EarSketch

JavaScript Curriculum

Version: 8/8/2015

Table of Contents

PART 1

Welcome

Welcome to EarSketch! In these lessons, you will be learning computer science and music technology side by side. You will use either Python or JavaScript, two of the most popular computer programming languages in the world, to create and remix music within the same kind of digital audio workstation (DAW) software used throughout the music industry. Along the way, we will cover important computer science concepts like variables, functions, lists, loops, and conditionals, and we will connect them to important music and music technology concepts like rhythm, form, effects, and multi-track editing.

Writing computer programs to create music has been an important part of the music industry since the earliest days of computers over 50 years ago, and is at its most popular today. Musicians and programmers write computer code for many exciting uses: from creating new sounds or effects or musical structures, to designing entirely new ways to create and perform music.

Throughout this curriculum you will learn to write code that can help you more easily make music, and make music that's more unique to you. Once you learn to write computer code, you can take those skills with you to any career you can imagine, whether in the music industry or elsewhere.

Happy programming & composing!

1 Getting Started with EarSketch

Why Learn Programming for Music?

There are many ways to get involved in making music, including playing an instrument, writing music, designing sound for film, producing beats, and so on. Computers have greatly expanded these possibilities. The musician's toolbox has grown, and new skills are needed to use these tools.

In EarSketch, you will write code that the computer understands as a set of instructions, or an **algorithm**, to make music with. Not only does this make traditional styles of music-making more efficient, it also opens many new possibilities for music that could never have existed before computers.

The practice of creating music by programming is called **algorithmic composition**. Here are some reasons you might want to program to create music:

- **You can automate tedious tasks**. Imagine that you want to combine hundreds or even thousands of snippets of sound taken from dozens of audio files. You can do this through a **graphical user interface** (or **GUI**) by manually clicking and dragging audio files, but it will take many hours. If you can efficiently describe what you want to do through a computer program, you can create the same music in much less time.
- **You can experiment more easily.** After you create that song made out of thousands of pieces of audio, what happens if it's not perfect? You will probably want to tweak it to make it better. Some changes, like adjusting volume, are easy to make with a GUI. Other changes, like swapping out every occurrence of an audio file for a new one, would take a really long time to do with a graphical interface. Through programming, you can quickly explore these "what if" questions by changing just one or two lines of code.
- **You can roll the dice.** Algorithmic composition has a long history of using the computer to make random decisions, like flipping a coin or dice. One reason for doing this is to make the music sound more human. For example, you might randomly change the starting time of an audio clip by a tiny amount each time it is repeated, in order to simulate the imperfect timing of humans. Or you might randomly choose among 5 different var-

iations of a drum beat in each measure of music, so that the drum track always sounds just a little bit different and unpredictable.

- **You can turn anything into music.** Programming enables you to easily map data from any domain into music. The daily value of the stock market can control the changing volume of a track; the amount of rainfall in each month of the year can generate a new rhythm; or the words in a poem can control the order in which different musical clips are arranged on a timeline. This process is called **data sonification**.

Here's an example of some music made in EarSketch. This example is in an electronic dance music style, but you can make any kind of music you like here.
audioMedia/Buzzjam.mp3

Tools of the Trade: DAWs and APIs

Music-making is often broken up into different jobs, to spread the work out. These job titles include: composer, lyricist, performer, recording engineer, producer, mixing/mastering engineer, and many others. They often overlap, and personal computers make it easier than ever for one person to take on many roles. In EarSketch, you'll get a chance to take on several of these roles.

The main tool for producing music on a computer is the **Digital Audio Workstation**, or **DAW**. A DAW is a piece of computer software for recording, editing, and playing digital audio files. Audio files store information that the computer uses to play back music. In the context of a DAW, these audio files are called **clips**. The DAW allows you to edit and combine multiple clips simultaneously on a musical timeline, and to see and hear how they line up over time. It also makes it easy to synchronize clips with each other. DAWs are used in both professional recording studios and in home laptop-based studios. Two popular DAWs are **Pro Tools** and **Logic Pro**.

So what exactly is EarSketch? EarSketch is a DAW with extra features: *the ability to place audio clips into a DAW timeline using computer code*. This opens up musical possibilities that are difficult or impossible to create with a regular DAW, and makes many tasks much faster. Just like EarSketch adds extra features to a DAW, it also adds features to a programming language. Programming languages come with a set of built-in tools, most of which are general-purpose. EarSketch adds extra tools to this set to help us accomplish our specific goal of making music. This collection of tools is called an **Application Programming Interface**, or **API**. Other examples of APIs include the Google Maps API (a set of tools for embedding maps into websites or apps) and the Web Audio API (tools for working with audio in websites).

The EarSketch Workspace

Let's begin by making an account. Click *register new account* at the top right of the workspace to get started.

The webpage you are visiting now is the **EarSketch workspace.** It consists of several panels with different purposes. You can show or hide these panels by clicking the green icons near the top left of the webpage, in the navigation bar:

FIGURE 1-1

The navigation bar

From top to bottom, these 7 panels do the following:

Sound Browser - Sounds Tab: Here, you can browse and search a collection of short pre-made audio clips for you to use in your music. The clips were made by musicians/producers **Young Guru** and **Richard Devine**. We will learn to add these clips to the DAW by writing code. To upload or record your own sounds, press the + symbol. You can find your uploaded sounds by clicking *artists*, then selecting your username.

CHOOSING THE RIGHT SOUNDS

EarSketch's audio clips are grouped into folders, with names indicating the style of music they are usually used in. In the folders, you can find clips of different instruments: drums, bass, guitar, keys, synth, and so on. The clips in each folder are de-

signed to sound good together; if you're unsure about how to choose the right clips for your track, stick with one folder.

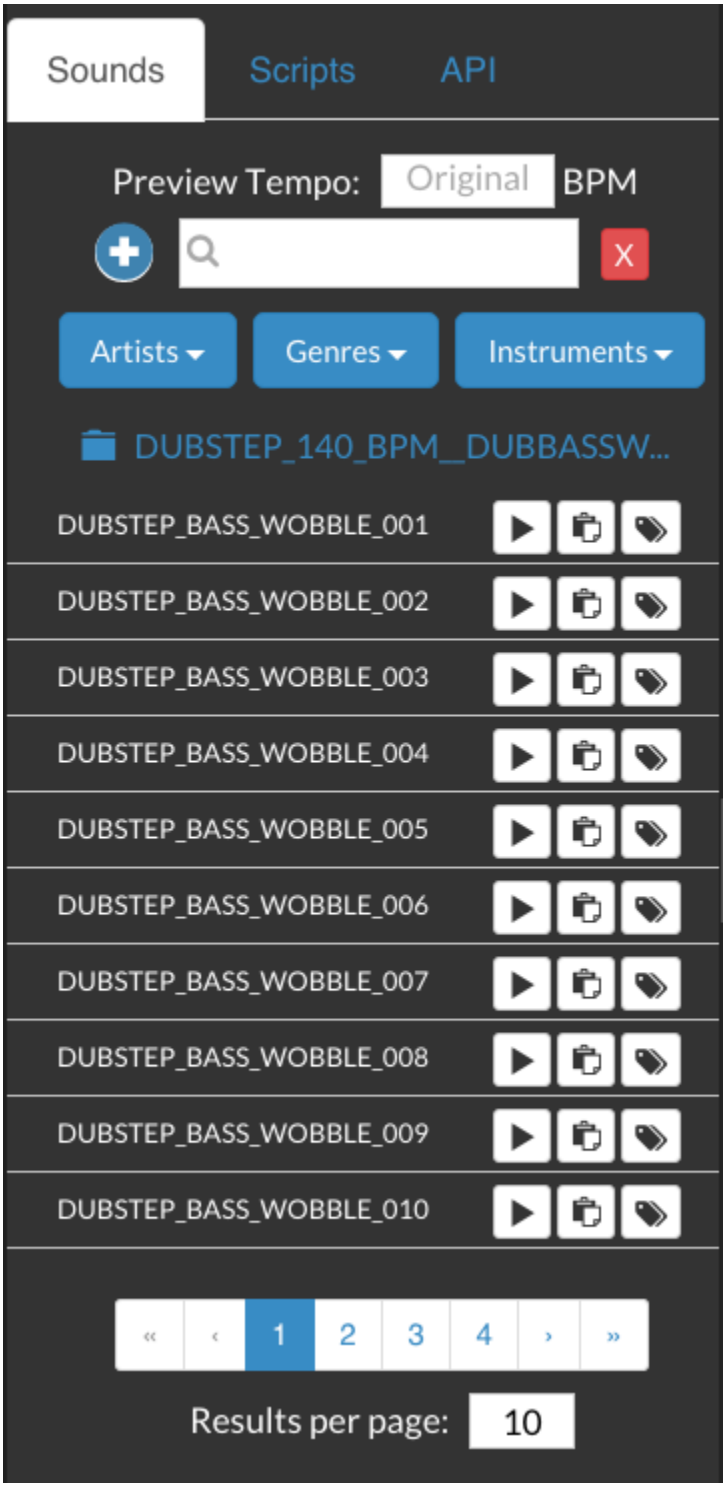


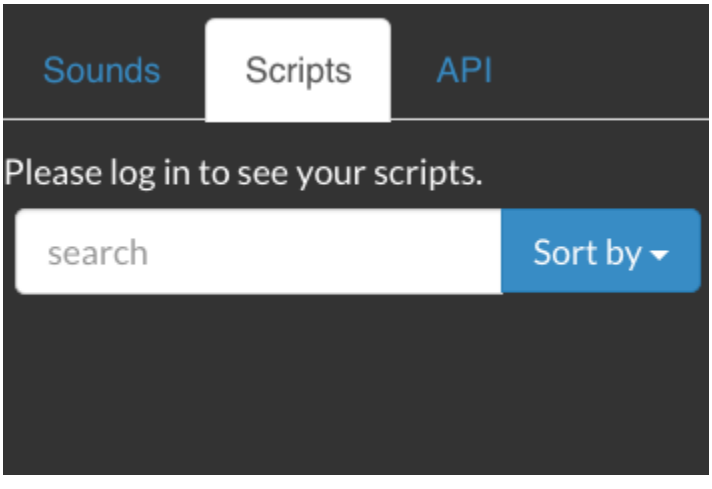
FIGURE 1-2

Sound Browser – Sounds

Sound Browser - Scripts Tab: A list of your saved EarSketch projects. Each script is a separate music project. Click a project title to open it in a new tab (in the code editor panel).

FIGURE 1-3

Sound Browser – Scripts



Sound Browser - API Tab: A description of every EarSketch function. We will learn more about this later.

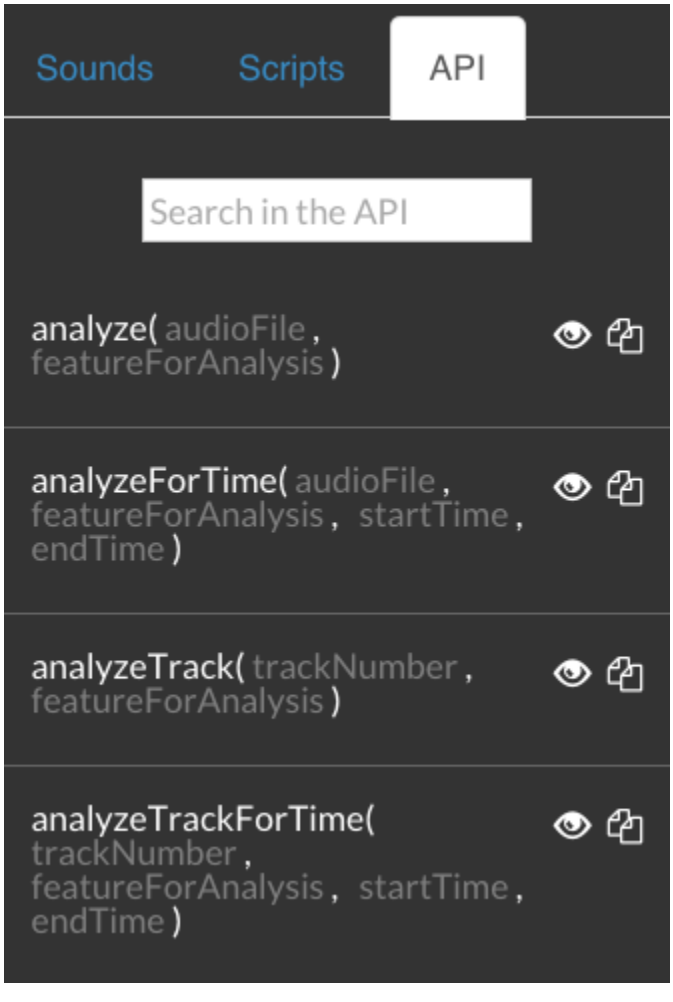


FIGURE 1-4

Sound Browser – API

Digital Audio Workstation (DAW): A timeline view of your current song, showing which audio clips you have added to the song and when they come in. It lets you hear your song, and also visualize its structure.

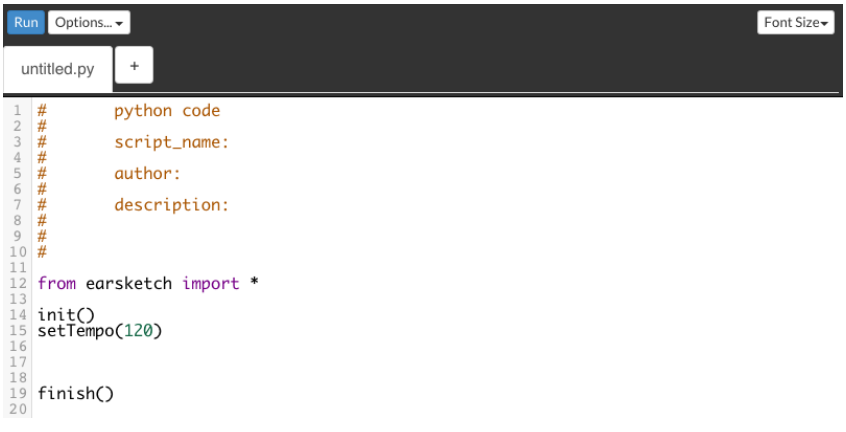
FIGURE 1-5

Digital Audio Workstation (DAW)

Code Editor: A text editor with numbered lines. Type your code here, press "Run", and it will turn into music in the DAW.

FIGURE 1-6

Code Editor



Console: The console displays important information about the code you have just run in the code editor, including the location of errors in your code. It is a common and important feature in programming interfaces.

FIGURE 1-7

Console

Curriculum: The current panel. Here, you can learn how to make music with code.

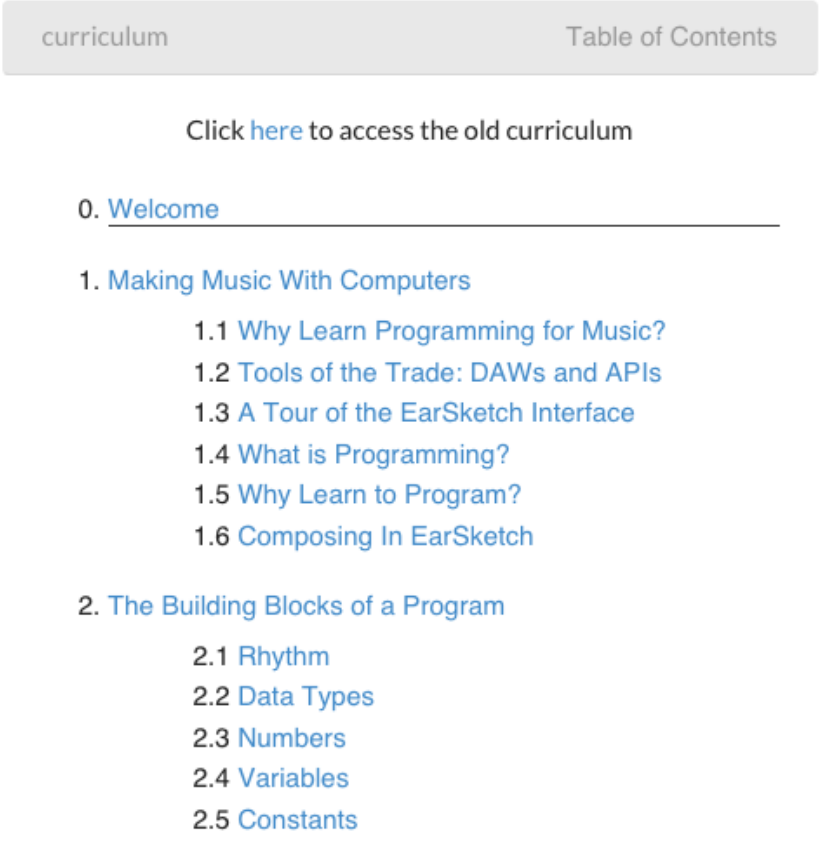


FIGURE 1-8

Curriculum

Running a Script

The basic workflow when making a song in EarSketch is something like this: type your musical code into the code editor, press the *run* button to execute the code and add it to the DAW, and press *play* in the DAW to hear it.

Take a look at the bar just above the Code Editor panel. There are some useful buttons and menus here: “Run” and “Options”.

LANGUAGE MODE

Click "Options" and select the programming language your class is working in:

"Python Mode" or "Javascript Mode". There are many programming languages out there, but these are two of the most popular. Keep in mind that the curriculum will look different depending on which language mode you have selected.

Let's try running a code example in EarSketch! First, make sure the code editor is visible (click its icon in the navigation bar). Now, take a look at the box of text below: this is our code example. Press the clipboard icon at its top right to copy it into the code editor. Don't worry about understanding the code at this point, we will learn its meaning later.

```
// javascript code
//
// script_name: Intro_Script
//
// author: The EarSketch Team
//
// description: This code adds one audio clip to the DAW
//
//
//

//Setup Section
init();
setTempo(120);

//Music Section
fitMedia(TECHNO_SYNTHPLUCK_001, 1, 1, 9);

//Finish Section
finish();
```

Python and Javascript are both called **scripting languages**, so your code is called a **script**. In "Options", press *Save Script to Cloud*, type a name like "MyFirstScript", and press *save*. Note that your script name cannot start with a number, and that all non-letter characters (including spaces) in the script name will become underscores. Be sure to save your work frequently! Take a look at the other items in the "Options" menu: you can start a new script, save a script or audio to your computer, and so on.

To turn your code into music, press the "Run" button (above the code editor), and you should see some changes in the DAW and the Console. When you press run, all of the instructions in your code are carried out. The console should tell you which audio clips have been added to the DAW. In the DAW, press the play button to hear the music.

Adding Comments

Let's make a small modification to the current project. We'll add our name to the project. On lines 1-10, notice that each line starts with two forward-slashes: `//`. The `//` tells the computer not to execute any code to the right of it on that line. This is called a **comment**. Comments are used by programmers to make notes on their code, for them or other programmers to read later. On line 5, to the right of "author:" type your name.

EarSketch projects are developed largely for personal expression, so if you're working alone on a project you might use your own preferred commenting style. Bigger programming projects involving large-scale distribution will often involve different standards and methods. Later in this course, we will practice collaborative creative work in EarSketch. As you scale up, it may help to agree with your team upon standards for commenting.

The DAW in Detail

Take a look at the DAW. The DAW consists of several items:

Playhead: The red line, which represents your playback location in the timeline. The play button will start playback at the playhead's location.

Transport Controls: The blue buttons at the top left of the DAW. You've probably seen most of these in a media player like iTunes. From left to right, the buttons are:

- **Play/Pause:** Press this to hear the music you've added. Playback begins at the playhead.
- **Reset:** Press to jump the playhead back to the beginning.
- **Rewind:** Jump back in the timeline.
- **Fast-Forward:** Jump ahead in the timeline.
- **Loop:** When the playhead reaches the end of the timeline, automatically start playing from the beginning again.
- **Toggle Metronome:** Play a click track over your music.

Measure Numbers: At the top of the DAW timeline, there is a horizontal series of numbers. If this were a normal timeline, the numbers would represent minutes and seconds; however, here they represent measure numbers. A **measure** is a unit of musical time that depends on the speed (a.k.a. **tempo**) of a song.

The tempo has to be specified in every script. More on this in the **Rhythm** section. For now, think of a measure as a block of time. This is how we tell EarSketch where to place our audio clips. Click on a measure number to move the playhead to it.

Audio Clips: If you have added music to the DAW, the DAW should display some boxes with blue squiggly lines inside. These are audio clips. They provide a visual representation of the sounds they contain.

Tracks: Every audio clip is placed on a specific track. Tracks are the rows that run across the DAW; they are numbered on the left. Tracks help you organize your sounds by instrument-type: for example, in a recording studio you would record each instrument (vocals, lead guitar, rhythm guitar, bass, drums, etc.) on a separate track. You can only have one audio clip at a given time on each track, so having multiple tracks also means you can overlap them.

Effects Toggle: Show or hide the effects added on each track, if you have any. Note that the effects will still play back; the toggle is just for visuals.

Solo/Mute: Next to each track number, the "S" and "M" stand for solo and mute. Mute turns off playback for that track, and Solo turns off playback for all other tracks.

What is Programming?

A computer **program** is a sequence of instructions that the computer executes, and that is used to accomplish a specific task or set of tasks. Programming is the process of designing, writing, testing, debugging, and maintaining the code of computer programs. This code can be written in a wide variety of computer programming languages. Some of these languages include Java, C, Python, and JavaScript.

Programming languages consist of a collection of words and symbols that the computer can understand, and a syntax for organizing them. You can think of this like the vocabulary and syntax of spoken language. At the deepest level, computers operate in combinations of 1s and 0s, or binary. Thankfully we don't have to write programs in binary, as it would be very hard for us to understand! Just as a human might translate from English to French, the computer can translate human-readable programming languages into binary code.

Computer programs implement algorithms; in other words, a computer program describes a set of instructions for a computer to follow.

We can think of the different lines of our code as being individual instructions that we give to the computer. The computer follows these instructions explicitly to execute our written code.

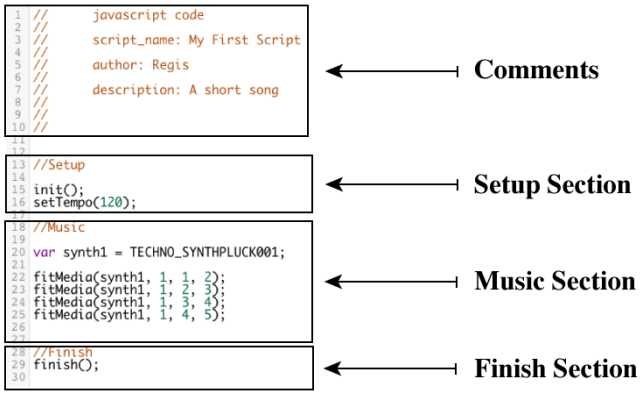
Programs are developed for a wide variety of purposes. In EarSketch, our goal in developing programs is creative musical expression. Computer pro-

grams can be built to deal with many kinds of **inputs** and **outputs**. In EarSketch, we focus on creating output in the form of digital audio, which you can listen to in the browser or save to your computer.

Composing In EarSketch

Now for the fun part: making your own music in EarSketch.

In this section we will familiarize ourselves with the basic way an EarSketch project is built. To make things easier, we will structure all of our sample projects in roughly the same way:



1. **Comments Section**
    - You can use comments anywhere in your code, but a block at the top is usually used to describe the whole project.
2. **Setup Section**
    - This code tells the DAW how to prepare to make music. `init()` initializes, or turns on, the DAW. `setTempo()` allows you to choose a tempo for the project.. Every project with music in it *must have* these parts in the setup section.

3. **Music Section**

- The most important section. All of the details of your composition go here.

4. **Finish Section**
    - Every project *must have* a `finish()` function at the end. It tells the DAW that you are done composing and are ready to play it.

Let's make our own script using the structure above.

1. To begin, open the Code Editor, and click *Options*. Select *New Script*.

This will give you a template to fill in with music. Make sure you are signed in, then save your script.

Add a script name, your name, and a description in the comments section. Label your sections with comments, as in the image above. Empty lines don't effect how the program runs, so add as many as you need to make space for each section. The music section will be empty for now.

2. In the template, you'll see some words that have parentheses after them, like`setTempo()`. These are all **functions**, and they stand for a set of instructions to be executed together. Their names are often verbs (*initialize*, *set*, *finish*, etc.), although this is not required. You can think of them as the verbs of the programming language. EarSketch comes with many functions to make music with, and you will also learn to write your own functions. More about this in the next lesson.

