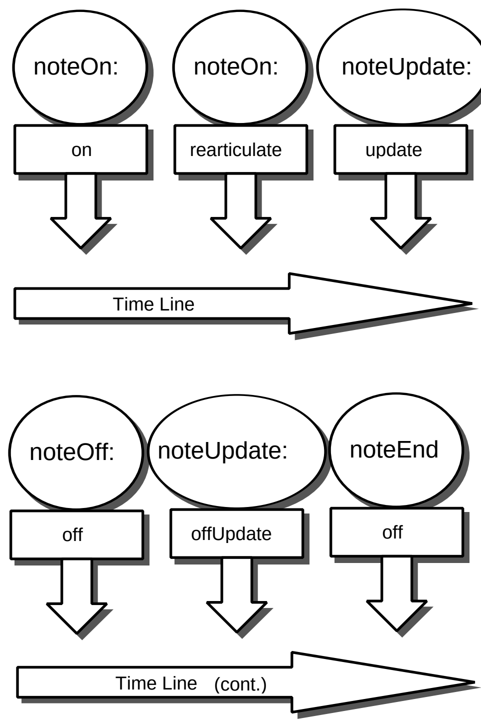
To add an amplitude envelope, we need to use an AsympUG MKUnitGenerator to create the envelope, write it to a patch point and use an oscillator that is capable of reading its amplitude from a patchpoint. The OscgafiUG supports reading both its amplitude and its frequency from a patchpoint. The C function MKUpdateAsymp() makes it easy to apply an envelope with AsympUG and supports attack and release times, scaling values, and phrase transitions:

```
MKUpdateAsymp(
 AsympUG *anAsymp, // asymp instance
 MKEnvelope *ampEnv, // the envelope
 double amp0, // value when env at 0
 double amp1, // value when env at 1
 double ampAtt, // attack time
 double ampRel, // release time
 double portamento,  // transition time on rearticulation
 MKPhraseStatus phraseStatus);   // see below
```

Any argument may be omitted. For double arguments, omitting the argument means supplying the special value MK_NODVAL (which stands for “No Double Value”).

Phrase status is obtained by sending [**self phraseStatus**];

Figure 5-2. noteTypes Time Line



5.13. The Complete Story About Phrase Status

Phrase status defines, within a `MKSynthPatch`, all the possible places we can be in a MusicKit phrase:

Table 5-5. Phrase States

MK_phraseOn	New phrase
MK_phrasePreempt	New phrase, but from preempted patch
MK_phraseUpdate	Note update
MK_phraseOff	Note off
MK_phraseOffUpdate	Note update during release
MK_phraseEnd	Note end
MK_noPhraseStatus	Not in a `MKSynthPatch` method.

Preemption occurs when there is not enough DSP resources to support the requested number of simultaneous notes. It is controlled by the `MKSynthInstrument`. The

MKSynthPatch designer need only implement a method:

```
+ preemptFor: newNote
```

A typical implementation aborts any running envelopes. Example:

```
+ preemptFor: newNote
{
 [ampEnvelopeAsymp preemptEnvelope];
 return self;
}
```

5.14. Assignment - Week 5

Study `Examples/exampleSynthPatch`.

Modify `Envy.m` to set the `MKWaveTable` of the synthesis. Then recompile it, create a scorefile that specifies a `MKWaveTable`, and test it.

Modify any one of the example synthpatches on that directory to do some other sort of synthesis, such as amplitude modulation.