The parentheses after the function name tell the computer to **call**, meaning execute, the function with that name. They also provide a space to add arguments. An **argument** has some effect on the instructions that the function executes. Some functions take arguments, some don't, and some are flexible about the number of arguments they take. When one function takes multiple arguments, they are separated by commas like this: `myFunction(argument1, argument2, argument3)`. The order is important!

The function `setTempo()` comes with a default argument of 120, but let's change it to 100. This sets the tempo of our project to 100 beats per minute. As you can see, the name of a function tells you what it does.

3. In our music section, let's call a function named `fitMedia()` to add sound to the DAW. `fitMedia()` requires four arguments: clip name; track number; starting measure; ending measure. In other words, you tell it the name of the audio clip you want to add to the DAW, which track to put it on, and which measures to put it between. For now, just type in `fitMedia()` without any arguments.

4. Let's pick some audio clips to add with `fitMedia()`. Open up the sound browser, and select the Sounds tab. We're going to do some **instrumentation:** choosing the combination of instruments for a composition. Let's go with a typ-

ical rock instrumentation: drums, bass, and guitar. Search in the sound browser for clips with 'drum' in the name, and find one you like by pressing the play button next to them. Click your text cursor in the parentheses of `fitMedia()`, and use the copy

button (next to the clip you want to add to your song) in the sound browser to add your chosen clip as an argument. You can also just type it, if you like, but make sure it is in all-caps! Audio clip names MUST BE IN CAPS.

CAPITALIZATION

Programming languages like JavaScript are **case-sensitive**, meaning that the computer recognizes the difference between capitalized and uncapitalized letters. `myfunction()`, `myFunction()`, `MyFunction()` and `MyFuNction()` refer to four completely different things. This applies to everything you type (except comments), not just functions. This is a common mistake; check for it if you run into problems.

In addition, the convention for naming things like functions and variables is to use **camel-caps**: the first word is lower case, and subsequent words are capitalized, as in `exampleFunctionName()`

5. We want the drum clip added to track 1, so the 2nd argument in `fitMedia()` should be the number 1. Our clip should start on the first measure, so the 3rd argument is 1. If we want our clip to play for one measure, the end measure should be 2 (meaning we stop at the beginning of measure 2). So our 4th argument should be 2. Your function should look something like this: `fitMedia(Y01_DRUMS_1, 1, 1, 2)`.

6. Now we'll add some bass. We will basically repeat steps 3-5, by adding a new `fitMedia()` call on the line below our previous one. In this new `fitMedia()` , find a bass clip to add as an argument, just like you found the drums. Clips that are in the same folder in the sound browser are designed to sound good together, so try to choose clips from the same folder. We can only have one clip at a time on a given track, so tell `fitMedia()` that this goes on track 2. Finally, choose a range of measures to fit the clip into, probably between measures 1 and 2 like before.

7. Repeat step 6, but add a guitar clip instead of a drum clip. With any luck, your code looks something like ours below, but with different clips. Press 'Run', and then play the music you made!

```
//      javascript code
//
//      script_name: Opus 1
//
//      author: The EarSketch Team
//
//      description: A magnificent measure
//
//
//

// Setup Section

init();
setTempo(100);

// Music Section

fitMedia(Y01_DRUMS_1, 1, 1, 2);
fitMedia(Y11_BASS_1, 2, 1, 2);
fitMedia(Y11_GUITAR_1, 3, 1, 2);

// Finish Section

finish();
```

Why Learn to Program?

When we write computer programs in EarSketch, we use the Python or JavaScript programming language. You are in JavaScript mode. JavaScript is one of the ten most popular programming languages in the world. It is primarily used in web development, but is also widely used for many other purposes such as game development. Almost every website uses JavaScript, in conjunction with HTML and CSS, to build the interactions with the user (front-end web development). Many also use it to manipulate data on the server side (back-end web development). Don't be confused by the name: it is an entirely different language than Java!

Programming involves a lot of creativity, so it fits well with making music. The skills you learn from practicing it can be extremely useful. You'll learn to think in both a structured and creative way, which is a valuable combination. Learning programming also opens the door to many great and lucrative careers. Nearly every field today uses computer programs. Whether you're interested in biology, physics, finance, math, robotics, education, making games,

graphic design, music, literature, chemistry, or any other field, knowing how to program will help you to get a great job, succeed, and become more well-rounded. Most importantly, anyone (incuding you) can learn to program. Like learning a musical instrument, it takes consistent practice to make progress. Don't get discouraged if you get stuck, this is part of the process: ask for help in your class or look online.

So far, we have learned about what it means to program, and how we can program to make music in EarSketch. We type code into the code editor panel, press run, and then play our music in the DAW panel. We can find sound clips to use in our code in the sound browser, and we refer to them in our code by typing their name (a constant, in all caps).

We learned that commented code is not executed by the computer, but is useful to the programmer. Functions are instructions for the computer to take some action, such as `fitMedia()` . Some functions take arguments, which specify exactly what the function should do.

In the next section, we will look at a few more types of data we use in EarSketch to make music.

The Building Blocks of a Program 2

Rhythm

When we talk about the **rhythm** of a song, we are describing how the music moves through time. Musicians have many words to describe rhythm, including: tempo, meter, measure, beat, sub-beat. These are useful in DAWs like EarSketch because they help you to organize the elements of your music in time.

A **beat** is the basic unit of time in music. If you have ever clapped along to a song, you were probably clapping on each beat. So how long does a beat last? The length depends on the overall speed of the song, called the **tempo**. Tempo is measured in beats per minute (bpm). If we are clapping at 60 bpm, then each beat lasts one second. At 120 bpm, each beat takes half a second. The higher the bpm, the faster the song, the shorter the duration of each beat.

BEATS, BEATZ, BEETS?

You might have noticed that the word 'beat' is used in several ways. The beat described above is a unit for measuring musical time.

The *other* meaning is short for a **drum-beat**: a repeated rhythmic pattern for a set of percussive sounds. You can usually tell which kind of 'beat' someone is talking about from the context.

Copy the following code into your code editor, press run, and press play. Press the loop button to keep the pattern repeating. Try counting '1, 2, 3, 4', with one count for each hit of the kick drum. Notice that the timeline starts at measure 1 and ends at measure 2.

```
// javascript code
//
// script_name: Beats
//
// author: The EarSketch Team
//
// description: Counting beats and sub-beats in a measure.
//
//
//

//Setup Section
init();
setTempo(120);

//Music Section
fitMedia(TECHNO_LOOP_PART_002, 1, 1, 2); // Each kick drum hit lasts a quarter note

// fitMedia(TECHNO_LOOP_PART_031, 2, 1, 2); // Each cymbal hit lasts a 16th note: 1,

//Finish Section
finish();
```

Beats are grouped into **measures**, with the same number of beats in each measure. In EarSketch, measures always have four beats. You may have noticed above that you can clap along to a song in quite a few ways that seem to fit. For example, if you clap once every 4 beats, you are clapping once every measure.

Often in music, a measure is said to have a duration of a **whole-note**. If we have 4 beats in a measure, then each beat is a **quarter-note**. We can also have half notes, eighth notes, sixteenth notes, and so on.

What about if you clap twice per beat, or 4 times per beat? We call these divisions of a beat **sub-beats**. In the previous example, uncomment (delete the // marks on) line 19 and run the code again to hear the sub-beats in the hi-hat (cymbal) part. The hi-hat cymbal plays 16 times per measure: each one is a 16th note long.

FIGURE 2-1

Data Types

Computers store and process information, and we call this set of information **data**. It's useful to know what kinds of data you can put in a program. If we think of a program as a recipe, then the **data types** are the kinds of ingredients you can use. For example, some common ingredient types are vegetables, meats, herbs. Paints, on the other hand, are not a valid ingredient type (in normal restaurants). The same goes for programming languages: they can only work with certain kinds of data. The basic data types that most programming languages can understand are:

- Numbers
- Strings
- Variables
- Constants
- Functions
- Arrays

Everything you build in EarSketch will involve some combination of these. In the following section we will focus on using numbers, variables, and constants to make music in EarSketch.

Numbers

The fundamental data type in computing is the **number**. At a deeper (or "lower") level, everything in a computer is encoded as a binary number. At a higher level, where we will be working, numbers are great for describing rhythm to the computer. Think of how we verbally described rhythm in the previous section: as *numbers* of measures, beats, and sub-beats. Every EarSketch script must include a setTempo() function, with some number in the parentheses. This tells

the computer how fast to playback the music. Try changing this number yourself, in the example below, and listen to it (press the loop button again).

```
// javascript code
//
// script_name: Beats
//
// author: The EarSketch Team
//
// description: Counting beats and sub-beats in a measure.
//
//
//

//Setup Section
init();
setTempo(120);

//Music Section
fitMedia(TECHNO_LOOP_PART_002, 1, 1, 2); // Each kick drum hit lasts a quarter note

// fitMedia(TECHNO_LOOP_PART_031, 2, 1, 2); // Each cymbal hit lasts a 16th note: 1,

//Finish Section
finish();
```

TYPES OF NUMBERS: FLOATS AND INTS

There are often multiple types of numbers in programming languages, the most common being **integer** and **floating point**.

Integers (often abbreviated as "int") are positive or negative whole numbers, including 0: for example, -23.

Floating point (or "float") numbers are positive and negative numbers with a decimal (or fractional) component. In other words, rational numbers. For example: 3.14159, or -21.0. All values of the number type in JavaScript are stored as floats, so the integer `10` is stored as `10.0`.

Variables

In EarSketch, we are doing **Algorithmic Composition**. This is like writing up musical recipes, which the computer will then cook up to produce a delicious

piece of music. When making a recipe, we might want to try adding different amounts of our ingredients. For example, 1 vs. 2 cups of butter. Values like this that can change are called **variables**. In cooking, your variables might be called `temperature`, `timeInOven`, or `amountOfButter`. To set the temperature, you could write `temperature = 400`. In EarSketch, variables are often used to hold musical values, like `measureNumber` or `trackNumber`.

VARIABLES AND MEMORY

A variable creates a space in the computer's memory to store something. What makes them useful is that you can change what they store.

In math, a variable usually represents a number. In a programming language, variables can represent almost anything, including numbers and clip names.

`track = 1` creates a variable named "track", and assigns the number `1` to it. If we wrote `fitMedia(Y_11_BASS_1, track, 1, 5)` in the same program, it would be interpreted as `fitMedia(Y_11_BASS_1, 1, 1, 5)`. If we wanted our clip to be added to track `2` instead, we could simply type `track = 2` above our `fitMedia()` call.

How can we use this to make music? Take a look at the example below and try to figure out what is happening.

```
//              javascript code
//
//              script_name: Variables
//
//              author: The EarSketch Team
//
//              description: Using variables to store measure numbers
//
//
//

// Setup
init();
setTempo(80);

// Music
var startMeasure = 1;                   // Here, we assign number values
var endMeasure = startMeasure+2;        // to our variables.

fitMedia(HIPHOP_DUSTYGROOVE_001, 1, startMeasure, endMeasure); // Now we can use our variable
```

```
fitMedia(HIPHOP_DUSTYCOINLEAD_002, 2, startMeasure, endMeasure);

// Finish
finish();
```

We created 2 variables, `startMeasure` and `endMeasure`, and assigned values to them using the **assignment operator**: =. Notice how endMeasure's value depends on startMeasure's.

We could have named our 2 variables x and y, or almost any other name you can imagine, and the program would have run exactly the same. However, names like x and y don't tell us anything about what the values they store are being used for, so it makes the code much less human-readable.

Try to add another set of clips after the current last measure, using variables as arguments. There are many approaches to this. When you're done, take a look at our solution below.

```
//              javascript code
//
//              script_name: Variables (continued)
//
//              author: The EarSketch Team
//
//              description: Using variables to store measure numbers
//
//
//

// Setup
init();
setTempo(80);

// Music
var startMeasure = 1;            // Here, we assign values to our variables.
var endMeasure = startMeasure + 2;

fitMedia(HIPHOP_DUSTYGROOVE_001, 1, startMeasure, endMeasure);       // This adds m
fitMedia(HIPHOP_DUSTYCOINLEAD_002, 2, startMeasure, endMeasure);     // Measures 1
fitMedia(HIPHOP_DUSTYGROOVE_001, 1, endMeasure, endMeasure + 2);     // Measures 3
fitMedia(HIPHOP_DUSTYCOINLEAD_002, 2, endMeasure, endMeasure + 2);   // Measures 3

// Finish
finish();
```

In summary, variables are used to store a value, and to name it. This value can be of any data type. They tell the computer to create a space in its memory for a value to be stored, and they give a name to that space so it can easily be referred to. You pick the name and the value. We could technically use a variable called `measureNumber` to set the tempo or any other value, but that would be pointless and confusing. Give your variables names that describe what they will be storing.

Continually editing and evaluating your work is important in both music-making and programming. This is called **iterative design**. In EarSketch, we do this by listening to our music, editing the code, listening to our changes, editing, and so on. Variables help us generalize parts of our code, which makes editing easier. For example, we can quickly swap out different clips to see what they sound like in our song, even if they are used in many different places in the song.

Constants

A constant stores values that never change. In EarSketch, they are used to refer to audio files, which you can add to your project. The "value" that these constants refer to is the address for a specific sample. If we changed this value, the name wouldn't correspond with the correct sample anymore. So we don't change them. By convention, their names are capitalized.

`TECHNO_SYNTHPLUCK_001` is a constant. It refers to an audio file that can be used within an EarSketch project. The actual audio file is located on the EarSketch server. Its unique location on the server is described by a file path (for example, C:/Program Files/EarSketch would specify the EarSketch program being located in the Program Files folder located on the C drive). A file path for an audio file can thus be a complicated way to refer to the file. But EarSketch assigns that long file path to a single value that never changes: a constant.

The name of the data type explains it all: the value of a variable varies, and the value of a constant stays constant.

3 The Core EarSketch Functions

The EarSketch API has many functions to help you compose music. A complete description of each one can be found in EarSketch API document, which you can find in the API tab of the Sound Browser. We have already looked at `fitMedia()`, which lets you add audio clips to the DAW. In this section, we will learn to use the two other main EarSketch functions: **`setEffect()`**, and **`makeBeat()`**.

Effects

With `fitMedia()` we focused on composing music by arranging different audio clips in the DAW. As a composer and producer, you'll also want to not only pay attention to the order and arrangement of clips in your project, but also the characteristics of those sounds. One way to change the quality of these sounds in your project is by adding **effects**. Audio effects are analogous to photo filters. They manipulate the audio to varying degrees. Listen to this reference clip with no effects, and then compare it with the clip below that has a **distortion** effect applied:

No effect:
audioMedia/reference.mp3

Distortion effect:
audioMedia/distortion2.mp3

To add an effect to a track in our DAW, we use the function `setEffect()`. As we experiment with effects in this section, we will hear how the sound changes as different effects are applied to it. Below, our code adds a **delay** effect to track 1:

```
// javascript code
//
// script_name: Effects-Delay
//
// author: The EarSketch Team
//
```

```
// description: A track with delay effects
//
//
//

//Setup

init();
setTempo(120);

//Music

fitMedia(DUBSTEP_LEAD_003, 1, 1, 13);

setEffect(1, DELAY, DELAY_TIME, 500);   // Adds a delay (echo) effect, at intervals

//Finish

finish();
```

A delay effect plays back the original audio as well as a delayed, quieter version of the original that sounds like an echo. After the first echo it plays an echo of the echo (even quieter), then an echo of the echo of the echo (still quieter), and so on until the echo dies out to nothing.

With the delay effect, we can control how much time passes between each echo (delay time). If we set the delay time to match the length of a beat, or some division of the beat, we can create rhythmic effects with delay. In our example, each delayed version of the original clip comes in 500 milliseconds after the previous one. How did we get this number? It is equal to one beat. Our tempo is 120 beats per minute, and since there are 60 seconds in a minute, we have 60 seconds per 120 beats. Simplified, this gives us 60 seconds/120 beats = 0.5 seconds per beat. This is the length of each beat at 120bpm. So we can set our delay time to this length, but we need to convert it to milliseconds. If 1 second = 1000 milliseconds, then 0.5 seconds = 500 milliseconds, which is the delay time we pass to `setEffect()`.

For a challenge, try coding this calculation of beat length in your script. The equation is: (60/tempo) x 1000 = one beat in milliseconds.

setEffect

How does `setEffect()` work? Similarly to `fitMedia()`: it takes 4 arguments that specify what exactly it does. You can think of the arguments as settings. With `setEffect()`, our arguments are:

1. **Track Number**: The effect is added to this track.
2. **Effect Name**: This is the specific effect being used.
3. **Effect Parameter**: Each effect has several settings of its own, so you'll have to choose which one you want to set. Note that you can set multiple parameters on a single effect, in combination.
4. **Effect Value**: Effect parameters usually accept a range of numbers, so you can set that here. For example, if your effect is volume then this specifies how loud or quiet to make the track.

Let's use setEffect to mix a composition. In music production, **mixing** is the process of balancing multiple audio tracks so that they sound cohesive when played together. You can do this in EarSketch by using effects.

In the code below, we assigned several audio clips (constants) to variables. We then use these variables in a number of `fitMedia()`: calls. Notice how we organized our `fitMedia()` calls by track number, to make it easier to read. Run the code and have a listen:

```
//              javascript code
//
//              script_name: Techno Mix
//
//              author: The EarSketch Team
//
//              description: Mixing by adjusting volume with setEffect.
//
//
//

//Setup

init();
setTempo(120);

//Music

var introLead = TECHNO_ACIDBASS_002;   // Store clips in variables
var mainLead1 = TECHNO_ACIDBASS_003;
var mainLead2 = TECHNO_ACIDBASS_005;
var auxDrums1 = TECHNO_LOOP_PART_025;
var auxDrums2 = TECHNO_LOOP_PART_030;
var mainDrums = TECHNO_MAINLOOP_019;
var bass = TECHNO_SUBBASS_002;

fitMedia(introLead, 1, 1, 5);   // Add clips to DAW
fitMedia(mainLead1, 1, 5, 9);
fitMedia(mainLead2, 1, 9, 17);

fitMedia(auxDrums1, 2, 3, 5);
```

```
fitMedia(auxDrums2, 2, 5, 8);
fitMedia(auxDrums2, 2, 9, 17);

fitMedia(mainDrums, 3, 5, 8);
fitMedia(mainDrums, 3, 9, 17);

fitMedia(bass, 4, 9, 17);


//Effects

// setEffect(3, VOLUME, GAIN, 10.0);   // Adding different volume gain to each trac
// setEffect(4, VOLUME, GAIN, 12.0);

//Finish

finish();
```

It sounds good, but if we wanted to DJ with this we would probably want a more powerful drum and bass section. Those instruments contribute heavily to the rhythmic feel of the music, which is important for dancing. We currently have the main drums and bass in tracks 3 and 4. Let's turn the volume up on those tracks by uncommenting the setEffect calls in the effects section. Delete the `//` characters in front of both `setEffect` calls, run the script, and press play to hear the difference.

EarSketch supports a variety of effects that are common in music production. For a complete list of the effects and how to use them, see **Chapter 16**.

Making Custom Beats

`fitMedia()` allows us to make music out of audio loops. There are many great possibilities when using `fitMedia()`, but when we want to make music (and particularly drum beats) note by note, we'll want another tool. **`makeBeat()`** is EarSketch's function for this: instead of composing at the measure-level, we can work at the note-level. In music production, this approach is often called **step sequencing** and is done with a drum machine or a groove box (see the image below). In EarSketch, `makeBeat()` gives us a powerful way to do step sequencing.

FIGURE 3-1

A Roland TR-808 drum machine.

Strings

To use `makeBeat()`, we first need to understand the **string** data type. In JavaScript, a string is a series of characters with quotation marks around it, like `"Hello World!"` or `"This is test sentence #1"`. Single or double-quotes are both fine. Strings are often used in programming to represent non-numerical data such as words, but you can also have numerical characters in strings. For example, when you type your address into a website, it probably starts by saving it as a string. That string contains several types of characters: numbers, spaces, letters, punctuation marks: `"123 Fake St."`

Note that 5 and `"5"` mean very different things to the computer: 5 is a number that it can do math with, while math operations usually* won't work with the string `"5"`. Just like with numbers (and other types of data), strings can be assigned to variables: `address = "123 Fake St."`

We use strings with the `makeBeat()` function to define rhythmic patterns, which we call beat patterns.

*In some cases, the language will automatically convert it to the correct type for arithmetic, but you should not rely on this.

Beat Patterns with Strings

Beat patterns in EarSketch use strings to refer to sub-beats of a measure in order to place clips at specific places in the measure, as well as define the clip's play length, all at once. Here's an example of a beat pattern using a string, assigned to a variable called myDrumBeat:

```
myDrumBeat = "0-00-00-0+++0+0+"
```

Every character stands for one sixteenth-note sub-beat of a measure.

- `0` starts playing the clip.
- `-` is a rest, meaning that there's nothing being played.
- `+` extends the audio clip into the next sixteenth-note sub-beat, so it should always follows either a `0` or a `+`.

Going from left to right, the above beat string is telling EarSketch that it should:

0	play the clip for one sixteenth of a measure
-	rest for one sixteenth
0	play for one sixteenth
0	play for one sixteenth again
-	rest for one sixteenth
0	play for one sixteenth
0	play for one sixteenth again
-	rest for one sixteenth
0+++	play the clip for four sixteenths (or one quarter)
0+	play for two sixteenths (or one eighth)
0+	play for two sixteenths (or one eighth)

It should be noted that although a 16-character beat string does comprise a measure, beat strings do not have to be this length. For example, if you want a

cymbal crash that lasts for one beat, the beat string `"0+++"` is perfectly acceptable.

Take another look at the image of the Roland TR-808 drum machine above. Do you see anything in common with `makeBeat()` strings? The 16 colorful buttons work much like a `makeBeat()` string! The machine reads through all 16 from left to right, only making a sound for the ones that have been pressed down. Each depressed button on that machine works like a 0 in a beat string.

makeBeat

Now we are ready to use `makeBeat()`. Like our other functions, `makeBeat()` also takes four arguments:

1. **Clip Name**
2. **Track Number**
3. **Measure Number**: note that it only requires a starting measure, as the length of the string determines where the end measure will be.
4. **Beat String**

Starting with an empty script, try to add your own makeBeat call, and add it to measure 1 in track 1. You will have to choose your own clip, and make a beat string. The EarSketch Sound Browser contains a type of clip designed for use with makeBeat: **one-shots**. These are typically the length of one drum hit. You can browse them by clicking “Artists”, then “MAKEBEAT”. Compare your script with ours below:

```
// javascript code
//
// script_name: Simple Beat
//
// author: The EarSketch Team
//
// description: Making a rhythm with makeBeat and a string
//
//
//

//Setup

init();
setTempo(120);

//Music

makeBeat(DUBSTEP_FILTERCHORD_002, 1, 1, '0--0--000--00-0-');
```

```
//Finish

finish();
```

This only adds one instance of our beat string to the DAW. Try calling makeBeat several times (with different measure number arguments) to repeat your pattern. You could even add a complementary rhythm on a new track, like this:

```
// javascript code
//
// script_name: Multi Beat
//
// author: The EarSketch Team
//
// description: Using several makeBeat calls, and overlapping rhythms
//
//
//

//Setup

init();
setTempo(120);

//Music

makeBeat(DUBSTEP_FILTERCHORD_002, 1, 1, "0--0--000--00-0-");
makeBeat(DUBSTEP_FILTERCHORD_002, 1, 2, "0--0--000--00-0-");
makeBeat(DUBSTEP_FILTERCHORD_002, 1, 3, "0--0--000--00-0-");

makeBeat(OS_CLOSEDHAT01, 2, 1, "-00-00+++00--0-0");
makeBeat(OS_CLOSEDHAT01, 2, 2, "-00-00+++00--0-0");
makeBeat(OS_CLOSEDHAT01, 2, 3, "0--0-00--00--000");   // This rhythm gives us some

//Finish

finish();
```

4 Debugging

What is Debugging?

As a programmer, you will make many mistakes that will cause your code to work incorrectly, or not run at all. This is common: even the most experienced of programmers make mistakes! These mistakes are called **errors** or **bugs**. The process of finding and fixing these is called **debugging**.

There are 3 types of errors you will encounter while programming:

1. **Syntax errors**: Your program doesn't run, because your code does not follow the syntax rules that JavaScript requires to interpret your code. The **syntax** of a programming language is like the grammar of a spoken language: it is a set of rules of how to combine its symbols. For example, every open parenthesis `(` must eventually be followed by a closing one `)`.
2. **Runtime errors**: Your program starts to run, but halts because of an error.
3. **Logic errors**: Your program runs, but it doesn't do what you expected it to do.

There are many ways to debug a program depending on the kind of code you are writing and your development environment. In EarSketch, you can usually catch logic errors by looking at the DAW: you can see if the clips you meant to add were actually added in the right places. For syntax and runtime errors where we don't get to see anything in the DAW, we will use **print debugging**. Print debugging uses the `println` function to send messages to the console.

Using the Console

The console is used to get information about the **state** of a program as it runs. The state refers to what the memory of your computer is holding as it runs the program.

The program is changing what is stored in the memory as each statement of your code is executed, which is another way of saying the state changes. You

might think of the memory as the collection of every variable in your program; these variables might hold different numbers or strings throughout the execution of your program.

All good software development environments (also called Integrated Development Environments, or IDEs) have some kind of console.

Let's practice using the console. Make sure your console is open: it should be at the bottom of your window. Now try running a script with just one fitMedia call, as well as the usual 'Setup' and 'Finish' sections. If all goes well, the console should display something like this:

```
EarSketch Client Beta V 2.77
Running script ...
```

The console has displayed three things: it tells us which version of EarSketch we are running (2.77), that it is indeed running our script, that it has finished running, and which audio files it has loaded into the DAW.

It looks like all of these appeared in the console at the same time, but that is because the computer runs quickly. If we were to step through our program line by line, we would see them appearing one at a time.

If EarSketch encounters an error while running your code, the console will display information about what caused that error. See if you can spot the error in this script:

```
// javascript code
//
// script_name: Error Script
//
// author: The EarSketch Team
//
// description: This script causes an error... check the console
//
//
//

//Setup
init();
setTempo(120);

//Music
makeBeat(OS_CLAP01, 1, 1, "0--0++0-);

//Finish
finish();
```

Now, run the above code. The console should tell you something like this:

EarSketch Client Beta V 2.77
Running script ...
ERROR: ParseError: bad token on line 17

You don't need to know what 'ParseError' or 'bad token' mean. Just note that it says 'ERROR' and points to 'line 17'. We forgot to add a closing quotation mark to the right of the beat string on line 17, a common kind of error.

You don't need to know what 'ParseError' or 'bad token' mean. Just note that it says 'ERROR' and points to 'line 19'. We forgot to add a closing quotation mark to the right of the beat string on line 19, a common kind of error.

Printing in the Console

Printing to a console is useful for learning about the state of your program. Rather than only getting certain information from the console (as in the example above), you can tell it to print exactly what you want at any point in the program.

The `println()` function tells the console to display something. This "something" can be any kind of data: a string, numbers, the value a variable is holding, etc. The argument is the data that you want to be printed. `println()` will **evaluate** the argument, and print the final value. Evaluating means showing the statement in its most basic form. For example, 2+2 is evaluated as 4; a variable x (holding the number 4) is evaluated as 4. The syntax is the same as with other functions in JavaScript:

```
println('String to be printed');
```

This would display *String to be Printed* in the console.

Run the following code in a new tab:

```
// javascript code
//
// script_name: Printing Demo
//
// author: The EarSketch Team
//
// description: Using println to print messages in the console
//
//
//

//Setup
init();
        // (We don't need setTempo here, since there's no music)
```

```
//Printing
println(3 * 4);          //Prints the result of 3*4: 12

var x = 3 * 4;
println(x);              //Prints the value of x, which is also 3*4, on the next l

var y = 2;
println(x / y);          //Evaluates x/y, then prints it on the next line

//Finish
finish();
```

Take a look at the console. You can see how the computer is evaluating the arguments you give to `println()`, in the order they were called.

`println()` doesn't need an argument that evaluates to a number; it works with any data type. With the `println()` function, you can not only find errors, but also gain a better understanding of how your program is running.

The Debugging Process

You can use printing, commenting, and the console to find errors in your code. As your program gets larger, it can become increasingly difficult to find the source of error, so it helps to have some strategies in case you get stuck. Try following these steps if you run into an error:

Reproduce the error:

Run your code. In EarSketch you should get the same error every time you run the same code, so that will be enough to reproduce the error. However, one exception to this is when using random number generators (covered later in the curriculum), where only certain values will cause errors.

In other programming environments there may be other inputs that only occasionally cause your program to throw an error, so you would also need to figure out the kind of input that is causing errors.

Read the console for clues:

The console will often tell you which line of code caused the error, and what type of error it is.

Locate the error in your code:

If the console provided a line number, take a look at that line in your code.

A useful strategy for checking if a line (or group of lines) in your code is causing an error is to **comment out** that code. This disables that part of your code from running. If your code works fine without it, then you've found the bug. We've seen single-line comments, but you can also create **multi-line comments** by surrounding the lines of code you want to comment with these symbols: **/* */**

QUICK COMMENTING

You can quickly comment or uncomment code by clicking on a line (or highlighting multiple lines), then using these keyboard shortcuts:

Mac: **Command + /**

Windows: **Control + /**

Another useful strategy is to get the state of your program by printing. The goal is to ensure you understand the state of the program that is causing the error, and printing can reveal the state. Read through the problem section of your code, and try to follow the logic. Think about which variables are being assigned which values, and make sure those are valid values. Insert print statements where you are unsure of the logic, getting the value of certain variables at those points. In this way, you can check your understanding of the program against what is actually happening in it.

If the console gives you no clues about the location of your error, you may have to take educated guesses at what is causing the error. Walk through the sections of your code, making sure the program flow makes sense. If you are uncertain about a section, go through it line by line, just like the computer reads it. Insert print statements along the way, to check the program's state.

As your program grows, this process can become increasingly complex. An error might be caused by several interacting lines of code in different parts of your script.

Squash the bug:

You might immediately recognize the problem and be able to fix it. Hooray! In the likely case that you don't see the problem, check for syntax errors. This often involves closing parentheses, quotations, or brackets. Other common errors include spelling function or variable names wrong (including incorrect capitalization), adding media to the wrong measure number, and counting to the

wrong number in a loop index (you'll learn about loops in the next section). Edit the offending code, and try running it to see if it works.

Ask for help:

It can be a good learning experience to try to solve a problem on your own for a bit, but if you find you have spent too much time on a bug then by all means ask someone for help! Vast swaths of the internet are dedicated to helping programmers debug their code (nice to know you're not the only one), so you can usually find answers there. Better yet, ask a teacher or a classmate. A fresh pair of eyes can do wonders for spotting mistakes.

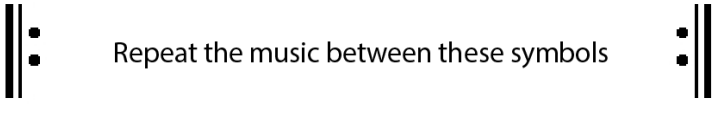
Looping 5

Repetition in Music and Technology

Almost all music uses repetition. Repetition makes things more familiar for the listener: you hear something again that you already know.

For instance, a drummer might repeat a series of rhythms many times to form a drum beat. A band will often play certain parts of a song more than once, such as the chorus. So, you can repeat things on many levels in music: from small amounts of time (a few hits of a drum) to longer durations (entire sections of a song). Technology makes repetition easy. With guitars, a loop pedal can be used to record a sample of your playing, and then plays it back over and over again. Computers also happen to be very good at repetition.

In music notation, you can use a repeat sign that tells the performer to play a measure again.



Notation like this is efficient: you avoid writing the same measure over and over again. Similarly, programming languages have a notation that tells the computer to execute a section of code repeatedly. This repetition is called **looping**. In JavaScript it looks something like this:

```
for (var i = 0; i < 2; i++) {
```

```
    //Repeat any code in here (between the {} curly braces) several times.
}
```

A Loop Example

In EarSketch, we've been creating repetition in our music by simply typing `fitMedia()` or `makeBeat()` again and again, with different measure numbers:

```
// javascript code
//
// script_name: Drum beat (no loops)
//
// author: The EarSketch Team
//
// description: Musical repetition created without code loops
//
//
//

//Setup
init();
setTempo(120);

//Music
var beat = "0-00+00-0+++0+0+";
var drum = ELECTRO_DRUM_MAIN_BEAT_008;

// All of these makeBeat calls could be replaced with a single one placed in a loop

makeBeat(drum, 1, 1, beat);
makeBeat(drum, 1, 2, beat);
makeBeat(drum, 1, 3, beat);
makeBeat(drum, 1, 4, beat);
makeBeat(drum, 1, 5, beat);
makeBeat(drum, 1, 6, beat);
makeBeat(drum, 1, 7, beat);
makeBeat(drum, 1, 8, beat);

//Finish
finish();
```

There is a better way to code: we can write this more concisely by using a loop. There are several kinds of loops, but we'll be learning about the **for-loop**. The code below creates the same music as the code above, using fewer lines of code.

```
// javascript code
//
// script_name: Drum beat (with loops)
//
// author: The EarSketch Team
//
// description: Musical repetition created with code loops
//
//
//

//Setup
init();
setTempo(120);

//Music
var beat = "0-00-00-0+++0+0+";
var drum = ELECTRO_DRUM_MAIN_BEAT_008;

// Using a loop instead of repeatedly writing similar lines of code. Much better!

for (var measure = 1; measure < 9; measure = measure + 1) {
 makeBeat(drum, 1, measure, beat);
}

//Finish
finish();
```

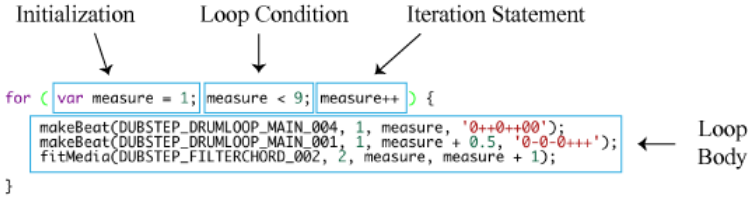
The code says "execute this statement several times, and increase the measure number each time." Each repetition of the code is called an **iteration**. If we didn't increase the measure number on each iteration, `makeBeat()`would keep adding clips to the same location, and the loop would be pointless.

Writing a loop is much faster than writing each statement out; imagine if we wanted to repeat this drum beat 1000 times! This demonstrates one of the main benefits of using coding for music: music often uses plenty of repetition, and coding allows you to repeat yourself very efficiently. Instead of writing 1000

lines of code, we can accomplish the same with only two or three. Let's look at the details of how loops work, so you can write your own.

Components of a For Loop

For-loops in JavaScript consist of 4 basic parts:



- **Loop Body:** The body of a loop contains the statements that you want to be executed repeatedly. It includes everything surrounded by the curly brackets `{ }`. You can loop as many statements as you like (you can even loop loops!).
- **Initialization**: Creates a variable to be used as a loop counter. A simple loop counter holds the current count of how many times we have looped through our code, so we don't loop forever (an **infinite loop** is a good way to crash a program!). We named it `measure` here, but you can name it anything reasonable you would like.
- **Iteration Statement:** This is how we update our loop counter, so it changes on each iteration (often by counting up). `measure++` is a short-cut for writing `measure = measure + 1`
- **Loop Condition:** This checks whether the loop should run again. If the statement is true, we keep looping. If the counter gets too high here, the statement will be false, and we exit the loop. The computer then continues executing code after the loop.

CODE BLOCKS: { }

A code block in JavaScript is simply a group of statements (one or more lines of code) that are meant to be run together. It is shown by placing curly brack-

ets { } around the statements. Control flow structures like for-loops use this notation to show which code will be looped.

Later, we will see that **Conditionals** also use this notation.

INDENTATION

Indentation in JavaScript does not affect how your code is executed; however, it is very important for keeping it human-readable. As a convention, all of the statements inside the body of a for-loop should have the same level of indentation: one more tab than the level of indentation the first line has.

We will see some other structures later, such as **Conditionals**, which also use this kind of indentation.

A loop is a **control flow** statement. Control flow is the order that the computer reads and executes code in. Up until now, the computer has executed our code sequentially (top to bottom, line by line), just like you would read a page of text. However, we can use statements like loops in our code to change the reading order: at the end of the loop code, it jumps back to the top of the loop.

Following the Control Flow

Let's walk through a code example that uses a loop, and follow the control flow from the beginning. Take another look at this example from above:

```
// javascript code
//
// script_name: Drum beat (with loops)
//
// author: The EarSketch Team
//
// description: Musical repetition created with code loops
//
//
//

//Setup
```

```
init();
setTempo(120);

//Music
var beat = "0-00-00-0+++0+0+";
var drum = ELECTRO_DRUM_MAIN_BEAT_008;

// Using a loop instead of repeatedly writing similar lines of code. Much better!

for (var measure = 1; measure < 9; measure = measure + 1) {
 makeBeat(drum, 1, measure, beat);
}

//Finish
finish();
```

The **interpreter** is the part of JavaScript that reads and executes our code. It starts at line 1, and goes down line by line unless we tell it to do otherwise. Lines 1-12 are either comments or empty, so it skips those. Lines 13-21 are executed in top-to-bottom order (again skipping commented or empty lines).

Things get interesting at line 22. We've reached our for-loop! This tells the interpreter that we will be repeating a block of code: our loop body.

On line 22, the `measure` variable (our loop counter) is assigned the number 1. Then, the loop condition is checked (is `measure` less than 9?). `measure` is indeed less than 9, which tells the interpreter that we should move into the loop body (inside of the brackets).

Since `measure` is currently 1, it executes `makeBeat(drum, 1, 1, beat)`. Once this line has finished, we have reached the end of our loop body, so we jump back up to line 22.

The increment statement `measure = measure + 1` is now applied, making `measure` equal 2. We check again if measure is less than 9; it is, so we move into the loop body again. Since measure is currently 2, it executes `makeBeat(drum, 1, 2, beat)`.

This looping process repeats until after `measure` is assigned to 9. At this point, `measure` is not less than 9 (meaning our loop condition is not met), so control jumps down to line 25, and continues to the bottom of our script in regular top-to-bottom order.

INCREMENTING AND DECREMENTING

In for-loops, we need to increment (or sometimes decrement) a counter. We wrote this above as measure = measure + 1, but there is an easier way.

- **`measure++`**
  - This adds 1 to `measure`'s current value. It has the exact same meaning as `measure = measure + 1`, but is faster to write. You will see this often.
- **`measure--`**
  - This subtracts 1 from `measure`'s current value.

What if we want to increment/decrement by a different number than 1?

We can use += or -=. For example:

- **`measure += 2`**
  - This adds 2 to `measure`'s current value. It is exactly the same as `measure = measure + 2`. You can also do `measure += 3` to increment by 3, or any number you want.
- **`measure -= 2`**
  - This subtracts 2 from `measure`. You can also decrement by any number you want.

Repeatedly adding beats or media to our project, as above, is just one of many possible uses for for-loops in EarSketch. In the following example we use two loops to add some clips and beats to the DAW, just as we saw before. However, we also use a third loop to add a panning effect on each track. If you have headphones available, use them to listen to this script (they will make the panning effect more noticeable):

```
// javascript code
//
// script_name: Panning Loop
//
// author: The EarSketch Team
//
// description: Adding music with loops; panning incrementally with loops
//
```

```
//
//

//Setup
init();
setTempo(130);

//Music
var drums1 = OS_KICK01;
var drums2 = HIPHOP_TRAPHOP_BEAT_001;
var drums3 = OS_LOWTOM03;
var synth1 = DUBSTEP_FILTERCHORD_002;
var synth2 = Y02_KEYS_1;
var guitar1 = Y09_WAH_GUITAR_1;
var guitar2 = Y61_WAH_GUITAR_1;


for (var measure = 1; measure < 5; measure++) {   //Add music to measures 1 through
        makeBeat(drums1, 1, measure, '0++-++00');
        makeBeat(drums2, 3, measure + 0.5, '0++0++++');
        fitMedia(synth1, 2, measure, measure + 1);
}

for (var measure = 5; measure < 9; measure++) {   //Add music to measures 5 through
        makeBeat(drums1, 1, measure, '0+++0+-0');
        makeBeat(drums3, 3, measure + 0.5, '00-00+00');
        fitMedia(synth2, 2, measure, measure + 1);
}

for (var track = 1; track < 6; track++) {     //Add panning effects to all tracks (
        panAmount = ((track-1) * 50) - 100;
        setEffect(track, PAN, LEFT_RIGHT, panAmount);
}

fitMedia(guitar1, 4, 1, 9);    // Add a few extra tracks to measures 1 through 8
fitMedia(guitar2, 5, 1, 9);

//Finish
finish();
```

There are three for-loops. Let's walk through the first one line-by-line:

Line 26: Creates a variable named `measure`, and assigns it the value 1. This is our loop counter. Checks if `measure` 's value is less than 5. It is, so the program moves into the loop body.

Line 27: Adds a beat to the first half of the current `measure` (which is 1).

Line 28: Adds a different beat to `measure + 0.5` (the second half of the measure).

Line 29: Adds a clip to the current `measure`.

Line 30: At the end of the body, the iteration statement runs: `measure++`. `measure` now equals 2. Control jumps back to the top of the loop.

Line 26: Checks if `measure`'s value is less than 5.`measure` equals 2, so we loop again.

Lines 27-29: Same as before, but our measure numbers have increased. We add clips to the next measure.

Line 30: `measure++` runs again, so `measure` equals 3. Back to the top...

...a few iterations later...

Line 30: `measure++` runs again, so `measure` equals 5. Back to the top...

Line 26: `measure` is NOT less than 5, so the program exits the loop, moving onto line 18.

The second loop (lines 32-36) is similar, but starts the `measure` counter at 5, so the loop adds clips to measures 5-8.

The third loop (lines 38-41) is used to add a **panning** effect to all of the tracks. Panning changes the amount of audio that each speaker outputs (this also applies to headphones). As the track loop counter increases, so does the amount of pan applied.

Try commenting out that entire loop to hear the difference.

Loops tell the computer to do something repeatedly. Just like in music, you can specify how many times a repetition takes place. This simple concept turns out to be extremely versatile. We used loops to add clips to different measures, and to add effects to different tracks. There are many other possibilities; see if you can find your own use for them. Remember that the loop counter does much more than count: it changes the values of variables in your loop on each iteration.

Making Decisions 6

Musical Repetition vs. Contrast

With looping, we focused on adding repetition to our music. Most music, however, involves a balance between repetition and **contrast**. Contrast gives the listener something new and unfamiliar. This can be exciting to hear, and helps to drive the music forward. In this section, we will learn to make musical contrast with code, using conditional statements.

Conditional Statements: if...then

Conditional Statements make a decision: to run code or not to run code. They contain a block of code that will only be executed if a condition is met. They can be written in this form: "if [*this condition*] is true, then do [*this action*]". We use this kind of logic constantly in everyday life. For example, a recipe might tell you the following:

1. Put cookies in oven.
2. If cookies are brown around edges, then remove them from oven.
3. Let cookies cool before devouring.

Can you spot the conditional statement here? It is step 2. We have the condition "cookies are brown around edges" and an action to take if this condition is true: "remove them from oven". Both parts make up a conditional statement.

CONDITIONS AND BOOLEANS

A **condition** is an expression that can be either **true or false**. This true/false value is held by a data type called a **Boolean**. Think of how it compares to our other data

types... the 'number' data type has many possible values (1, 2, 3, 4...). Boolean has only two possible values: true or false.

We have already used conditions and Booleans in for-loops: the condition decided whether our code should run again or stop looping. For example, `measure < 5` is a condition, which could be true or false depending on measure's value. If measure is 6, for example, then `measure < 5` equals `true`.

Below, we replace our cookie condition with a more generic one.

```
// javascript code
//
// script_name: Conditionals
//
// author: The EarSketch Team
//
// description: Change the value of x to see how it affects the conditional stateme
//
//
//

//Setup
init();
setTempo(120);

//Conditionals

var x = false;      // Assigns the Boolean value "false" to a variable

if (x) {            // Checks if x is true or false. It is false, so the code block

        println("The condition is true!");

        fitMedia(YG_NEW_HIP_HOP_ARP_1, 1, 1, 9);
        fitMedia(RD_TRAP_MAIN808_BEAT_1, 2, 1, 9);
}

//Finish
finish();
```

Run it... and nothing happens. Our condition was false. Let's change the first statement to `var x = true;`. Now our code block runs! It prints something to the console, and adds some clips to the DAW.

Else

In the previous example, our program does nothing if the condition is false. This is fine in many cases, but sometimes we will want to add a *different* clip if our condition is *not* met. For this, we use an **if...else** statement:

```
// javascript code
//
// script_name: Conditionals
//
// author: The EarSketch Team
//
// description: Change the value of x to see how it affects the conditional statement
//
//
//

//Setup
init();
setTempo(120);

//Conditionals

var x = false;

if (x) {

        println("The condition is true!");

        fitMedia(YG_NEW_HIP_HOP_ARP_1, 1, 1, 9);
        fitMedia(RD_TRAP_MAIN808_BEAT_1, 2, 1, 9);

} else {        //The following block of code (lines 28-32) only runs if x is false.

        println("The condition is false!");

        fitMedia(YG_NEW_FUNK_ELECTRIC_PIANO_4, 1, 1, 9);
        fitMedia(YG_NEW_FUNK_DRUMS_4, 2, 1, 9);
}

//Finish
finish();
```

As before, try changing the value of `x` so our else statement runs. You will see that it only runs when `x` is false.

You might be thinking "Can't we just do this by putting a `fitMedia()` call after the conditional statement?" In this case, the clip would be added when our condition is *not* met (which is our intended effect). However, it would also be added if the condition *is* met (which is not our intended effect: we don't want both clips), so this is not the same as using `else`.

In short, `else` lets you use a condition to decide between two actions. Notice that an `if` statement (no `else`) only lets you decide between either doing or not doing a *single* action.

Conditional Statements In Loops

We won't usually manually change a Boolean value to get a conditional statement to run. As we will soon see, a Boolean can change as a result of our program running. For example, the loop counter in a for-loop changes with each iteration. We can use a Boolean to indicate whether or not our loop counter is greater than some number, and if it is, then execute a conditional statement.

Here is an example of using conditional statements inside of a loop:

```
// javascript code
//
// script_name: Looped Conditionals
//
// author: The EarSketch Team
//
// description: Using conditionals in loops to execute parts of the code only after
//
//
//

//Setup
init();
setTempo(100);


//Music
var drums = DUBSTEP_DRUMLOOP_MAIN_006;
var perc = DUBSTEP_PERCDRUM_002;
var bass = DUBSTEP_BASS_WOBBLE_011;


for (var measure = 1; measure < 9; measure++) { //This loop runs 8 times.
```

```
        fitMedia(drums, 3, measure, measure+1); //This is executed on every iteration. It add

        if (measure > 2) {
                fitMedia(perc, 2, measure, measure + 1); //This is only executed after 2 itera
        }

        if (measure > 4) {
                fitMedia(bass, 1, measure, measure + 1); //This is only executed after 4 itera
        }
}

//Finish
finish();
```

We have 3 statements inside of our for-loop. The `drums` are added to every measure in the loop. The `perc` is only added when `measure` is greater than 2 (measures 3-8). The `bass` comes in for measures 5-8.

This incremental build-up of instruments, called an **additive structure**, is typical of an **introduction** to a composition. A good example of this is from Kanye West's song *POWER*: in the introduction, listen as the different tracks are added and the sound becomes more complex and intense. **Kanye West's "POWER"**

Notice that the syntax for conditionals is very similar to for-loops. Each pair of open and closed curly braces { } makes a **block** of code: a group of statements that are meant to be run together (as in a loop or conditional).

Code blocks can also contain other code blocks! So when we place loops or conditionals inside of each other, we **nest** them. A single block has one pair of braces { `...` *one block...* }, while nested blocks have pairs of braces inside of other braces: { *...outer block...* { *...inner block...* } }.

Notice that we also increase the indentation for each level of nesting (i.e. for every new loop or conditional block inside of another one). Indentation doesn't have any effect on how your code runs in JavaScript, but is extremely important for user-readability. Be sure to use it.

Fills and Modulo

A drum **fill** is a short break in a drum beat where the drummer plays something different. Drummers often play fills at the end of a musical phrase, when the singer or other instruments have paused. It "fills" the empty space, and adds contrast to an otherwise repetitive rhythm.

We can add fills using the **modulo operator**: %. Note that this symbol does NOT mean percent in computer science. A modular division operation returns the remainder of the division of two integer numbers, rather than the quotient (which would be returned by the "/" sign). For example, 8 % 3 would return 2, whereas 8 % 4 would return 0. The modulo operator is useful for counting in cycles, just like a clock counts. Here, we use it to skip count:

```
// javascript code
//
// script_name: Modulo
//
// author: The EarSketch Team
//
// description: Skip-counting with the modulo operator
//
//
//

//Setup
init();
setTempo(208);

//Music

// This uses an "if statement" to place a synth clip on every even measure of track

for (var measure = 1; measure < 9; measure++) {
        if (measure % 2 == 0) {         // When measure is divisible by 2 (a.k.a. e
                fitMedia(EIGHT_BIT_ATARI_SYNTH_003, 1, measure, measure+1);
        }
}

//Finish
finish();
```

Here is a drum fill example:

```
// javascript code
//
// script_name: Drum Fills
//
// author: The EarSketch Team
//
// description: Using modulo to add drum fills
```

```
//
//
//

//Setup
init();
setTempo(100);

//Music
var mainBeat = "0+--0+-0+0++0+++";
var fill = "0+--0+-00+0-0000";

drums = ELECTRO_DRUM_MAIN_BEAT_005;


for (var measure = 1; measure < 17; measure++) {
        if (measure % 4 == 0) {             //Check if measure is divisible by 4. If it is (a
                makeBeat(drums, 1, measure, fill);
        } else {                            //... otherwise, add this beat!
                makeBeat(drums, 1, measure, mainBeat);
        }
}

//Finish
finish();
```

We added a fill to every fourth bar. An `else` statement runs if the conditional preceding it did not run; think of this statement as saying “otherwise, do this”.

Operators, Expressions, and Statements

We’ve used arithmetic operators like `+ - / *` and `%` to do math in our programs. Whenever you use an operator, you create an expression. The computer **evaluates** expressions to produce a single numerical value. For instance, the expression `(7+8)/3` is evaluated as 5.

There are several other operators in JavaScript. Here are the **comparison operators** that we’ve been using to specify conditions. Instead of evaluating to numerical values, they evaluate to the values `true` or `false`.

- `==`: is equal to
- `!=`: is not equal to
- `<`: is less than
- `>`: is greater than
- `<=`: is less than or equal to

- >=: is greater than or equal to

For more advanced material on Booleans, see **Teaching Computers to Listen**.

We've used the word **statement** a lot, but what exactly does it mean? It is when you tell the computer to take an action, like assign something to memory. Statements contain and combine expressions (and sometimes other statements): `volume = (28%9) / (11-(5*2))`. They are the complete sentences of a computer program.

Conclusions

We now know about two ways to change the control flow of a program: loops and conditionals. There are many ways of using these, either alone or in combination. If you find yourself writing many lines of similar code, try rewriting it as a loop. If you want something to change while you are looping, write a conditional statement. These tools are useful for creating musical repetition and variation within that repetition and much more conducive to creative experimentation than simply copying and pasting. When making changes, instead of having to change every copied code block, the code in the loops and conditionals can be changed in one place to accomplish the same result. You'll save a lot of time and effort using these new tools!

7 Musical Form

Using conditionals and loops helped us add small changes (like fills) to our music. Here we will learn about larger-scale changes in music, and how we can code them in a smart way. This will help you to create longer compositions with EarSketch.

Sections and Form

In music, a **section** generally refers to several measures of music (often 2, 4, 8, or 16 measures) that sound like a single musical unit. Compositions can be broken down into unique sections, each one expressing a different idea or feeling. You can think of sections of a composition as paragraphs in an essay.

By assigning capital letters to these unique sections (A, B, C, etc.) we can describe the structure of a composition as a series of sections. This high level structure is called **form**.

A-B-A Form

Let's begin with a simple form: A-B-A. This is a common form, and it tends to work well musically because the B section adds variety, while the second A section returns to something familiar. The code below creates an ABA form:

- **Section A**: measures 1-4.
- **Section B**: measures 5-7. Features contrasting sounds to Section A.
- **Section A (repeated)**: measures 7-10.

```
// javascript code
//
// script_name: A-B-A Form
//
// author: The EarSketch Team
//
```

```
// description: A song with A and B sections
//
//
//

//Setup
init();
setTempo(120);

//Music

// Create an A section

fitMedia(Y01_GUITAR_1, 1, 1, 5); // guitar
fitMedia(Y01_DRUMS_1, 2, 1, 5); // drums
for (var measure = 1; measure < 5; measure++) {
        makeBeat(Y01_BASS_1, 3, measure, "0---00--000-0000"); // bass part from rhy
}
for (var measure = 1; measure < 6; measure += 2) {
        fitMedia(Y01_WAH_GUITAR_1, 4, measure, measure+1); // second guitar
}
setEffect(4, DISTORTION, DISTO_GAIN, 10); // distortion on track 4

// fit a 2 measure B section between measures 5 and 7

fitMedia(Y01_OPEN_HI_HATS_1, 1, 5, 7); // drums breakout
fitMedia(Y01_WAH_GUITAR_1, 2, 5, 7); // lead
fitMedia(Y01_CRASH_1, 3, 5, 6); // cymbal crash

// Then back to section A at measure 7

fitMedia(Y01_GUITAR_1, 1, 7, 11); // guitar
fitMedia(Y01_DRUMS_1, 2, 7, 11); // drums
for (var measure = 7; measure < 11; measure++) {
        makeBeat(Y01_BASS_1, 3, measure, "0---00--000-0000"); // bass part from rhy
}
for (var measure = 7; measure < 11; measure += 2) {
        fitMedia(Y01_WAH_GUITAR_1, 4, measure, measure+1); // second Guitar
}
setEffect(4, DISTORTION, DISTO_GAIN, 10); // distortion on track 4

//Finish
finish();
```

This code creates an EarSketch project with an A and B section, and an ABA form. The B section is contrasting because it gets sparser and less energetic, by using fewer instruments and cutting the drums out. There are many other ways

to make contrasting sections, including adding effects and using different samples or rhythms.

Notice that the above code looks somewhat messy and confusing. Imagine if we extended this to an ABABCAA form! Also note the large section of repeated code (the last Section A). Repeated code is a sign that the program can be written more efficiently. To make this code more modular, we can create our own functions for section A and section B.

User-Defined Functions

`fitMedia()`, `setTempo()`, `init()`, and `range()` are all examples of function calls. A function call consists of the function's name (i.e. fitMedia) followed by parentheses which may contain parameters, and tells the function to run. The above listed functions are provided and defined by EarSketch, whereas some others come with JavaScript.

You can also create your own functions! **User-defined functions** allow you to write some code and execute it anywhere in a script without having to write it all over again.

Here we define a new function, and call it.

```
// javascript code
//
// script_name: User-Defined Functions
//
// author: The EarSketch Team
//
// description: Defining our own function that makes a section of music
//
//
//

//Setup

init();
setTempo(100);

//Music

// Here we define a function with one parameter (it takes one argument when called). Parameter

function myFunction(endMeasure) {
 fitMedia(ELECTRO_DRUM_MAIN_BEAT_003, 1, 1, endMeasure);
 fitMedia(ELECTRO_ANALOGUE_PHASERBASS_003, 2, 1, endMeasure);
}
```

```
// Now we call our function, passing it an argument of 17.
myFunction(17);

//Finish

finish();
```

You can name the function anything you like. We called ours `myFunction()`. It can also have as many parameters as you like (or none). A parameter is the variable that holds the argument you pass to a function. You can use this variable inside of the function's body.

Let's apply this knowledge to the ABA example. We make functions for section A and B, and then call them in the order they should appear in the music: ABA.

```
// javascript code
//
// script_name: Better A-B-A
//
// author: The EarSketch Team
//
// description: Making form with user-defined functions
//
//
//

//Setup

init();
setTempo(120);

//Music

// A section

function sectionA(leadGuitar, secondGuitar, drums, bass, startMeasure, endMeasure) {
        // create an A section
        fitMedia(leadGuitar, 1, startMeasure, endMeasure); // lead
        fitMedia(drums, 2, startMeasure, endMeasure); // drums
        // bass beat from startMeasure (inclusive) to endMeasure (exclusive)
        for (var measure = startMeasure; measure < endMeasure; measure++) {
                makeBeat(bass, 3, measure, "0---00--000-0000");
        }
        // second guitar every other measure from startMeasure (inclusive) to endMea
        for (var measure = startMeasure; measure < endMeasure; measure+=2) {
```

```
                fitMedia(secondGuitar, 4, measure, measure+1);
        }
        setEffect(4, DISTORTION, DISTO_GAIN, 10); // distortion on track 4
}

// B section

function sectionB(guitar, drums, cymbalCrash, startMeasure, endMeasure) {
        fitMedia(drums, 1, startMeasure, endMeasure);
        fitMedia(guitar, 2, startMeasure, endMeasure);
        fitMedia(cymbalCrash, 3, startMeasure, startMeasure+1);
}

// Setting up an ABA musical form through function calls

sectionA(Y01_GUITAR_1, Y01_WAH_GUITAR_1, Y01_DRUMS_1, Y01_BASS_1, 1, 5);
sectionB(Y01_WAH_GUITAR_1, Y01_OPEN_HI_HATS_1, Y01_CRASH_1, 5, 7);
sectionA(Y01_GUITAR_1, Y01_WAH_GUITAR_1, Y01_DRUMS_1, Y01_BASS_1, 7, 11);

//Finish

finish();
```

This is more concise, and now we can easily play with the form: by calling our section functions as many times as we like, in any order, and using any measure numbers. This allows us to make much more complex forms than with simple repetition because the parameters can be defined by the user with each call.

Return Statements

Some functions have a return statement. It is optional. Return does 2 things:

1. Returns (sends back) a value to the function call.
2. Exits the current function

Note that you do not have to have a return statement to exit a function. If there is no return statement, the function simply runs to the end of its definition.

Thus far we've mainly used functions to take an action on the DAW, like adding clips and effects.

Let's make a function that checks if its argument is even (divisible by 2), and returns a Boolean value as an answer.

```
// javascript code
//
// script_name: Simple Return
//
// author: The EarSketch Team
//
// description: Defining a function that returns a value to its caller
//
//
//

//Setup
init();

//Function Return Values

// A function that returns true if its argument is even, false if odd.

function isEven(myNumber) {
        var evenBool = (myNumber % 2 == 0);
        return evenBool;  // This is our return statement. It makes the function "p
}

var returnedValue = isEven(77);  // We assign the function's return value to the va

println(returnedValue);  // Check the console to see if your number was even.

//Finish
finish();
```

As the form of your music becomes more complex, you will find that it can be tedious and error prone to keep track of the starting and ending measure numbers for each section: if we want to change the length of one section, we have to change all of the others! A better way to do this is to have each section function return its *ending* measure number. Then, we can pass that to the next section function as its *starting* measure number. So if we want to adjust the length of a section, we can simply edit that section's function without worrying about the other sections.

```
// javascript code
//
// script_name: A-B-A Return
//
// author: The EarSketch Team
```

```
//
// description: Linking sections by returning an end measure
//
//
//

//Setup

init();
setTempo(120);

//Music

// A section

function sectionA(leadGuitar, secondGuitar, drums, bass, startMeasure) {
        // create an A section
        endMeasure = startMeasure + 4;
        fitMedia(leadGuitar, 1, startMeasure, endMeasure); // lead
        fitMedia(drums, 2, startMeasure, endMeasure); // drums
        // bass beat from startMeasure (inclusive) to endMeasure (exclusive)
        for (var measure = startMeasure; measure < endMeasure; measure++) {
                makeBeat(bass, 3, measure, "0---00--000-0000");
        }
        // second guitar every other measure from startMeasure (inclusive) to endMeasure+1 (ex
        for (var measure = startMeasure; measure < endMeasure; measure+=2) {
                fitMedia(secondGuitar, 4, measure, measure+1);
        }
        setEffect(4, DISTORTION, DISTO_GAIN, 10); // distortion on track 4
        return endMeasure;
}

// B section

function sectionB(guitar, drums, cymbalCrash, startMeasure) {
        endMeasure = startMeasure + 2;
        fitMedia(drums, 1, startMeasure, endMeasure);
        fitMedia(guitar, 2, startMeasure, endMeasure);
        fitMedia(cymbalCrash, 3, startMeasure, startMeasure+1);
        return endMeasure;
}

// set up an ABA musical form through function calls

var measure = sectionA(Y01_GUITAR_1, Y01_WAH_GUITAR_1, Y01_DRUMS_1, Y01_BASS_1, 1);
measure = sectionB(Y01_WAH_GUITAR_1, Y01_OPEN_HI_HATS_1, Y01_CRASH_1, measure);
measure = sectionA(Y01_GUITAR_1, Y01_WAH_GUITAR_1, Y01_DRUMS_1, Y01_BASS_1, measure);

//Finish
```

```
finish();
```

Abstraction

Just as we group musical ideas into sections, in programming we can create **abstractions**. An abstraction, in general, is a bundling of ideas to form a single concept. For example: a car, in one sense, is a collection of components: motor, radiator, steering wheel, etc. But considering the way we use all of these components together, it is useful to have a name for the whole collection: car. Car is our abstraction here. By themselves, none of the components can perform the function of driving, but the entire car can.

Functions are one kind of abstraction in computer science. They pack multiple statements into one tool, so the user doesn't have to worry about the complex inner workings, and so the user can easily refer to this group of statements.

It can be useful to think of programs as having their own kind of form. Abstractions can make this form more understandable to the programmer, which is helpful when writing and debugging large programs.

Conclusions

Form adds structure to the music and brings in more variety while remaining repetitive on a larger scale. This can give an ebb and flow to the music, making it more interesting for the listener. User defined functions and abstraction are great ways to implement form in code which make the program more efficient and easier to understand.

8 Making a Drum Set

Arrays

One very useful data structure that we'll frequently use is called an **array**, also called lists in some programming languages. Arrays are a way to store many items simultaneously in a single variable. These items, or **elements**, can be almost anything: numbers, strings, and so on. The two basic things you do with an array are **store items** and **retrieve items**.

If we wanted to make an array of all the clips we are using in our program, we could write something like this: `var myEnsemble = [Y02_DRUM_SAMPLES_1, Y01_BASS_1, Y02_GUITAR_1];`

`myEnsemble` is our variable that holds the array, and the elements in our array are inside of the square brackets.

So how do we retrieve an element from an array? Everything you put in an array gets an index number associated with it. The first (left-most) element in your array has an index number 0, the second has 1, the third has 2, and so on. *Notice that the count starts from 0, not 1!* You can access any array element with with its index number. The syntax for retrieving an array element looks like this: `myEnsemble[0]`. This retrieves the element at index 0 from the element (which is `Y02_DRUM_SAMPLES_1`).

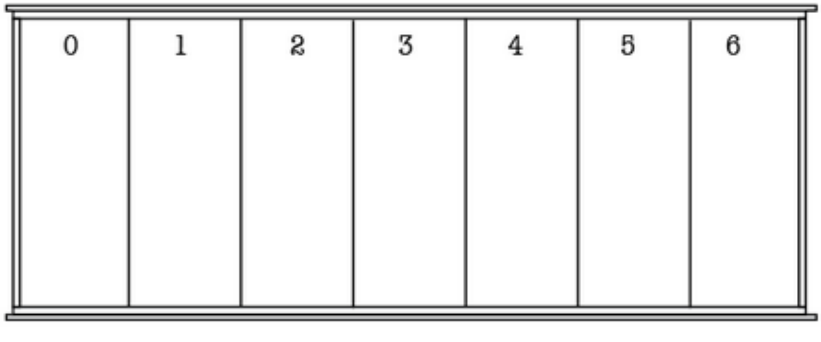
Here is a more typical example, where we use the retrieved array elements in several fitMedia calls.

```
// javascript code
//
// script_name: Arrays
//
// author: The EarSketch Team
//
// description: Using an array to hold several audio clips
//
//
```

```
//

//Setup
init();
setTempo(100);

//Music

var myEnsemble = [Y02_DRUM_SAMPLES_1, Y01_BASS_1, Y02_GUITAR_1];  // This is our ar

fitMedia(myEnsemble[0], 1, 1, 5); // We can access items (like clips) in an array w
fitMedia(myEnsemble[1], 2, 1, 5);
fitMedia(myEnsemble[2], 3, 1, 5);

//Finish
finish();
```

Instead of referring to each sample individually, we can make different groups of them and refer to them by their array name and index number. You can think of the array as a bookshelf that contains several related audio clips. Imagine that this bookshelf is complete with numbered placeholders in which to put books. The numbered placeholders represent index values in the array. An empty array might look something like this:

FIGURE 8-1



Iterating through Arrays

A common task with arrays is reading through them, going from the first to last index. For example, if you think of an array as an album (an array of songs), then you might want to play each song in order (beginning to end). We use loops to read through arrays in this way. This loop prints each string in an array, in order.

```
// javascript code
//
// script_name: Iterating Through an Array
//
// author: The EarSketch Team
//
// description: Using a loop to retrieve every item in an array, and print each one.
//
//
//

//Setup
init();

//Array Iteration

var myArray = ["Get", "thee", "to", "the", "console!"];

for (var i = 0; i < 5; i++) {  // Since there are 5 items in our list, we know it uses indice
        println(myArray[i]);  // We use i as the index number
}

//Finish
finish();
```

Above, we counted the number of array elements to determine how many times to loop. While easy enough in this example, you will generally want the computer to find the length of the array for you. It won't make any mistakes, and it can count very long lists quickly. You can find the length of an array using this syntax: `myArray.length`. Every array has several properties that describe it. Length is just one of these properties, and you can access it by writing `.length` after your array name.

```
// javascript code
//
```

```
// script_name: Length of a List
//
// author: The EarSketch Team
//
// description: Using array.length to get the number of elements in an array and it
//
//
//

//Setup
init();

//Using the Array's Length

var myArray = ["Get", "thee", "to", "the", "console!"];

for(var i = 0; i < myArray.length; i++) { // The stopping condition is determined b
        println(myArray[i]);
}

//Finish
finish();
```

We can use iteration through an array to build an introduction to a song. One effective way to build an introduction is through an additive structure. An additive structure is one that begins with one track (or more) and, as the intro progresses, adds more and more tracks at regular time intervals. A good example of this is from Kanye West's song POWER: in the introduction, listen as the different tracks are added and the sound becomes more complex and intense.

Kanye West's "Power"

Below is an example of how an additive introduction can be implemented in EarSketch using an array and for loop.

```
// javascript code
//
// script_name: An Introduction
//
// author: The EarSketch Team
//
// description: Making an additive introduction to a song using arrays
//
//
//

//Setup
```

```
init();
setTempo(108);

//Music

var drum1  = HIPHOP_DUSTYGROOVE_003;
var drum2  = TECHNO_LOOP_PART_006;
var synth  = TECHNO_CLUB5THPAD_001;
var whoosh = HOUSE_SFX_WHOOSH_001;

var introArray = [drum1, drum2, synth, whoosh];


// intro generator

for (var track = 1; track < 5; track++) {
        // make a new array index that is 0-based
        var mediaIndex = track - 1;
        // make a measure variable that skips every other measure
        // this one will have values 1, 3, 5, 7, 9
        var measure = (track * 2) - 1;
        fitMedia(introArray[mediaIndex], track, measure, 9);
}

//Finish
finish();
```

Using Arrays with makeBeat

Thus far, we've used `makeBeat()` with strings made of '0' '+' and '-'. A 0 plays the sample in `makeBeat()`'s argument.

makeBeat has another feature: if we want to use multiple samples, makeBeat can use the numbers 0 to 9 in its string to refer to array indices, where our array stores a set of samples. You can think of each array element as a drum in our drum kit; each drum is labelled with a unique index number, and makeBeat plays it whenever it reads that number in its beat string.

```
// javascript code
//
// script_name: Making a Drumset
//
// author: The EarSketch Team
```

```
//
// description: Using arrays with makeBeat
//
//
//

//Setup
init();
setTempo(100);

//Music
var drums = [ELECTRO_DRUM_MAIN_BEAT_001,  //This is our "drumset": an array of drum
                ELECTRO_DRUM_MAIN_BEAT_002,
                ELECTRO_DRUM_MAIN_BEAT_003,
                ELECTRO_DRUM_MAIN_BEAT_004];

var drumPattern = '0+0+11112+2+3+++';   // Each number is actually an index into an

makeBeat(drums, 1, 1, drumPattern);

//Finish
finish();
```

Array Operations

There are many ways of manipulating arrays in your code. So far, we have looked at creating them manually, and accessing their elements in different ways. What if we want to change their contents after they have been created? We can use array operations to add, remove, and search for elements.

Musically, we have been using arrays to store clips that make up a “drum set”. What if we want to replace some of these clips later? To add an element to the end of an array, we use the `push()` function. So, if we write `myArray.push(newElement)`, then our `newElement` gets placed in `myArray`, directly after the current last element (if there is one).

How about deleting or inserting an element at a specific position in an array? We can use the array `splice()` function for both of these. If we write `myArray.splice(startIndex, deleteCount, elementToInsert)`, then splice will go to the `startIndex` of `myArray`. At that point, it will delete however many elements are specified by `deleteCount` (after `startIndex`). Finally, it inserts our `elementToInsert` at `startIndex`.

Suppose you want to create a “drum kit” array by picking clips at random. We can do this by using the EarSketch function `selectRandomFile()` ! It takes

a folder name (from the sound browser) as an argument, and returns a single clip name.

Below, we use selectRandomFile() , append() , and splice() to make a drum kit that has a one of it's clips replaced every measure. Be sure to run the code a few times and listen to the differences... you will hear different random clips selected each time!

```
//              javascript code
//
//              script_name: Random Clips
//
//              author: The EarSketch Team
//
//              description: Using array operations to randomly replace clips in an array
//
//
//

//Setup
init();
setTempo(120);

//Music

var folder1 = MAKEBEAT;
//var folder2 = DUBSTEP_140_BPM__DUBDRUM;

var drums = [];

var drumString = "0+1+22230+302121";

for(var i = 0; i < 16; i++) {
        var newClip = selectRandomFile(folder1);               // Picks a file at random fro
        if (i < 4) {
                drums.push(newClip);    // Fill up our array first
        }
        else {
                var measure = i-3;

                makeBeat(drums, 1, measure, drumString);
                drums.splice(i%4, 1, newClip);  // After our array is full, we replace one cl
        }
}

//Finish
finish();
```

If you listen carefully, you can here that the basic rhythm stays the same, while the clips used in the rhythm are gradually changed at random. Try using a different folder to select random clips from! In the next lesson, we will learn more about how to incorporate random elements into your music.

9 Randomness and Strings

When musicians use computers to make music, they often do so because they want complete and precise control over the music they create. At the same time, many musicians also use technology to give up control, introducing elements of randomness into their music. In a sense, they let the computer improvise, within the guidelines the composer gives it.

There is a long history of randomness in music. In 1787, Mozart created a musical "dice game." He composed many different one-measure musical fragments, and then rolled two six-sided dice to decide which fragment to choose for each measure in the piece, a short minuet. Mozart designed the musical fragments so that no matter how they were rearranged based on the dice rolls, the music would sound good. Many other composers have used techniques similar to this, especially as technology made it more feasible.

Random numbers are used often in computing: from shuffling a playlist of songs, to encrypting personal data. In EarSketch, we can use random numbers to introduce some surprise and some novelty into our music, to give a more improvisational feel to our music. You'll use random numbers in combination with **string operations**: tools for programmatically rearranging strings.

Random Numbers

You probably have an intuitive sense that randomness means unpredictability. That is exactly what it means for a series of numbers to be random: you cannot predict an upcoming random number from any that have been previously generated.

We can generate a random number in JavaScript using the `random()` function. To call this function, we have to retrieve it from something called Math. Math is a collection of math-related utilities, such as our random number generator. We call the random() function in Math by writing `Math.random()`.

`Math.random()` generates floating point numbers between 0 (inclusive) and 1 (exclusive). We will need to scale this by multiplying the result by our desired

range of numbers. When generating random numbers, we always need to choose a range for these numbers to be generated in. For example, if we have 6 items in an array, and we want to randomly choose them by their indices, then we need random numbers from 0 to 5. Since `Math.random()` generates numbers between 0 and 1 (exclusive), `Math.random() * 6` will generate random numbers between 0 and 6 (exclusive). Our random number call would look like this: `Math.random() * 6`.

There is one final snag: array indices must be whole numbers! If Math.random() returned 0.27, and we multiplied by our range of 6, we would get an index of 1.62, which doesn't exist. To solve this, we use `Math.floor()`, which rounds it's argument down to the nearest whole number: `Math.floor(Math.random() * range)`. You can use this expression as a formula for making random numbers.

Let's use random numbers to pick random clips from an array.

```
// javascript code
//
// script_name: Random Clip
//
// author: The EarSketch Team
//
// description: Randomly selecting clips from a list
//
//
//

//Setup
init();
setTempo(100);

//Music
var sampleBank = [YG_TRAP_SYNTH_BELL_1,
                          YG_TRAP_STRINGS_1,
                          YG_TRAP_SHORT_SYNTH_1,
                          YG_TRAP_HIT_1,
                          YG_TRAP_SYNTH_LEAD_6,
                          YG_TRAP_BELLS_1];


for(var i = 1; i < 9; i++) {
 // Generate a random index number
        var index = Math.floor(Math.random() * 6); // Generates a random index numbe

        // Use the random index to pick a sample
        fitMedia(sampleBank[index], 1, i, i+1);
}
```

```
fitMedia(YG_TRAP_KICK_4, 2, 1, 9);
fitMedia(YG_TRAP_BASS_1, 3, 1, 9);
fitMedia(YG_TRAP_SNARE_5, 4, 1, 9);

//Finish
finish();
```

String Operations: Concatenation

Concatenation. This simply means linking strings together in a series. If we concatenate the string “hello” with the string “world”, we get a new string: “helloworld”. In this section, we will concatenate beat strings to make longer rhythmic patterns.

We can concatenate strings using a plus (+) sign.

```
// javascript code
//
// script_name: Concatenation
//
// author: The EarSketch Team
//
// description: Combining two strings into one string
//
//
//

//Setup
init();

//Concatenation

var stringA = "Great";
var stringB = "concatenations";

var newString = stringA + " " + stringB; // In this context, the plus symbol means "concatena

println(newString);

//Finish
finish();
```

We concatenated three strings, including a space " ".

Let's make some music with this. We can make a drum beat from randomly ordered strings. Below, we start by defining several short rhythms. Then, in a loop, we choose a random beat from our list and concatenate it with `finalBeat`. On each iteration, we concatenate another string, so `finalBeat` grows as the loop continues.

```
// javascript code
//
// script_name: Random Concatenation
//
// author: The EarSketch Team
//
// description: Randomly combining beat strings
//
//
//

//Setup

init();
setTempo(100);


//Music
var beatList = ["0++01-0+",

                        "-00+1---",

                        "0+++1-0+",

                        "0+01++10",

                        "0+++1-11"];

var endBeat = "0+001-2+";

var drums = [OS_KICK03, OS_SNARE01, OS_OPENHAT04];

var finalBeat = "";  //We initialize an empty string, so that we have something to

for(var i = 0; i < 6; i++) {
        //println(i);
        var beatIndex = Math.floor(Math.random() * 5);
        //println(beatIndex);
        finalBeat += beatList[beatIndex];   // This is the same as writing finalBeat
```

```
}

finalBeat = beatList[0] + finalBeat + endBeat;   //Combining the beat strings we have created

makeBeat(drums, 1, 1, finalBeat);
fitMedia(YG_WEST_COAST_HIP_HOP_PIANO_1, 2, 1, 5);

//Finish
finish();
```

Notice that we only iterate 6 times, then redefine `finalBeat` so it has `beat[0]` concatenated at the beginning and `endBeat` at the end. This balances out the randomness with some regularity, and also gives the listener a cue that the phrase is ending: the hi hat plays.

Choosing the right arguments for your random number generator and indices for your loop can sometimes be tricky. If something doesn't seem quite right, add print statements to check the range of random numbers you are getting. We have added two print statements in the example above; try uncommenting each one separately.

String Operations: Substrings

Suppose we want to go in the other direction: instead of assembling a beat string from pieces, let's chop one up and rearrange it. This kind of approach is very popular in electronic music, especially remixing.

We can "chop up" a string using JavaScript's `substring()` function. It makes a new shorter string from part of another string. `substring()` takes two arguments: a starting index (inclusive), and an ending index (exclusive). Like this: `a = substring(start, end)`. The characters between start and end are copied to the new string `a`.

```
// javascript code
//
// script_name: Substrings
//
// author: The EarSketch Team
//
// description: Getting a part of a string using .substring()
//
//
```

```
//

//Setup
init();

//Substrings

var a = "Pulling a rabbit out of a string";
var b = a.substring(10, 16); // Makes a new string from the 10th-15th characters of

println(b);

//Finish
finish();
```

Remixing a rhythm

Let's remix a rhythm by using substrings. We can start with a well-known rhythm called the "Amen break". This is widely sampled in many styles of electronic music. Here we construct a beat string of it:

```
// javascript code
//
// script_name: Amen Break
//
// author: The EarSketch Team
//
// description: Building a classic drum solo with concatenation
//
//
//

//Setup
init();
setTempo(170);

//Music
var drums = [OS_KICK05, OS_SNARE06, Y24_HI_HATS_1, Y58_HI_HATS_1, OS_OPENHAT01];

var a = "0+0-1+-1+1001+-1";
var b = "0+0-1+-1-10---1+";
var c = "-1001+-1+10---1+";
```

```
var amenBreak = a + a + b + c; // Concatenating all of the fragments to make the final patter

makeBeat(drums, 1, 1, amenBreak);

//Finish
finish();
```

We could have just written it out as one long string, but instead we concatenate four strings to build it. This makes it easier to see the structure, since each starting string is a measure long.

Our rhythm is missing some cymbals. The cymbals should play at the same time as the main rhythm, so we need to make a separate string and `makeBeat` call for them:

```
// javascript code
//
// script_name: Amen Cymbals
//
// author: The EarSketch Team
//
// description: Adding the cymbals to our amen break
//
//
//

//Setup
init();
setTempo(170);

//Music
var drums = [OS_KICK05, OS_SNARE06, Y24_HI_HATS_1, Y58_HI_HATS_1, OS_OPENHAT01];

var a = "0+0-1+-1+1001+-1";
var b = "0+0-1+-1-10---1+";
var c = "-1001+-1+10---1+";
var cym1 = "2+2+2+2+2+2+2+2+";
var cym2 = "2+2+2+2+2+3+2+2+";
var cym3 = "2+2+2+2+2+4+2+2+";

var amenBreak = a + a + b + c;
var amenCymbals = cym1 + cym1 + cym2 + cym3;

makeBeat(drums, 1, 1, amenBreak);
makeBeat(drums, 2, 1, amenCymbals);
```

```
//Finish
finish();
```

Now for the remixing part. Let's try inserting a random drum pattern somewhere into the middle of our `amenBreak` string. First we make a random beat string, 8 characters long, by generating random numbers in a loop; this is our random drum pattern. These random numbers are concatenated with the same string, 8 times, so the string grows on each iteration. Notice that += works with concatenation, not just addition!

```
var insertSection = "";

for(var i = 0; i < 8; i++) {
        insertSection += Math.floor(Math.random() * 5);
}
```

Now, we want to insert `insertSection` into the middle of our main `amenBreak` string. We want our final string to be the same length as the original string. We will need to make substrings to do this:

```
var insertLocation = 16;
var numBeats = 8;

var frontSection = amenBreak.substring(0, insertLocation);
var endSection = amenBreak.substring(insertLocation + numBeats, amenBreak.length-1)

var newBeat = frontSection + insertSection + endSection;
```

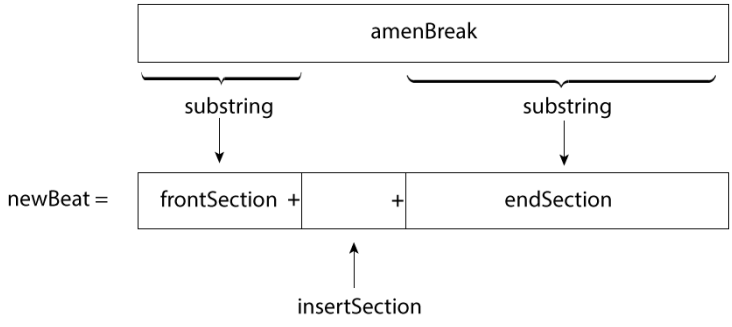


FIGURE 9-1

Inserting a section in the middle of amenBreak, using substrings

Putting everything together, we have this:

```
// javascript code
//
// script_name: Amen Remix
//
// author: The EarSketch Team
//
// description: Replacing part of the amen break string with a random beat string
//
//
//

//Setup
init();
setTempo(170);

//Music
var drums = [OS_KICK05, OS_SNARE06, Y24_HI_HATS_1, Y58_HI_HATS_1, OS_OPENHAT01];

var a = "0+0-1+-1+1001+-1";
var b = "0+0-1+-1-10---1+";
var c = "-1001+-1+10---1+";
var cym1 = "2+2+2+2+2+2+2+2+"
var cym2 = "2+2+2+2+2+3+2+2+"
var cym3 = "2+2+2+2+2+4+2+2+"
```

```
var amenBreak = a + a + b + c;
var amenCymbals = cym1 + cym1 + cym2 + cym3;


var insertSection = "";

for(var i = 0; i < 8; i++) {
        insertSection += Math.floor(Math.random() * 5);  // We build our random str
}

var insertLocation = 16;  // Our random string gets inserted to the amenBreak at th
var numBeats = 8;
var frontSection = amenBreak.substring(0, insertLocation);
var endSection = amenBreak.substring(insertLocation + numBeats, amenBreak.length-1)
var newBeat = frontSection + insertSection + endSection;


makeBeat(drums, 1, 1, newBeat);
makeBeat(drums, 2, 1, amenCymbals);

//Finish
finish();
```

For a more advanced version of this script, see **Abstracting the Remix**.

10 More Effects

Collaboration has always played a part in music-making. In recorded music, you typically have many different people working on one project: composer, songwriter, producer, sound engineer, and so on. There are many tasks involved in creating a finished product, and one of these is the mixing stage. This means applying effects to make the final "mix" of tracks sound balanced and interesting. We will use some additional features of `setEffect()` to do this.

Beyond mixing and mastering, there are many other creative uses for `setEffect()` . You can completely transform a piece of music by applying high levels of effects to your tracks. You can even make time-varying effects, so that the effect amount changes with the rhythm of your composition!

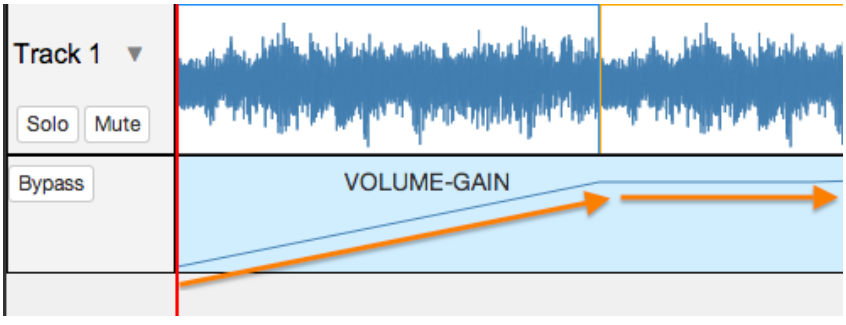
Envelopes

The effects we have added with `setEffect()` have been applied to entire tracks, with the effect parameters remaining constant throughout the track. What if we only wanted to add effects to part of a track or change the effect values over time? We can use envelopes to do this.

Envelopes define how an effect's parameter changes over time:

FIGURE 10-1

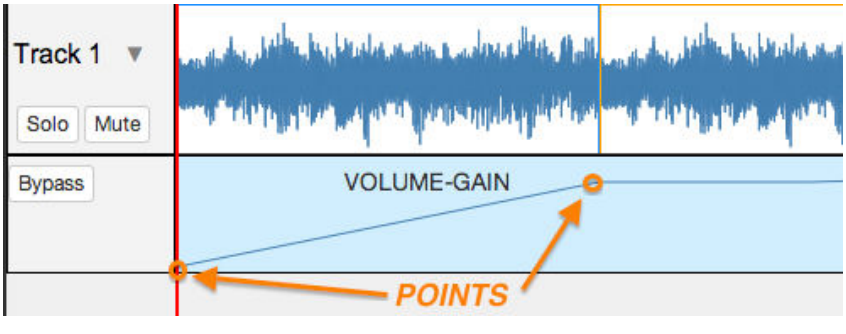
Changing volume over time with envelopes.



We describe an envelope using a series of value-time pairs. Each pair contains a value for an effect parameter, and a point in time to set it to that value. For example: (-60, 1, -5, 5) means to place a point at measure 1 at value -60, and also place a point at measure 5 with value -5. The envelope creates lines between these points, like a game of connect the dots.

FIGURE 10-2

Envelopes are defined by points.



The smooth transition between points is called a **ramp**.
Envelopes can be used with any effect parameters.

Envelopes with setEffect

Some functions can accept different numbers of arguments. `setEffect()` is one of these functions. So far we have used it with 4 arguments: trackNumber, effect, parameter, and effectValue.

Take a look at `setEffect()` in the EarSketch API. You'll see that it can take up to 7 arguments!

1. trackNumber
2. effect
3. parameter=None
4. effectStartValue=None
5. effectStartLocation=1
6. effectEndValue=None
7. effectEndLocation=None

The parameters followed by equal signs are **optional parameters**. If you don't pass an argument for one of them, they are set to the default value (specified after the equals sign in the API).

For the simplest use, we would just call `setEffect()` with 2 arguments: trackNumber and effect. In this case, the remaining arguments would be set to their respective default values. This applies the default effect settings to the entire track.

To use envelopes, we will need to use all of `setEffect()`'s parameters. Remember the {value, measure number} pairs above? You can think of each `setEffect()` call as taking 2 of these pairs: {effectStartValue, effectStartLocation}; {effectEndValue, effectEndLocation}. Take a look at this example:

```
// javascript code
//
// script_name: Envelopes
//
// author: The EarSketch Team
//
// description: Making envelopes with 7 parameter setEffect
//
//
//

//Setup
init();
setTempo(120);

//Music
fitMedia(ELECTRO_ANALOGUE_LEAD_012, 1, 1, 9);
setEffect(1, VOLUME, GAIN, -60, 1, 0, 3);  // Makes an effect ramp between measures 1 and 3, r

//Finish
finish();
```

This is a called a **fade**. It makes the sound “fade in" by gradually increasing the volume between measures 1 and 3. Fades are a great way to start or end a composition. You can even use them to transition, by fading one track out (decreasing volume) while fading another in.

You can make more complex envelopes by making several consecutive `setEffect()` calls. Below, we define the time points as variables to make things easier to see. Many of the end points for one `setEffect()` call are the same as the starting point for the next. Note that you can also stack multiple effects on a single track. Or if you want to tweak a single effect even more, you can set multiple parameters for that single effect! Use a separate setEffect call to change each parameter on a single effect.

```
// javascript code
//
// script_name: Complex Envelopes
//
// author: The EarSketch Team
//
// description: Using multiple setEffect calls on a track to make changes in the ef
//
//
//

//Setup
init();
setTempo(120);

//Music
fitMedia(ELECTRO_ANALOGUE_LEAD_012, 1, 1, 9);

// Envelope time points
var pointA = 1;
var pointB = 4;
var pointC = 6.5;
var pointD = 7;
var pointE = 8.5;
var pointF = 9;

setEffect(1, VOLUME, GAIN, -60, pointA, 0, pointB);   // fade in
setEffect(1, VOLUME, GAIN, 0, pointB, 12, pointC);  // crescendo
setEffect(1, VOLUME, GAIN, 12, pointD, 0, pointE);  // begin fade out
setEffect(1, VOLUME, GAIN, 0, pointE, -60, pointF); // end of fade out

setEffect(1, FILTER, FILTER_FREQ, 20, pointA, 10000, pointF); // another effect sta

//Finish
```

```
finish();
```

Automating Effects

You can get creative with effects by automating them: using algorithms to define envelopes. Suppose we wanted to make a volume ramp every measure. We can do this with a for loop:

```
// javascript code
//
// script_name: Rhythmic Ramps
//
// author: The EarSketch Team
//
// description: Automating effects with a for loop
//
//
//

//Setup
init();
setTempo(120);

//Music
for(var measure = 1; measure < 17; measure++) {
        setEffect(1, VOLUME, GAIN, -60, measure, 0, measure+1);
}

fitMedia(Y33_CHOIR_1, 1, 1, 17);
fitMedia(RD_ELECTRO_DRUM_PART_10, 2, 1, 17);

//Finish
finish();
```

Our `setEffect()` function is called every measure with a new pair of measure numbers as arguments.

Let's make our automation a bit more complex. What if we wanted to have several ramps per measure? We'll need to do some math.

```
// javascript code
//
// script_name: Fast Effects
//
// author: The EarSketch Team
//
// description: Making multiple volume ramps per measure
//
//
//

//Setup
init();
setTempo(120);

//Music
var measures = 17;
var subdivision = 12;

for(var measure = 1; measure < measures * subdivision; measure++) { // Notice that
        var startLocation = measure/subdivision;
        var endLocation = measure/subdivision + 1/subdivision;

        setEffect(1, VOLUME, GAIN, 0, startLocation, -60, endLocation);
}

fitMedia(Y33_CHOIR_1, 1, 1, 17);
fitMedia(RD_ELECTRO_DRUM_PART_10, 2, 1, 17);

//Finish
finish();
```

We define `subdivisions` (the number of ramps we want per measure), and `measures` (the total number of measures) at the top.

Let's walk through this in detail:

We want 12 ramps for every measure, and there are 17 measures, so we need to call `setEffect()` a whopping 12 * 17 = 204 times. Rather than write these numbers in, we can see that the total number of ramps is subdivisions * measures, so we use that as our stopping condition for the loop.

The `setEffect()` call is a bit tricky. Our starting location is defined as i/subdivisions. To get the right number of loops, we had to multiply measures by subdivisions. Now we need to scale things back down to actual measure numbers, so we divide i by subdivisions.

To get the ending point of the `setEffect()` call, we take our starting point and add the desired length of our `setEffect()` call: 1/subdivision. Why is this the length? Picture having 4 subdivisions per measure: each subdivision is 1/4 of a measure long. Adding the length to the starting point gives you the ending point.

Try changing the number of subdivisions, or flipping the slope of the envelope so it starts low and increases.

Teaching Computers to Listen 11

Up until now, you have been combining audio clips to create music and adding effects to these clips to make it more interesting. What if you could get the computer to analyze parts of your composition, and even add effects based on how different parts sound? This kind of analysis is called **Music Information Retrieval** (MIR), and is widely used in the music technology industry. These techniques allow you to do things such as automatically categorize music (as part of a music recommendation system), identify a song by humming it into your phone, and so on.

Here, we will learn to do some simple MIR using EarSketch's `analyze()` function. We will use the analysis to control various parts of our composition.

Music Information Retrieval

Audio, like all data, is stored in a computer as a series of 0s and 1s. This binary representation is at a very *low level* of abstraction. On the other end of the spectrum: when we hear this audio being played, we recognize *high level* features like harmony, rhythm, instrumentation, and so on. These features are very abstract: they give us a simple name for a mental process that is actually extremely complex. In a sense, abstractions are useful because they hide (or **encapsulate**) the details, and allow us to focus on the big picture.

Computer code can analyze that audio data to determine certain unique characteristics of the sound (how loud, how high or low, etc.). Any sound can be analyzed based on these various features to allow the computer to understand something about the sound. This process is called **music information retrieval**, or MIR. MIR can be used to teach computers to "listen" to and distinguish sounds. Humans do this naturally and subconsciously, such as when we distinguish between two people's voices.

A challenge in computing is to try to find high level features in low level data. In other words, to teach the computer to find relevant information in a stream of bits. There are many tasks that involve this: for example, making sense of

how a human's DNA (low-level) might be expressed to cause a genetic disease (high-level). Or perhaps, how local temperature and barometric measurements (low-level) in many different places can be combined to form a prediction of the weather (high-level).

EarSketch has a function that performs this kind of analysis for several higher-level audio features: `analyze()`

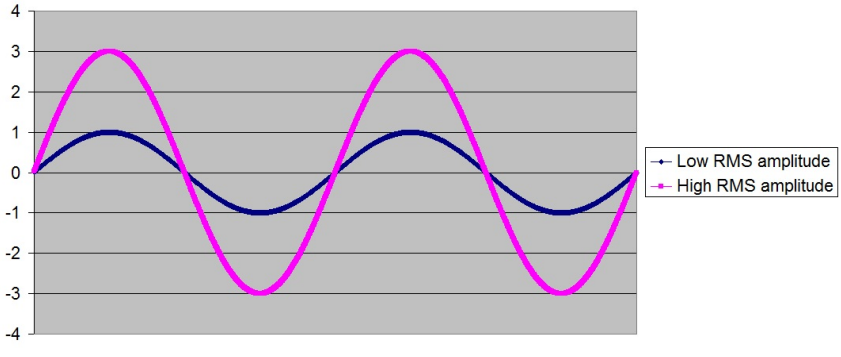
Analysis Features

Every sound can be analyzed in terms of **features**, which are high-level descriptions of our binary data. Here we will focus on two of the most essential: **RMS Amplitude** and **Spectral Centroid**.

RMS Amplitude is defined as the root mean square (a kind of average) of the amplitude of a sound and is related to the volume of the sound – a sound with a high RMS amplitude will be loud, and a sound with a low RMS amplitude will be soft. This relationship is easy to visualize with graphs because the amplitude of a sound is defined visually as the vertical distance between the crest (top) and trough (bottom) of a sound wave. In the figure below, the blue wave has a lower RMS amplitude than pink wave.

FIGURE 11-1

RMS Amplitude



Let's start by using `analyze()` with RMS Amplitude. `analyze()` looks at a clip and analyzes it using a method of our choice. It returns a value that corresponds to how high or low the level of our feature is in the audio we analyzed.

If we want to analyze a clip with RMS Amplitude, we write: `analyze(clip, RMS_AMPLITUDE)` . In the example below, we do this and then print the value that analyze returns:

```
//              javascript code
//
//              script_name: Analyze
//
//              author: The EarSketch Team
//
//              description: Using analyze to get the RMS_Amplitude of a clip, then printing
//
//
//

//Setup
init();
setTempo(120);

//Analysis

var clip = YG_TECHNO_ELECTRIC_PIANO_2;

var analysisValue = analyze(clip, RMS_AMPLITUDE);

println(analysisValue);


//Finish
finish();
```

Now, consider the following code, which analyzes the RMS amplitude of an audio clip and adds it to a track if this amplitude is greater than a certain threshold:

```
// javascript code
//
// script_name: RMS Threshold
//
// author: The EarSketch Team
//
// description: This script adds media to the DAW only if it has an RMS amplitude value above
//
//
//
```

```
//Setup
init();
setTempo(120);

//Music

var clipToAnalyze = HIPHOP_TRAPHOP_BEAT_001; //Choose the audio clip to use
var analysisMethod = RMS_AMPLITUDE; //Define the feature to analyze
var rmsThreshold = 0.5; //Define the minimum RMS amplitude necessary for the clip t

//Define the starting and ending measures
var start = 1;
var end = 3;

//Analyze the RMS amplitude of the entire audio clip, and print the value to the co
var rmsAmplitude = analyze(clipToAnalyze, analysisMethod);
println(rmsAmplitude);

//Set up conditional statement to add clip to track 1 only if its RMS amplitude is
if(rmsAmplitude >= rmsThreshold) {
        //Insert clip on track 1
        fitMedia(clipToAnalyze, 1, start, end);
}

//Finish
finish();
```

Line 13 uses the `analyze()` function, which takes as its parameters an audio clip and an analysis feature, to obtain the RMS amplitude (average loudness, roughly) of an audio clip:

```
var rmsAmplitude = analyze(clipToAnalyze, analysisMethod);
```

The `if` statement at lines 17-20 tests to see if the RMS amplitude of the audio clip is larger than the threshold value (line 9); if it is, the audio clip is added to track 1 from measures 1-3. Try changing the value of rmsThreshold so that the clip is added to the track.

To make MIR more useful, the computer must be able to execute conditional statements to make decisions based on the values of various features. Try running the following code, which creates a noise gate that mutes a track whenever another's volume falls below a certain threshold:

```
// javascript code
//
// script_name: Auto-Volume
//
// author: The EarSketch Team
//
// description: We compare the RMS amplitude of two samples at different points in time, and
//
//
//

//Setup
init();
setTempo(120);

//Music
var sound1 = ELECTRO_DRUM_MAIN_BEAT_001;
var sound2 = ELECTRO_DRUM_MAIN_BEAT_002;
var analysisMethod = RMS_AMPLITUDE;
var hop = 0.0625;  // analyze in 1/16th note chunks
var start = 1;
var end = 3;
var numChunks = 32;

fitMedia(sound1, 1, start, end);
fitMedia(sound2, 2, start, end);

for(var i = 0; i < numChunks; i++) {

    var position = 1 + i * hop;
    var feature1 = analyzeTrackForTime(1, analysisMethod, position, position + hop);
    var feature2 = analyzeTrackForTime(2, analysisMethod, position, position + hop);

        if (feature1 > feature2) {
        setEffect(1, VOLUME, GAIN, 0, position, 0, position + hop);
        setEffect(2, VOLUME, GAIN, -60, position, -60, position + hop);
        } else {
        setEffect(1, VOLUME, GAIN, -60, position, -60, position + hop);
        setEffect(2, VOLUME, GAIN, 0, position, 0, position + hop);
        }
}

//Finish
finish();
```

A for loop at line 15 steps through each 1/16th note section of an audio clip. Notice that the hop variable is defined as 1/16 = 0.0625, and line 17 updates the location variable by adding 1/16 to the current measure value. Then, line 21 uses a conditional to check if the volume of track 1 at each location is greater than the volume of track 2 at each location. The track with the greater volume has its volume at the current location set to 0dB using setEffect, and the track with the lesser volume is set to -60dB.

```
if (feature1 > feature2) {
        setEffect(1, VOLUME, GAIN, 0, position, 0, position + hop);
        setEffect(2, VOLUME, GAIN, -60, position, -60, position + hop);
        } else {
        setEffect(1, VOLUME, GAIN, -60, position, -60, position + hop);
        setEffect(2, VOLUME, GAIN, 0, position, 0, position + hop);
        }
```

This code uses a method called `analyzeTrackForTime()` to analyze the audio clip. The function has four parameters: the track number, the feature to be analyzed, and the starting and ending locations for the analysis. The function returns a number between 0 and 1, representing the average value of the specified feature between the starting and ending point.

Spectral centroid is defined as the average frequency of a sound and relates to the "brightness" of the sound – a sound with a high spectral centroid (like a cymbal) will be brighter and usually sound higher in pitch, whereas a sound with a low spectral centroid will be less bright and usually sound lower in pitch.

A graph of the average frequency of a drum sound might look like this:

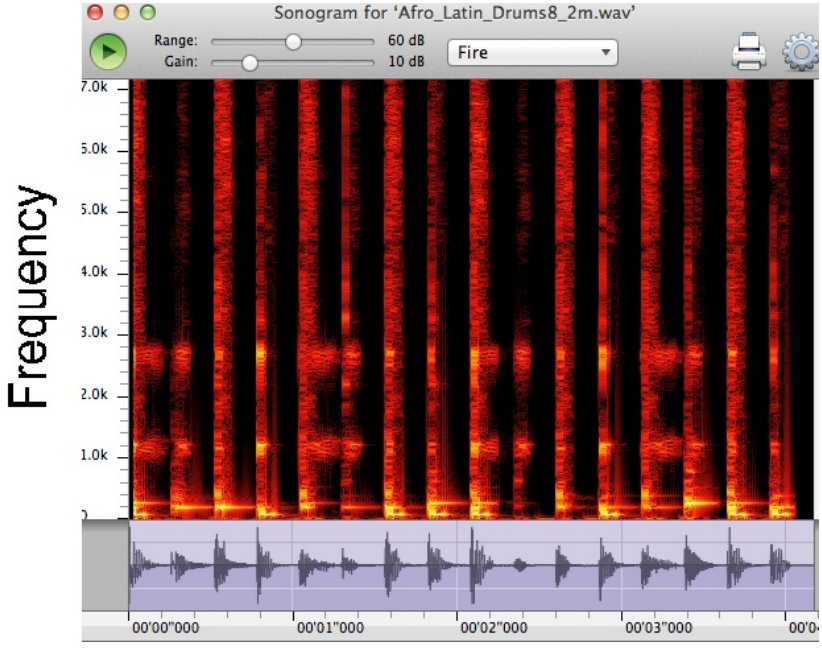


FIGURE 11-2

Spectral Centroid of a drum

Boolean Operators

Suppose we want to add a clip to the DAW if both the spectral centroid AND the RMS amplitude are above a certain threshold. So far, we have only seen conditional statements with one condition. How can we check for two conditions together?

We could do this by having a chain of if...else statements, or by nesting conditionals, but this can get unwieldy and difficult to read. A much better way is to use Boolean operators.

Boolean operators (a.k.a. logical operators) allow us to combine conditions. Remember that conditions evaluate to true or false, aka Boolean values. *A Boolean operator combines two Boolean values and produces one Boolean value*. Let's take a look at what this means:

```
// javascript code
//
// script_name: Boolean AND
//
```

```
// author: The EarSketch Team
//
// description: Using && (AND) to make a condition out of two conditions
//
//
//

//Setup
init();
setTempo(120);

//Music
var threshold = 0.2;
var spectralCentroid = 0.4;
var rms = 0.3;

if( (spectralCentroid > threshold) && (rms > threshold) ) {
        fitMedia(DUBSTEP_LEAD_006, 1, 1, 4);
}

//Finish
finish();
```

Above, we use the **AND** operator. The AND operator, written in JavaScript as &&, evaluates as true only when *both* of its operands (the conditions on either side) are true. Both our rms and spectralCentroid values are greater than threshold, so our AND statement is true: `true && true = true`.

There are several Boolean operators. Here are the three fundamental ones:

- **AND**: Returns true only when *both* of its operands are true. Written as &&.
    - `true && true == true`
    - `true && false == false`
    - `false && false == false`
- **OR**: Returns true only when *at least one* of its operands is true. Written as `||`.
    - `true || true == true`
    - `true || false == true`
    - `false || false == false`
- **NOT**: Returns the opposite Boolean. Written as `!`.
    - `! false == true`
    - `! true == false`

Let's use a combination of features to determine whether to add a set of clips to the DAW. Below, we chose a set of clips to add to a track based on their analysis values. Each set of clips is analyzed for both spectral centroid and RMS amplitude.

We sum the analysis values of the clips in each set (to obtain an overall value for the set). The analysis values of both features are compared between two clip sets: this gives us our condition that decides which of 3 clip sets to add to the DAW.

```
// javascript code
//
// script_name: Booleans
//
// author: The EarSketch Team
//
// description: Using Boolean operators to decide which clips get added to a song
//
//
//

//Setup
init();
setTempo(120);

//Music
var sectionAClips = [ELECTRO_STARLEAD_001,
                                YG_FUNK_FUNK_GUITAR_1,
                                ELECTRO_MOTORBASS_005,
                                Y01_DRUMS_1];

var sectionBClips = [EIGHT_BIT_ATARI_LEAD_010,
                                RD_UK_HOUSE__WARMPIANO_2,
                                DUBSTEP_SUBBASS_014,
                                ELECTRO_DRUM_MAIN_BEAT_002];

var sectionCClips = [YG_EDM_LEAD_2,
                                Y10_KEYS,
                                YG_NEW_HIP_HOP_BASS_9,
                                RD_POP_MAINBEAT_16];



function addSection(startMeasure, endMeasure, clips) {
        var length = clips.length;
        for(var i = 0; i < length; i++) {
                var track = i+1;
                fitMedia(clips[i], track, startMeasure, endMeasure);
```

```
        }
}


var spectralCentroidA = 0;
var spectralCentroidB = 0;
var rmsA = 0;
var rmsB = 0;

for(var i = 0; i < 4; i++){
        spectralCentroidA += analyze(sectionAClips[i], SPECTRAL_CENTROID);
        spectralCentroidB += analyze(sectionBClips[i], SPECTRAL_CENTROID);
        rmsA += analyze(sectionAClips[i], RMS_AMPLITUDE);
        rmsB += analyze(sectionBClips[i], RMS_AMPLITUDE);
}



if (spectralCentroidA > spectralCentroidB || rmsA > rmsB) {
        addSection(1, 5, sectionAClips);
} else if (spectralCentroidA < spectralCentroidB || rmsA < rmsB) {
        addSection(1, 5, sectionBClips);
} else {
        addSection(1, 5, sectionCClips);
}

//Finish
finish();
```

Sonification 12

Images as Data

If you could hear this image of the Orion Nebula what would it sound like?

FIGURE 12-1

Image: NASA/JPL-Caltech/STScI

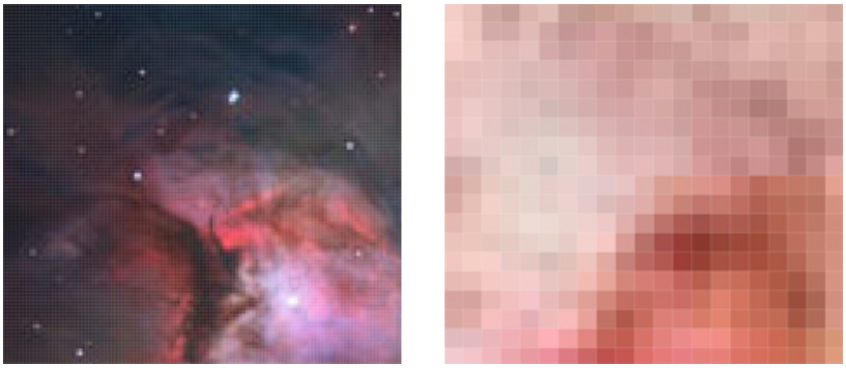
It might seem strange to think about hearing a picture, but think about it this way: the code that you write to make music in EarSketch is a kind of data, and so is this image. Sonification is a way to use non-speech audio to convey information, or in other words, turning data into sound. Sonification has a lot of practical applications, like giving the visually impaired a way to experience visual information through sound. We can also use sonification in music to convey

some thematic or structural connection to data or images or, as with any computer algorithm, to help us generate new musical materials that we might not otherwise think of.

Though you can sonify many different kinds of data, images are a fun place to start.

So how can we get music from an image? A digital image is really just made up of a lot of small pieces of data – and sonification lets us turn data into sound. Think about what happens when you zoom in on a picture, looking at it closer and closer. Eventually, you get to the smallest single component of an image. If you zoom in far enough, any image is just a lot of little squares of different colors. These are called *pixels*. Pixels are kind of like the atoms of images – the smallest part, with a big group of them making up the whole.

FIGURE 12-2



The color of each of these pixels is a piece of data. Every pixel has RGB values – red, green, blue. Your computer reads colors by knowing these values, which tell it how much red, how much green, and how much blue there is in that color. Each pixel also has a luminosity value – how light or dark it is. So you can use these numbers to either tell a computer what color you want it to produce, or in the case of image sonification, the computer can look at an image and tell what color each pixel is – producing numbers that you can use in EarSketch programs.

Multidimensional Arrays

We now know that any image is made up of a large number of pixels, each of which has number values that represent colors and brightness. Consider this row of four (zoomed in) pixels:

FIGURE 12-3

These pixels are grayscale, so they're easily represented by one number each that tells us how light or dark they are. Luminosity is on a scale of 0 to 255, from black (0) to white (255). The values of these four pixels are: 255, 189, 80, 0.

Remember our array data structure that can store any kind of information? The values of this row of pixels could be easily stored in an array.

```
var pixels = [255, 189, 80, 0]
```

Of course, most of the time an image isn't a single row. Instead, it's 2-dimensional. Any image, zoomed in to its pixels, could look something like this:

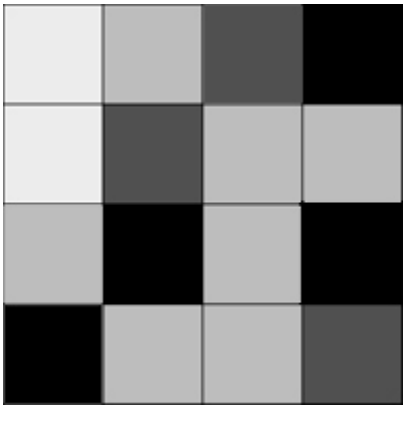


FIGURE 12-4

Now we have a square image with four rows. You could think of this as four different arrays:

row1 = [255, 189, 80, 0]
row2 = [255, 80, 189, 189]
row3 = [189, 0, 189, 0]
row4 = [0, 189, 189, 80]

As we learned, arrays can store any data type… this includes arrays! We can also represent our image as an array of arrays, or a multidimensional array. It's a matrix of luminosity values.

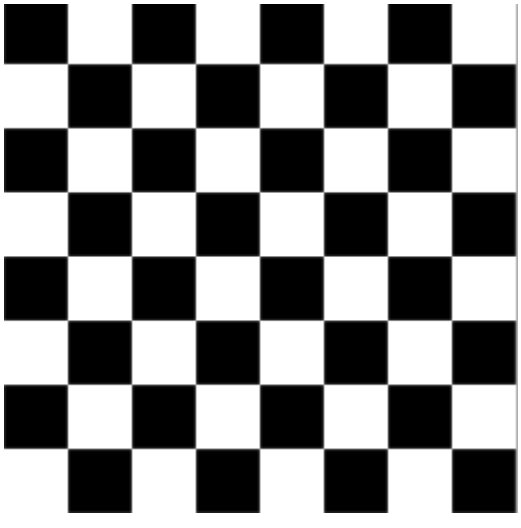
matrixExample = [[255, 189, 80, 0],
[255, 80, 189, 189],

[189, 0, 189, 0],
[0, 189, 189, 80]]

Notice how each array within the "matrix" array represents a single row of pixels. If we had the ability to step through that matrix and look at every value inside it, what could we do with those values? Using sonification, we can turn those numbers into sound.

Here's a simpler example. What if our matrix was just black and white? This is a zoomed-in image of an 8×8 square – each square is a single pixel.

FIGURE 12-5



What do you think this image looks like as data? Remember that the luminosity value of a black pixel is 0 and a white pixel is 255.

importImage

After you've made a guess about the data, here's a way for you to check it. If you want to work with images in EarSketch, you can use our **importImage** function that converts any image into data that can be used within EarSketch. `importImage` takes 3 required arguments (and one optional argument for color data). These are the 3 basic arguments:

```
importImage( imageURL , numberOfRows , numberOfColumns )
```

The image URL is the source of the image, so you will have to use a picture on the internet. If you have your own image file you want to sonify, you can use an online image host to upload it to the internet. To get the URL of an image online, try right-clicking it and choosing "Copy Image Location".

The number of rows and columns refers to the size of the matrix that `importImage` will return. The function actually divides the picture up into a grid of size numberOfRows x numberOfColumns, and returns the average value of each grid space.

To convert the black-and-white grid image from above into a multidimensional array of pixels, right-click it to copy the URL, and paste that in as your first argument. Choose 8 rows and 8 columns. It should look something like this:

```
init();

var myImage = importImage("http://earsketch.gatech.edu/wp-content/uploads/2013/03/grid_zoomed

println(myImage);

finish();
```

Take a look at what gets printed to the console. Was the data for that image what you expected? It should look something like this:

```
[[0,255,0,255,0,255,0,255],
[255,0,255,0,255,0,255,0],
[0,255,0,255,0,255,0,255],
[255,0,255,0,255,0,255,0],
[0,255,0,255,0,255,0,255],
[255,0,255,0,255,0,255,0],
[0,255,0,255,0,255,0,255],
[255,0,255,0,255,0,255,0]];
```

You can also specify any array dimensions for the output – you don't have to just use the original number of pixels. Our image here is only 8×8 pixels, which is very small. Imagine how many pixels most images you look at are! When you're working with images in EarSketch, you probably won't want to work with hundreds or thousands or millions of pixels – so instead, the Image Converter lets you turn any image into a multidimensional array of whatever size

you want. If you can't decide what size to use, choosing "16" for columns might work well, since that's the number of sixteenth notes in an EarSketch measure and the number of measures in many musical phrases. For now, put in "8" and "8" for the number of rows and columns, to match the data for our 8×8 grid.

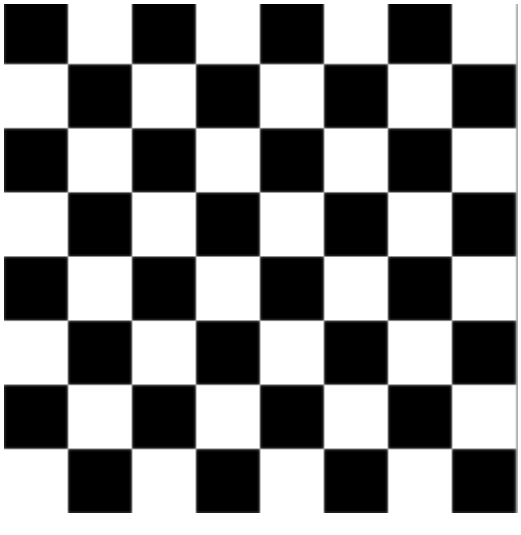
Most of the time, you'll just want your image to be in black and white (grey-scale), unless you're doing something in EarSketch specific to RGB color values. If you choose to use a color image, then you have the option to convert it to a 3D matrix. Since our grid is black and white, we don't have to worry about that for now.

Now onto sonification: we have data that EarSketch can understand, and we can turn that data into sound.

Nested Loops

Now that we know how to turn an image into data using `importImage`, we can use EarSketch to turn that data into sound. Let's go back to our checkerboard. We can turn it into a drum beat by writing code that will create a beat with hits on the black squares and rests on the white squares.

FIGURE 12-6



Remember from our lesson about arrays that we can access an element in a single-dimensional array like this: `array[index]`. For a two-dimensional array, we give coordinates: `array[index1][index2]`. So for the example above, `my-`

`Image[0][0] = 0` and `myImage[0][1] = 255`. This represents the first two pixels in the first row.

When we learned about iterating through arrays, we saw how easy it is to step through a one-dimensional array using a for loop. In order to step through a multidimensional array, we just need to use two for loops. We **nest** them together, so that an outer loop steps through each row, and an inner loop steps through each column for the current row.

If we wanted to step through the grid above to create a beat string (in a variable called drumBeats) with hits on black and rests on white, our nested loop could look like this:

```
for (var outerCounter = 0; outerCounter < myImage.length; outerCounter++) {
        for (var innerCounter = 0; innerCounter < myImage[0].length; innerCounter++) {
                // println("row:");
                // println(outerCounter);
                // println("column: ");
                // println(innerCounter);

                if (myImage[outerCounter][innerCounter]== 255) { // if the color is white
                        drumBeats[outerCounter] = drumBeats[outerCounter] + "-"; // then rest
                } else {
                        drumBeats[outerCounter] = drumBeats[outerCounter] + "0"; // otherwise
                }
        }
}
```

Let's step through this code line by line in order to understand it completely. First, line 1 defines a loop over the rows of the checkerboard image and defines the variable outerCounter to hold the value of the current row:

```
for (var outerCounter = 0; outerCounter < myImage.length; outerCounter++) {
```

Look closely again at the printed results of `myImage` (a 2-dimensional array):

```
[[0,255,0,255,0,255,0,255],
[255,0,255,0,255,0,255,0],
[0,255,0,255,0,255,0,255],
[255,0,255,0,255,0,255,0],
[0,255,0,255,0,255,0,255],
[255,0,255,0,255,0,255,0],
```

```
[0,255,0,255,0,255,0,255],
[255,0,255,0,255,0,255,0]];
```

The length of the `myImage` array is simply the number of arrays inside the 2-dimensional array, which is 8. We'll call this the number of rows in the `myImage` array. So the outerCounter variable will run from 0 to 7, which is exactly what we want.

Line 2 does nearly the same thing but specifies a different range of values to loop over:

Notice the `myImage[0].length`. This simply says the length of the `myImage[0]` array. We can see that `myImage[0]` is the first element in the outer array, which is the first array inside the `myImage` array:

```
[0,255,0,255,0,255,0,255];
```

```
[0,255,0,255,0,255,0,255]
```

The length of this array is 8. Thus, the `innerCounter` variable counts from 0 to 7. Thus, we can think of the inner loop as looping over each column in the current row of the `myImage` multidimensional array. The result is that `myImage[outerCounter][innerCounter]` will start at `myImage[0][0]`. At the end of the inner loop, the value of `innerCounter` will increase by one, and the index of the `myImage` array will change to `myImage[0][1]`. The inner loop will keep incrementing until it has reached the end of the first row, and then will finish and move again to the outer loop, which will increase the `outerCounter` value by one. Then the inner loop will start over again at the second row: `myImage[1][0]` and so on. To summarize, the `outerCounter` starts at 0, then it goes to the inner loop, where the `innerCounter` increases from 0 to 7. When the inner loop is finished with the current row, the computer will go back to the outer loop, increasing `outerCounter` to 1, and then inner loop will start again with the updated value, increasing the `innerCounter` variable from 0 to 7 again, and so on. To see this process in the console, uncomment the `println` statements in the loop.

Line 8 checks the value of the element in the 2-dimensional array, which (from the contents of our myImage variable) can only be either 0 or 255:

```
if (myImage[outerCounter][innerCounter]== 255) { // if the color is white
```

Here, the if statement checks if the current array value has a luminosity of 255 (that is, if it's white). What we want is to hit on the black pixels and rest on the white pixels. Here, the pixel is white, and so we rest by adding a "-" to a beat string that we create for the row (which is the drumBeats[outerCounter] list). If it's not 255, the else section of code is run:

```
} else {
        drumBeats[outerCounter] = drumBeats[outerCounter] + "0"; // otherwise
}
```

In that case, we add a "0" (or a hit) to the drumBeats[outerCounter] array. The end result will be one beat string for each row for each row of the checkerboard:

```
var drumBeats = [
'0-0-0-0-',
'-0-0-0-0',
'0-0-0-0-',
'-0-0-0-0',
'0-0-0-0-',
'-0-0-0-0',
'0-0-0-0-',
'-0-0-0-0',
'0-0-0-0-',
'-0-0-0-0'];
```

```
drumBeats = [
'0-0-0-0-',
'-0-0-0-0',
'0-0-0-0-',
'-0-0-0-0',
'0-0-0-0-',
'-0-0-0-0',
'0-0-0-0-',
'-0-0-0-0',
'0-0-0-0-',
```

```
'-0-0-0-0']
```

The code below does the rest of the work by using the drumBeats variable to create sound.

```
init();
setTempo(100);

println("here");

function createDrumbeats(myImage) {

        var drumBeats = [];
        for(var size = 0; size < myImage.length; size++) {
                drumBeats.push("");
        }

        for (var outerCounter = 0; outerCounter < myImage.length; outerCounter++) {
                for (var innerCounter = 0; innerCounter < myImage[0].length; innerC
                        // println("row:");
                        // println(outerCounter);
                        // println("column: ");
                        // println(innerCounter);

                        if (myImage[outerCounter][innerCounter]== 255) { // if the
                                drumBeats[outerCounter] = drumBeats[outerCounter] +
                        } else {
                        drumBeats[outerCounter] = drumBeats[outerCounter] + "0"; //
                        }
                }
        }
        return drumBeats;
}

// now, use the createDrumbeats function to create a list of drumbeats representing

var myImage = importImage("http://earsketch.gatech.edu/wp-content/uploads/2013/03/g

var snare = Y01_SNARE_1;

var drumBeats = createDrumbeats(myImage);

for (var counter = 0; counter < drumBeats.length; counter++){ //use the first row t
        makeBeat(snare, 1, counter + 1, drumBeats[counter]); // build a beat with t
}
setEffect(1, VOLUME, GAIN, 12);
```

```
finish();
```

A for loop at line 38 is used to to go through each beat string in the (one-dimensional) drumBeats array to place beats on consecutive measures of a track. Run this code, and then try tweaking it to use other images.

Of course, we can get a lot more complicated than checkerboards. One thing you can do is to try making beats with different images to see what happens. Yet another use for sonification is to think about it backwards – what might the beats that you want to make look like? Run the code below, listen for the beat, and think about what it might "look" like.

```
init();
setTempo(98);

var myImage = importImage("http://earsketch.gatech.edu/wp-content/uploads/2013/04/4x16_large.

var snare = Y01_SNARE_1;
var hats = Y19_CYMBAL_1;
var kick = Y01_KICK_1;
var drums = Y02_DRUM_SAMPLES_1;

var bass1 = Y32_BASS_1;
var bass2 = Y32_BASS_2;

var brass1 = Y32_BRASS_1;
var brass2 = Y32_BRASS_2;

var organ1=Y32_ORGAN_1;
var organ2=Y32_ORGAN_2;

var drumBeats=["","","",""];

for(var outerCounter = 0; outerCounter < myImage.length; outerCounter++) {

        for(var innerCounter = 0; innerCounter < myImage[0].length; innerCounter++) {

                if(myImage[outerCounter][innerCounter] >= 200) {  // if the color is white
                        drumBeats[outerCounter] = drumBeats[outerCounter] + "-"; // then rest
                } else {
                        drumBeats[outerCounter] = drumBeats[outerCounter] + "0"; // otherwise
                }
        }
}
```

```
var beatIndex = 5;

for(var counter = 0; counter < 8; counter++) {
        makeBeatSlice(hats,1,beatIndex, drumBeats[0], 1.0625);  // Open Hat
        makeBeatSlice(hats,2,beatIndex, drumBeats[1], 1+2*0.0625);  //closed hat
        makeBeatSlice(snare,3,beatIndex, drumBeats[2],1+ 4*0.0625); //snare
        makeBeatSlice(kick,4,beatIndex, drumBeats[3], 1+3*0.0625);  //kick

        beatIndex = beatIndex + 1;
}

for(var counter = 1; counter < 13; counter += 4) {
        fitMedia(brass1,6,counter,counter+2);
        fitMedia(brass2,6,counter+2,counter+4);
        fitMedia(bass1,8,counter,counter+2);
        fitMedia(bass2,8,counter+2,counter+4);
}

for(var counter = 5; counter < 13; counter += 4) {
        fitMedia(organ1,7,counter,counter+2);
        fitMedia(organ2,7,counter+2,counter+4);
}

//Outro
fitMedia(brass1,6,13,15);
fitMedia(bass1,8,13,15);
fitMedia(organ1,7,13,15);

setEffect(6,PAN,LEFT_RIGHT,30);
setEffect(7,PAN,LEFT_RIGHT,-30);
fitMedia(drums,5,5,13);

for(var counter = 5; counter < 13; counter += 2) {
        setEffect(5, VOLUME, GAIN,-60);
        setEffect(5, VOLUME, GAIN, -60, counter+1.75, -4, counter+1.75);
        setEffect(5, VOLUME, GAIN, -4, counter+2, -60, counter+2);
}

finish();
```

The project you just listened to used this image as the “myImage” value it was using to make a beat:

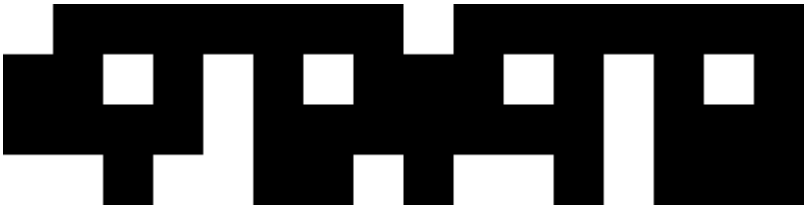


FIGURE 12-7

Exercises:

1. Some of the best images to use for sonification are those with an alternating pattern that can be used, for example, to create a beat string. It just so happens that QR codes typically have alternating (black and white) pixels with exactly these patterns. Try creating your own QR codes using a free QR code generator. After generating your code, right-click to get the image's URL. Then use the URL in `importImage` to obtain a list to use in your sonification code.

Sorting 13

Sorting and Analysis

In this section we'll learn a little bit more about what we can do with analysis features and arrays. If you recall, an array is a data structure that provides a way of storing and indexing many values in a single variable. These values will often be in no particular order (other than the order in which we inserted them into the array). However, we may want to order these values – perhaps from smallest to largest – to aid us in composition. To see how, try running the following code:

```
//Initialize EarSketch
init();
setTempo(120);

//Create an array of audio clips that you want to sort according to a particular analysis fea
var clipsList = [HIPHOP_MUTED_GUITAR_001, HIPHOP_MUTED_GUITAR_002, HIPHOP_MUTED_GUITAR_003, HI
HIPHOP_MUTED_GUITAR_007, HIPHOP_MUTED_GUITAR_008, HIPHOP_MUTED_GUITAR_009, HIPHOP_MUTED_GUITAR

//Declare what feature you'll be analyzing
var feature = SPECTRAL_CENTROID;

//SORTING
//Set up a left counter to step through clipsList, looking at each successive audio clip excep
for (var leftCounter = 0; leftCounter < clipsList.length - 1; leftCounter++) {
        //Set up a right counter to step through clipsList, looking at each successive audio
        for (var rightCounter = leftCounter + 1; rightCounter < clipsList.length; rightCounter
                //Obtain the clips at the positions of leftCounter and rightCounter
                var leftClip = clipsList[leftCounter];
                var rightClip = clipsList[rightCounter];
                //Obtain the clips' feature values using the analyze() function
                var leftCENTROIDValue = analyze(leftClip, feature);
                var rightCENTROIDValue = analyze(rightClip, feature);
                        //Use a temporary variable to swap the two clips if the feature of the
                if (rightCENTROIDValue < leftCENTROIDValue) {
```

```
                        var temp = clipsList[leftCounter];
                        clipsList[leftCounter] = clipsList[rightCounter];
                        clipsList[rightCounter] = temp;
                }
        }
}

var start = 1;
var end = 3;
//Insert the ordered audio clips on track 1
for (var index = 0; index < clipsList.length; index++) {
        fitMedia(clipsList[index], 1, start, end);
        start = start + 2;
        end = end + 2;
}

//The end!
finish();
```

This code sorts an array of similar audio clips according to their spectral centroids – darker clips are at the beginning of the array, and brighter clips are at the end. So, if you're creating a song and need the "brightest" of the hip-hop guitar clips, you would pick a clip closer to the end of this array. Consider that you could also sort an array of audio clips based on their RMS amplitudes (as we will soon see) to help you select either louder or softer clips, depending on what you want in your composition. In this way, a sorted array can give you a "palette" of sounds from which to choose for inclusion in your compositions, or you can gradually change from one type of sound (e.g. dark) to another type (e.g. bright) by placing the sorted clips one after another on a track. This example illustrates how teaching computers to listen through Music Information Retrieval can help you compose music. Next, we'll learn how to write the code to sort an array.

The method we will be using to sort arrays is called a **selection sort**; this name derives from the fact that this sort, on each successive "pass" through the array, "selects" the smallest element in the array and places it at the beginning of the array. The sort uses a nested loop with two counters, one beginning at the beginning of the array, and one beginning at the next position in the array. The left counter remains fixed while the right counter steps through the array elements, comparing each to the element at the left counter. Any time the element at the right counter is less than the element at the left counter, the two elements are swapped. So, at the end of the first "pass" through the array, the first element in the array is also the smallest. The process then repeats with the left counter starting at the second position in the array and the right counter

starting at the third position; after this pass, the second element in the array is the second smallest. The process repeats until the array is sorted. Let's take a look at how the selection sort works – here is some code that creates an array of five audio clips and sorts this array based on their RMS amplitudes, from low to high:

```
//Initialize Earsketch
init();
setTempo(120);

//Create a list of audio clips that you want to sort according to a particular analysis featu
var clipsList = [HIPHOP_MUTED_GUITAR_001, HIPHOP_MUTED_GUITAR_002, HIPHOP_MUTED_GUITAR_003, HI

//Declare what feature you'll be analyzing
var feature = RMS_AMPLITUDE;

//SORTING
//Set up a left counter to step through clipsList, looking at each successive audio clip excep
for (var leftCounter = 0; leftCounter < clipsList.length - 1; leftCounter++) {
        //Set up a right counter to step through clipsList, looking at each successive audio
        for (var rightCounter = leftCounter + 1; rightCounter < clipsList.length; rightCounter
                //Obtain the clips at the positions of leftCounter and rightCounter
                var leftClip = clipsList[leftCounter];
                var rightClip = clipsList[rightCounter];
                //Obtain the clips' RMS amplitudes using the analyze() function
                var leftRMSValue = analyze(leftClip, feature);
                var rightRMSValue = analyze(rightClip, feature);
                //Use a temporary variable to swap the two clips if RMS amplitude of the righ
                if (rightRMSValue < leftRMSValue) {
                        var temp = clipsList[leftCounter];
                        clipsList[leftCounter] = clipsList[rightCounter];
                        clipsList[rightCounter] = temp;
                }
        }
}

var start = 1;
var end = 3;
//Insert the ordered audio clips on track 1
for (var index = 0; index < clipsList.length; index++)  {
        fitMedia(clipsList[index], 1, start, end);
        start = start + 2;
        end = end + 2;
}

//The end!
finish();
```

Our first few lines of code create the list of audio clips to be sorted and define the feature to be analyzed. In lines 13 and 15, we create two counters: `leftCounter` starts at the beginning of `clipsList` and steps through all but the last element in the array; `rightCounter` starts at the second element in the array and steps through the remaining elements.

We then obtain the RMS amplitudes for the clips at the positions of both `leftCounter` and `rightCounter` using the `analyze()` function:

```
//Obtain the clips at the positions of leftCounter and rightCounter
                var leftClip = clipsList[leftCounter];
                var rightClip = clipsList[rightCounter];
//Obtain the clips' RMS amplitudes using the analyze() function
                var leftRMSValue = analyze(leftClip, feature);
                var rightRMSValue = analyze(rightClip, feature);
```

Lines 17 and 18 obtain the left and right clips for analysis from the `clipsList` variable (by using the `leftCounter` and `rightCounter` variables). Lines 20 and 21 use the `analyze()` function to derive the RMS amplitude of a clip. `analyze()` takes as its parameters an audio clip and an analysis feature. Our next few lines of code set up a conditional to determine whether or not the element at `rightCounter` is less than the element at `leftCounter`. If it is, the two elements are swapped using a temporary variable:

```
//Use a temporary variable to swap the two clips if RMS amplitude of the right clip
                if (rightRMSValue < leftRMSValue) {
                        var temp = clipsList[leftCounter];
                        clipsList[leftCounter] = clipsList[rightCounter];
                        clipsList[rightCounter] = temp;
                }
```

Swapping the left and right clips uses a very common programming technique called a “swap” of two variables. At line 26, a new `temp` variable is defined to temporarily hold the left clip. Then at line 27, the value in`clipsList[leftCounter]` is updated to the right clip (specifically, the value at `clipsList[rightCounter]`. Lastly, at line 28, the `clipsList[rightCounter]` val-

ue is updated to the original “left” clip. The end result is to swap the two values, thus sorting the two clips by their RMS amplitude!

During the first iteration of the outer “for” loop (at line 15), after the inner “for” loop (at line 17) has finished all of its iterations, the first element in `clipsList` will be the clip with the smallest RMS amplitude. Then,`leftCounter` is incremented in the outer “for” loop, and the process starts over, this time with `leftCounter` starting at the second element in the list and `rightCounter` starting at the third. The sort algorithm continues in this manner, placing clips with successively larger RMS amplitudes at successive positions in the list, until the entire list is sorted in order of ascending RMS amplitude. If this is confusing, view the animated graphic of selection sort in the **selection sort Wikipedia page**.

The last part of our code adds each element in our sorted list (which should now contain all of our audio clips in order from those with the lowest RMS amplitude to those with the highest) to track 1, allowing us to play them in order and hear the increase in RMS amplitude from clip to clip.

Exercises

See if you can sort these audio clips based on their spectral centroids from high to low.

14 Recursion

What is a Fractal?

In this module we introduce an important concept in both computer science and artistic practice: the idea of *self-similarity*, and its related programming technique *recursion*. Self-similarity is when a part of an object is similar to the entire object.

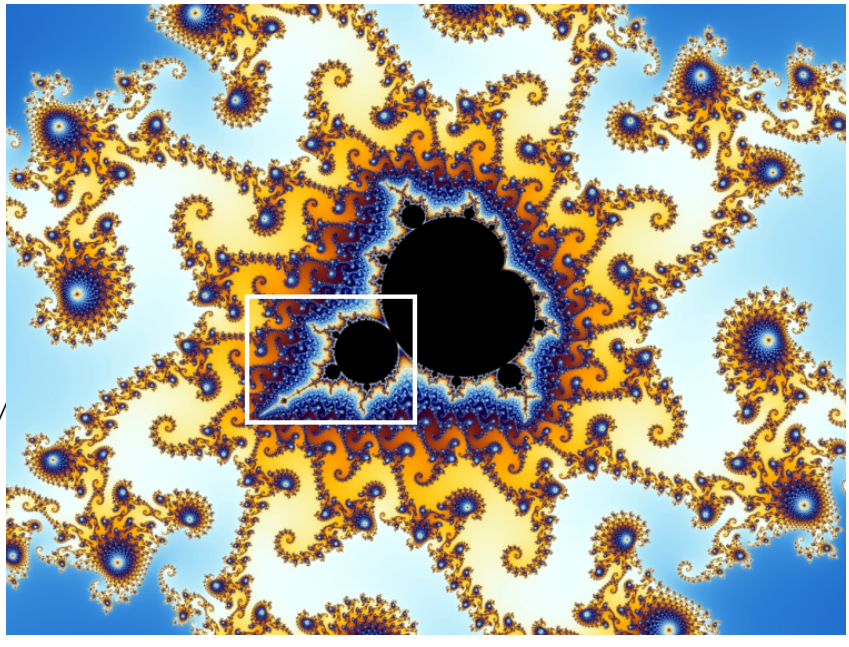
A common property of self-similar designs, which are sometimes called *fractals*, is that they contain the same pattern at various levels.

The four pictures below are examples of a popular fractal design known as the *Mandelbrot Set*.

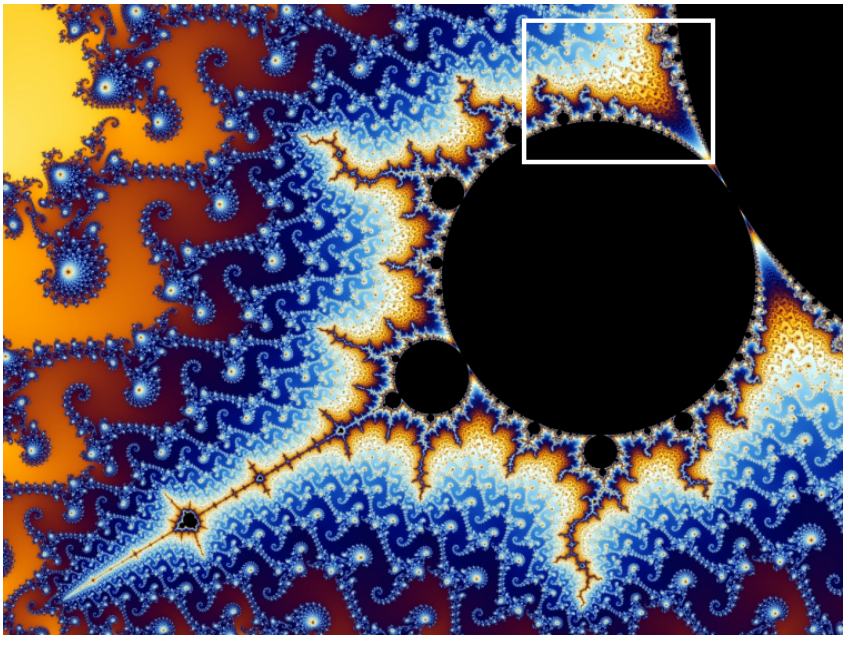
Quite profound examples of self-similarity can be discovered throughout the Mandelbrot Set. Looking at the first picture, notice the part outlined in the white rectangle…

FIGURE 14-1

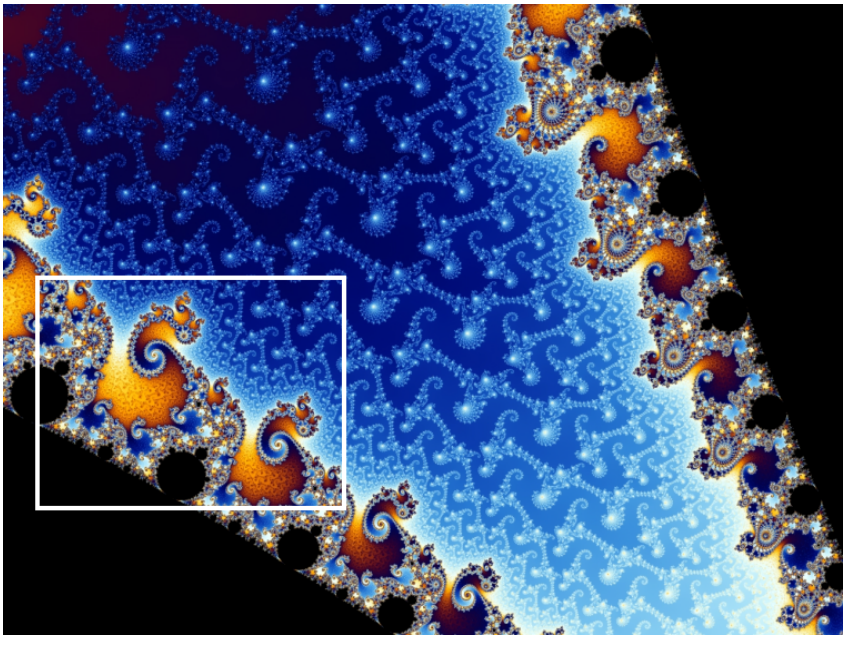
Created by Wolfgang Beyer – from the Center for Image in Science and Art – Flicker photostream – used with permission through Creative Commons free-use license. http://www.flickr.com/photos/lcisa/4749984061/in/set-72157624105468823/



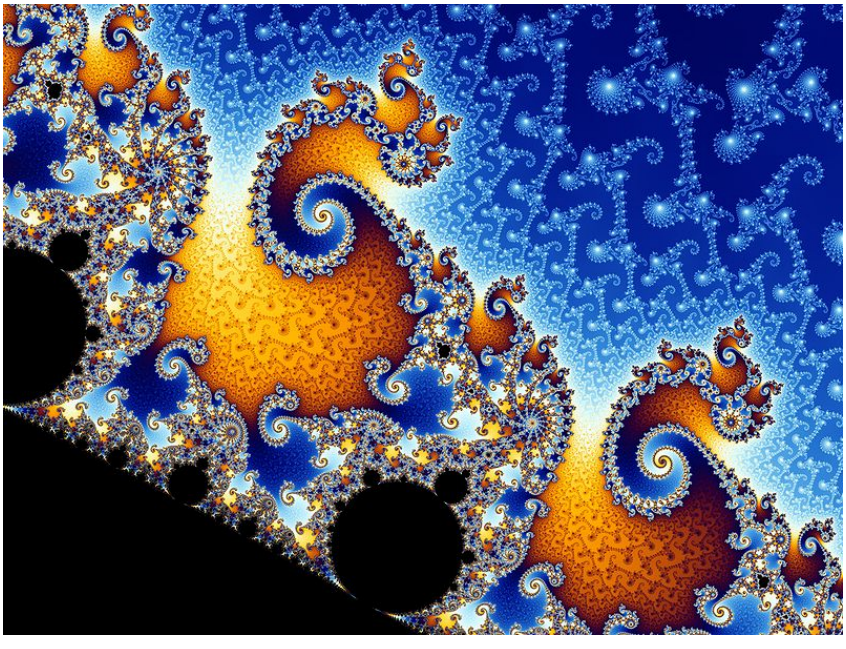
… is actually the entire second picture below!



Now look at the pattern within the white rectangle of the above picture, which when zoomed into, becomes this entire picture below:



And one more time, the design within the white rectangle above is blown up to reveal this whole picture below:



Notice the recognizable main pattern of the Mandelbrot Set in black at the bottom of the above picture. It looks like a beetle or a snowman. Referring back to the very first picture, you can see this exact same shape within the white rectangle, as well as scattered throughout each of the pictures. This is self-similarity on a grand scale!

ADDITIONAL LINKS

Interactive Fractal Zoomer: Here you can move around as well as zoom in and out of a Mandelbrot set.

Deep Mandelbrot Zoom: A very fast zoom video into the Mandelbrot set.

In the next section we'll be introduced to recursion, and we'll begin applying some simple self-similar designs to the composition of music in EarSketch.

What is Recursion? (Part 1)

In computer science, one of the main ways that self-similarity is expressed is through a technique called recursion. Recursion is found whenever a function calls itself from within the body of its own code. In the example function `countdown()` below, we can see at line 8 that it calls itself from within its own body of code. This means that `countdown()` is a *recursive function*, and the call at line 8 is a *recursive call*.

```
function countdown(n){
        if(n == 0) {
                println("GO!");
                return;
        }
        println(n);
        countdown(n-1);
}

countdown(3);
```

In order to understand how recursion works, it's helpful first to distinguish in your mind the difference between:

1. The definition of a function in code (in an EarSketch script, for example) and...
2. The actual running of that function (whenever the function is called).

A function can have only one valid definition, but may have an unlimited number of calls to it within a script. `countdown()`is a good first example of recursion, because it clearly shows the main elements of what is needed for recursion to work properly.

First of all, everything that happens directly within a function call is said to be within that function's *scope*. This includes local variable definitions, but does not include what may happen inside any other function that is called within its body.

Using `countdown()` as an example, an integer parameter n is passed in. This parameter is now within the scope of this particular call of `countdown()`.

At line 4, this parameter n is tested to see if it is equal to zero. This is called the *stopping condition*.

If the stopping condition turns out to be True (here it's when n is equal to zero), then that particular call of `countdown()` returns immediately, passing control back to whatever scope called it. This is either another function or the

"top level" of code that is written in the script, outside the scope of any function (shown by line 10 in the above code example).

If the stopping condition is False, print is called, which prints the value of n (shown by line 7).

It is important to understand that a correctly functioning stopping condition is vital for recursion to work properly. If the stopping condition fails to cause the function to return when it should, there is danger of entering an infinite loop, which will crash the program and possibly the computer. This is why the stopping condition is tested each time the function is called, and must evaluate to True at some point (e.g. n must equal zero at some point). This insures that at some point the most recent recursive function call will return, and start the chain reaction that will return each recursive call back up to the initially called function, which returns with the final result.

On line 8 we find a clear example of a recursive call. Here `countdown()` calls itself, and passes `n - 1` in as its integer parameter. This new call to `countdown()` is within its own newly created scope, which is fully separate from the scope of the particular use of `countdown()` which called it.

If this sounds confusing, then look at the definition of `countdown()`, follow it line by line and imagine what it does, while also referring to the two diagrams below. When you see what is going on here, you'll understand this process of recursion.

Now for a concrete example:

When the code example above is run as an EarSketch script, first the function definition of `countdown()` is processed, and then this very function is called from the "top level" of code, as shown by line 10. Thus `countdown(3)` is called directly from the "top level" of code, outside the scope of any other function.

Since n equals 3 here and not zero, this number 3 is printed to the EarSketch console on its own line, and then `countdown()` calls itself, passing in a parameter of 2… (n minus 1). The highest level scope of `countdown()` (which just made the recursive call to itself) is actually not finished yet, since control has passed to the recursive call of `countdown()`, that is shown by line 8 in the function's definition. Thus the highest level of `countdown()` is actually waiting for the recursive call it just made to return.

As we stated above, this recursive call to `countdown()` is passed a parameter of 2 (n minus 1, since n in the highest level scope of countdown() is equal to 3).

Now the whole process repeats:

2 is not equal to zero, so it's printed on its own line, and then is decremented by one and passed into a new recursive call of countdown(). Now we have n = 1 as the input to this third level of countdown().

1 is not equal to zero, so it's printed, and is then decremented by one and passed into to a new recursive call of countdown().

Since n = 0 here at last, the stopping condition test in line 4 returns True, and that particular call of countdown() immediately returns to the one that called it (which is the next higher scope of countdown(), one level up).

This higher scope of countdown() then immediately returns to the scope that called it. This keeps happening until the initial countdown() function call returns to the scope which called it, which is the "top level" of code in the EarSketch script, outside of any function (shown by line 10 in the code example above).

By viewing the output of countdown(3) in the picture below, and also the diagram below that which shows the flow of control starting from the original call to countdown(3) on line 10, you can see what recursion is all about. Once you understand its basic operation, you will know how recursion works at its core.

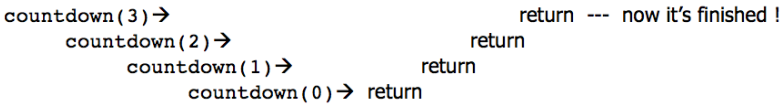
One more quick term: The parameter n that is passed into the original call to countdown() at line 10, can be thought of here as the depth of recursion. This is because there are n recursive calls between the top level function call and the very last one which returns because its input parameter of n is equal to zero.

This is what countdown(3) shows in the console when run from an EarSketch script:

FIGURE 14-2

This is an illustration of what is happening in the computer when a recursive function like `countdown(3)` is called from code:

FIGURE 14-3



Note that any two commands on a row (countdown(), return) belong to the same scope, while each separate row belongs to its own separate scope.

What is Recursion? (Part 2)

Now that we understand the basics of recursion, it's time to see how we can use recursion to make music with EarSketch.

In the example function `placeSounds()` below, we see at line 7 that it calls itself from within its own body of code.

This means that `placeSounds()` is a *recursive function*, and the inner call to itself on line 7 is a *recursive call*.

```
// A simple recursive function which places sound clips on consecutive measures
function placeSounds(soundlist, start){
        // this test is called a STOPPING CONDITION.. this particular one tests if "soundlist
        if(soundlist == []) return; // when the result of the stopping condition test is True

        fitMedia(soundlist[0], 1, start, start+1);
        placeSounds(soundlist.slice(1, soundlist.length), start+1);
}
```

Please note that for conciseness here, whenever we use the term "sound" we mean an audio clip (e.g. `placeSounds()` means "place audio clips").

`placeSounds()` works like this:

You supply it with a list of audio clips and a start measure location (shown by the function's parameters on line 3)

It places the first audio clip in the list on track 1 at the start measure location, and ends the clip one measure later (shown by line 6)

It then calls a new run of itself (at line 7), supplying as parameters:

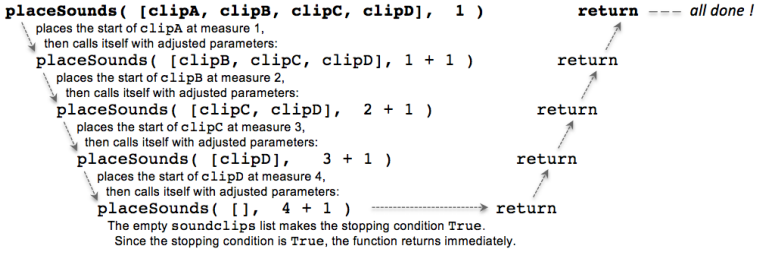
– the remaining audio clips in the list (every audio clip in the list except the first one, which was the one just placed)

– its start measure location increased by one measure

This keeps "recursing" until there are no more audio clips in the list, at which point the stopping condition tests True and the work of the function has ended.

Let's examine specifically what happens when we call **placeSounds()** from code, with a list of four audio clips and a start measure location of 1 (see diagram below):

FIGURE 14-4



As shown by the above diagram, the first thing that happens is that **clipA** is placed at measure 1 (on track 1), and ends one measure later (see line 6 of the code).

Then a recursive call is made to the same function, supplying as parameters a shortened audio clip list (**[clipB, clipC, clipD]**), and the just-used start measure location increased by 1 (which becomes 2).

Now **clipB** is placed at measure 2 (which is the value of the start parameter that was provided to this particular call of **placeSounds()**), and ends one measure later.

A new recursive call is made, supplying as parameters a further-shortened audio clip list (**[clipC, clipD]**), and the start measure location increased again by 1 (which becomes 3 here).

Next, **clipC** is placed at measure 3 and ends one measure later.

Another recursive call is made, supplying as parameters a further-shortened audio clip list (**[clipD]**), and the updated start measure location (which becomes 4).

The last audio clip of the original list, **clipD**, is placed at measure 4 (and ends one measure later).

Another recursive call is made, supplying as parameters: an empty list, and the updated start measure location (which becomes 5 — although this is never used… see the next line below).

Now since an empty list has been provided for the `soundlist` parameter, the test at line 4 of the code becomes **`True`**, and the function immediately returns. This causes the function that called it to return as well, which causes the function that called that one to return, etc… all the way back to the original function call from code, which finally returns (see the above diagram for a clear depiction of this process).

Here's the full code example of recursion:

```
init();
setTempo(128);

// a recursive function
function placeSounds(soundlist, start){
        if(soundlist.length == 0) return;    // if the soundlist parameter contains an empty

        // otherwise, place the first sound from the list on track 1 at the measure given by
        fitMedia(soundlist[0], 1, start, start+1);
        placeSounds(soundlist.slice(1, soundlist.length), start+1);  // now it calls itself w
        // updated parameter 1 : the rest of the soundlist (all remaining sounds, except the
        // updated parameter 2 : start+1 (as the new start measure for the recursive call)
}
// assign sounds
var clipA = Y45_SYNTHHARP_1;
var clipB = Y45_SYNTHHARP_3;
var clipC = Y45_SYNTHHARP_2;
var clipD = Y45_WHITEBUILD_1;

// create song

placeSounds( [clipA, clipB, clipC, clipD],  1);

finish();
```

In general, a recursive function works something like this:

- The function is called from code, with its required parameter(s) as input
- One of the input parameters is tested to see if it is equal to some value (this is called the stopping condition).
- If the result of this test is *true*, the function returns immediately, without running the rest of the code below the stopping condition.

- If the result of this test is *false*, the function keeps on going and runs the rest of its code.
- Assuming the stopping condition has failed, the rest of the code in the function body usually does something like this:
- Perform the main task(s) of the function
- Change the input parameter(s) to new value(s), and supply them as input to a new recursive call of the same function.

Here's a more complete musical example to run in EarSketch:

```
init();
setTempo(124);

// similar recursive function to placeSounds() in last example
// two extra parameters have been added here.. tracknum and clip length
function placeSoundsOnTrack(soundlist, tracknum, start, cliplength) {
        if(soundlist.length == 0) return;
        println(start);

        fitMedia(soundlist[0], tracknum, start, start+cliplength);
        placeSoundsOnTrack(soundlist.slice(1, soundlist.length), tracknum, start+cl
}
// set up new variables to access specific folders of audio clips
var DRUMFOLDER  = TMAINLOOP;
var BASSFOLDER  = EABASS;
var SYNTHFOLDER = ELEAD;
var BLIPFOLDER  = EIGHTATARISFX;

// set up arrays to hold the audio clips that will be randomly selected from the fo
var drumclips  = [];
var bassclips  = [];
var synthclips = [];
var blipclips  = [];

// fill up the arrays with random audio clip selections from specified folders:

// these audio clips will be placed every two measures, so 4 audio clips will fill
for(var i = 0; i < 4; i++) {
        drumclips = drumclips.concat(selectRandomFile(DRUMFOLDER));
}
// these audio clips will be placed every two beats (0.5 measures each), so 16 audi
for(var i = 0; i < 16; i++) {
        bassclips = bassclips.concat(selectRandomFile(BASSFOLDER));
}
// these audio clips will be placed every three 8th-notes (0.375 measures each), so
// since 8 measures / 0.375 = 21.333..., we can set this to use 21 audio clips, whi
for(var i = 0; i < 21; i++) {
        synthclips = synthclips.concat(selectRandomFile(SYNTHFOLDER));
```

```
        blipclips  = blipclips.concat(selectRandomFile(BLIPFOLDER));
}

placeSoundsOnTrack(drumclips, 1, 1, 2);          // place a new audio clip every 2 measures
placeSoundsOnTrack(bassclips, 2, 1, 0.5);        // place a new audio clip every 0.5 measures
placeSoundsOnTrack(synthclips, 3, 1, 0.375);      // place a new audio clip every 0.375 measu
placeSoundsOnTrack(blipclips, 4, 1.125, 0.375);  // start the audio clips on this track one e

// use volume effects to set up a balanced mix of the four tracks
setEffect(1, VOLUME, GAIN, 0);
setEffect(2, VOLUME, GAIN, -6);
setEffect(3, VOLUME, GAIN, -12);
setEffect(4, VOLUME, GAIN, -9);

finish();



// INTERESTING TIP:
//  since this script uses randomness, each time you run this it should produce a different s
```

Why not just use for loops? This is a valid question, because if examples like the above were all we were planning on doing, then for loops would be a more straightforward way to achieve the same thing. Here's an alternate version of placeSoundsOnTrack() that uses iteration instead of recursion:

In the next sections, we'll explore recursive techniques that are much more difficult to implement with for loops and which connect back to the idea of self-similarity and fractals.

MORE INFO ABOUT RECURSION

One of the basic principles of programming is that the same function may be called multiple times within a code script (as many times as it's needed).When this happens, each of the calls to the same function is run within its own separate scope. Each of these separate function calls runs the same procedure (as stated in the function's definition), in its own separate scope with its own set of parameter values as input. In the case of recursion, when a recursive function is called, that same function is called again within a new separate scope, and is called before the parent function ends.Thus we can think of any recursive function call (as it's running) and its scope as being fully "inside" of the scope of the particular function-call which had called it, like a set of nested Russian dolls. Using this analogy, the largest doll is the original call to the function from code (a call from outside of the function's definition), while the smallest doll is the final recursive call made – the one that the stopping condition tests True on, which tells that specific call of the function to return. This in turn causes the next-to-last call of the function to return (corresponding to the next larger doll, the one that the smallest doll is directly inside of). One-by-one, each of the recursive function calls return as they cascade upward and out until the original function call returns (the largest doll), and the process is complete.

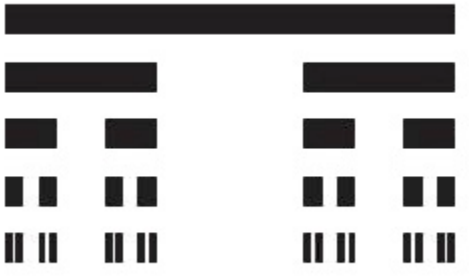
FIGURE 14-5

Russian nesting dolls

Cantor Set

As a straightforward example of self-similarity that may be used toward great musical effect, consider the Cantor Set shown below.

FIGURE 14-6



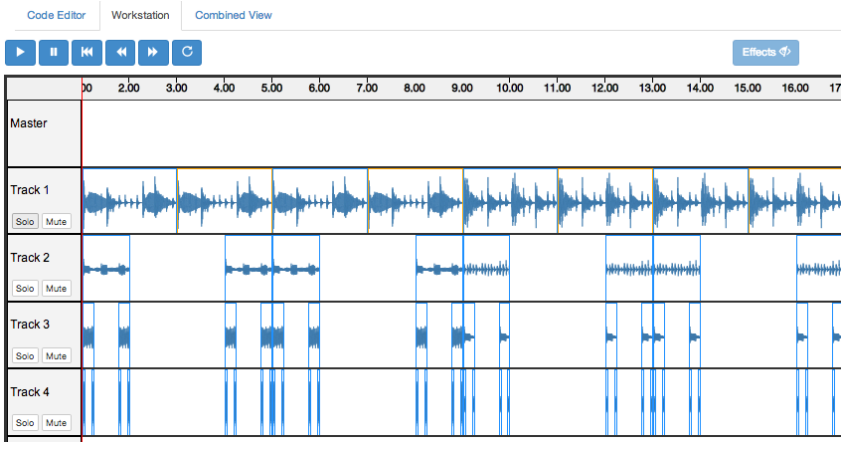
The steps required to create this self-similar design is as follows:

1. Start with a horizontal line segment.
2. Make a copy of this line immediately below it,
3. Divide the new line into three parts.
4. Remove the middle of the three parts - as shown in the first two steps of the diagram above, we change from a single solid line to two smaller lines with a space in the middle.
5. For each of the two lines just made, repeat from step 2 above

That's it! That is all one needs to know (either a human or a computer), to create the full design of the Cantor Set.

To apply this design toward an arrangement of sound clips in EarSketch, we can use the line-by-line pattern of a Cantor Set to specify where sound clips should be placed on consecutive EarSketch tracks, yielding something like this:

FIGURE 14-7



As part of the code example for this section, we've created a function that places sound clips on consecutive tracks in EarSketch, according to the pattern of a Cantor Set.

```
makeCantorSet(musicList, 1, 1, 4, 4)
```

When calling the above function, we supply it with these parameters:

audioclips	a list of audio clips (one for each track)
tracknum	the track number we want it to start at
start	the measure number we want it to start at
length	the length of the full pattern in bars
depth	a depth amount

The last parameter ***depth amount*** specifies the total number of tracks we want the function to create, which corresponds to the same number of lines in a Cantor set pattern (see the first diagram at the top).

audioMedia/1_CantorSetMix.mp3

```
init();
setTempo(135);
```

```
function makeCantorSet(audioclips, tracknum, start, length, depth) {
        // parameters: list of audio clips, starting track number, starting measure, total le

        if (depth == 0) return;                                          // when depth reache

        fitMedia(audioclips[0], tracknum, start, start+length);   // place the first audiocli
        var smallerLength = length / 4.0;                            // calculate a new leng
        var secondSectionStart = start + (smallerLength * 3.0);      // calculate the start
        // to make each of the two smaller sections on the next track, recursively call the f

        makeCantorSet(audioclips.slice(1, audioclips.length), tracknum+1, start, smallerLengt
        makeCantorSet(audioclips.slice(1, audioclips.length), tracknum+1, secondSectionStart,
}

var soundList1 = [DUBSTEP_DRUMLOOP_MAIN_001, Y36_ELECTRO_1, DUBSTEP_BASS_WOBBLE_025, ELECTRO_
var soundList2 = [DUBSTEP_DRUMLOOP_MAIN_007, Y43_SYNTH_HARP_1, Y36_ELECTRO_1, Y35_ELECTRO_2,


makeCantorSet(soundList1, 1, 1,  4, 4);
makeCantorSet(soundList1, 1, 5,  4, 4);
makeCantorSet(soundList2, 1, 9,  4, 4);
makeCantorSet(soundList2, 1, 13, 4, 4);

fitMedia(Y35_ELECTRO_2, 5, 1, 9);
fitMedia(Y35_ELECTRO_3, 5, 9, 17);

finish();
```

We see that on lines 14 and 15, `makeCantorSet()` calls itself twice from within itself. Thus `makeCantorSet()` is a **recursive function,** and its calls to itself on lines 14 and 15 are **recursive calls.**

Notice that in this example, there is more than one recursive call used within the definition of the function. These two recursive calls correspond to step 5 in the Cantor Set design instructions at the top of the page: one recursive call for each of the two new lines just created in step 4. And since there are *two* recursive calls here, each *level of recursion* has two times the number of recursive calls as the preceding level, shown by each level of the Cantor Set pattern – see both diagrams above !

The Thue-Morse Sequence

Binary numbers, the basic language of all computers and digital systems, are made up of just zeros and ones in various patterns.

The value of a single binary digit is either a zero or a one. These two possible values may also be thought of as False (0) and True (1).

We can take the **complement** of any binary digit, which means to change it to its opposite value (0 becomes 1, while 1 becomes 0…that's it!).
When using a binary sequence to define a rhythm, we can treat every zero value as a rest, and every value of one as a note or audio clip.

We will see this ahead shortly.

The **Thue-Morse sequence** is a binary sequence (just zeros and ones), which never fully repeats. Thus it is theoretically infinite!

It does have an elegant and striking self-similar pattern though, which makes it great to use for making musical rhythms that never fully loop or repeat, but that maintain a good balance between some repetition and some change. Here we build up this sequence as a string of zeros and ones.
This unique sequence is constructed by following a few simple steps:

1. Start with the two digit string, '01'.
2. Replace every '0' in the string by '01', and replace every '1' in the string by '10'
3. With the newly-created string from the previous step, go back to the beginning of step 2, and replace each '0' and '1' with the same values as before.

Here's what we get for the first few results of applying the above pattern:
'01' ->
'0110' ->
'01101001' ->
'0110100110010110' ->

There are some very cool properties of this pattern.
First of all, there is another way to construct this pattern besides the above steps:

Start with '01' again, find its **complement** (see above), and put it directly after.

'01' –get_complement–> '10' –place_it_after_the_original–> '01**10**' (see the original in regular text and its complement directly after it in boldface).

Notice that the result above is the same as the second line in the original method above.

You get the picture.. If we take the complement of the new string and put it after it, we'll get '0110 **1001**'.

These are the two ways to construct the Thue-Morse sequence, but there is one more interesting property worth mentioning:

You can form a string of every other digit of the Thue-Morse sequence.
Start with the first digit at the beginning of your created Thue-Morse string, and

take every other digit until you have a string of any length ***n*** you wish.
This length n string of every other digit of the TM-sequence will be the same exact string as the first n values of the original sequence!

We have just shown above the basic self-similar design pattern of the Thue-Morse sequence.

The coexistence of its special properties give this interesting sequence its uniqueness, and make it good musical material for use within an EarSketch script.

In our code example below, we will need to use an integer parameter n (depth of recursion) and a stopping condition (if n == 0) to specify and control how many times the above pattern recurses (repeats in on itself).

After listening to the audio example below, read over the code, then work through the diagram and its explanation while referring back to the code.
This will provide a solid understanding of how recursion is working to construct this sequence.
Then run the script in EarSketch. Follow the comments in the code to see how you can apply the Thue-Morse sequence to create interesting non-repeating rhythmic parts.

Start modifying some of the code to make your own piece:

- You can change the sounds
- You can use the `thuemorse()` function with a different number for the input parameter
- You can extract different substrings from the string created by `thuemorse()` and supply these different substrings as "rhythm-strings" to different calls to `makeBeat()`, similar to what is shown in the code example below (at lines 36 & 37).

audioMedia/2_ThueMorseSequence.mp3

```
init();

setTempo(135);

// this self-similar recursive function creates the Thue-Morse sequence

function thuemorse(n) {
        if (n == 0) {
                return '0';
        }
        var oldstring = thuemorse(n-1);
        var newstring = '';
        for (var i = 0; i < oldstring.length; i++) {
```

```
                if (oldstring[i] == '0') {
                        newstring = newstring + '01';
                } else newstring = newstring + '10';
        }
        return newstring;
}

// sounds and data used in the piece

var drumloop = DUBSTEP_DRUMLOOP_MAIN_001;
var soundList1 = [Y51_PERCUSSION_1, Y62_PERCUSSION_2];
var soundList2 = [TECHNO_ACIDBASS_011, Y04_DRUMS_SAMPLE_1]; // Y35_ELECTRO_2 HIPHOP_

var rhythmString = thuemorse(7);

// the thuemorse() function with a depth parameter of 7 generates a string of lengt
// since 128 consecutive 16th-notes lasts for eight measures of 4/4, the use in mak

// code to generate piece

fitMedia(drumloop, 1, 1, 9);

makeBeat(soundList1, 2, 1, rhythmString.slice(0, 64)); // use first half of rhythmS
makeBeat(soundList2, 2, 5, rhythmString.slice(64, 128)); // change sounds, and use

finish();
```

How does the `thuemorse()` function use recursion to build up the binary string that it returns?

Let's say we call `thuemorse(4)`. The first thing that is done is to test to see if its input parameter is equal to zero (see code line 8 above).

Since 4 does not equal zero, the stopping condition fails and the program moves to the next line of code (line 11).

Here, something interesting happens. A new variable named `oldstring` is created to hold the result of a recursive call to `thuemorse(3)`.

Since this recursive call is made here, the function needs to wait until `thuemorse(3)` finishes and returns with a value, before continuing on to the next line.

So we see here that `thuemorse(4)` is patiently waiting for `thuemorse(3)` to return.

What is `thuemorse(3)` doing?

It tests to see if 3 is equal to 0 and since it isn't, it creates the variable called `oldstring` to hold the result of `thuemorse(2)` .

Hold on! I thought we're already using a variable named `oldstring`. Why does this work?

Remember that each call to a function establishes a ***scope*** that is just for that particular call of the function. Each time a recursive call is made here to `thuemorse()`, a new scope is created to keep track of what's going on just inside that one particular call. The variable named `oldstring` that is created by `thuemorse(3)` is a totally different variable than the `oldstring` that was created by `thuemorse(4)`. This is because they each exist inside a different scope.

Say for example you have two friends (Joe A and Joe B), who live on different streets (Avenue A and Avenue B).

When you are on Avenue A, you ask if anyone has seen Joe, and it is understood that you are looking for Joe A.

If you go over to Avenue B and ask for Joe, it's understood there that you are looking for Joe B.

This is because Joe B is local to Avenue B, and Joe A is local to Avenue A.

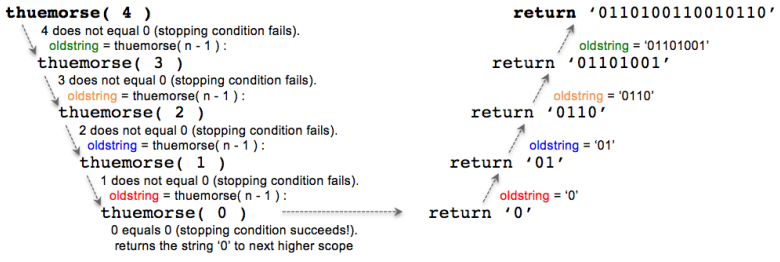
It's the same concept when thinking about scope:
Variables which have the same name (joe), but which are each local to the scope of a different function call (joe from Avenue A vs. joe from Avenue B) are actually different individual variables!
Remembering our local variable `oldstring` in the `thuemorse()` function, we can say that it has the same properties as "joe" in the above story.

Each call to `thuemorse()` will create and use a new variable named `oldstring`, which remains totally separate from any other variables named `oldstring` that were created by other calls to `thuemorse()`.

We see that the `oldstring` that is local to `thuemorse(3)` is waiting for `thuemorse(2)` to return, before `thuemorse(3)` can continue.

I think we know what's going to happen next. Follow the diagram below (down the left side, across, then up the right side) to quickly see what is going on.

FIGURE 14-8



Notice that each time an `oldstring` variable is created, it is shown in a new color to emphasize that it is a brand new variable, totally separate from any other `oldstring` of a different color.

The function keeps recursing like it's shown in the diagram below, until the stopping condition succeeds when `thuemorse(0)` is called. In line 9 of the code, it shows that `thuemorse(0)` returns the string '0'.

Since `thuemorse(1)` left off with its `oldstring` waiting for the return value of `thuemorse(0)`, this `oldstring` gets as its value the '0' that was returned by `thuemorse(0)`.

Now the rest of the code in the function can be worked through for `thuemorse(1)`. First we create a new variable which will hold some new information that will be created ahead. We call this new variable `newstring`, and initialize it with an empty string.

Next each single character of `oldstring` is tested to see if it's a 0 or 1, and its corresponding two character string is added on to `newstring`. This is the way that the Thue-Morse string is actually created.

Remember that we left off in `thuemorse(1)`, working our way back up the right-hand-side of the diagram. Since the local variable `oldstring` is '0', the local variable `newstring` gets the value '01' which is returned as the result of the call to `thuemorse(1)`.

Who does it get returned to? It gets returned to whomever called it! Specifically, `thuemorse(1)` was called by the line 11 of `thuemorse(2)`, which has been waiting for `thuemorse(1)` to return with a value.

Since `thuemorse(1)` returns with the string '01', this value gets assigned to the `oldstring` variable that is local to `thuemorse(2)`. Now `thue-`

`morse(2)` continues on through the rest of its code, which results in another new local variable called `newstring` getting the next level expansion of '01', which turns out to be '0110'.
This value of `newstring` gets returned as the value of `thuemorse(2)`, and assigned to the `oldstring` local variable of `thuemorse(3)`, then `thuemorse(3)` continues on through the rest of its code.

This repeated pattern happens here one more time, until the original, non-recursive call to `thuemorse(4)` is returned, retuning the value '0110100110010110', and the function is done.

It is interesting to note that a call to `thuemorse(`*n*`)`, with any number for *n*, works in the same way as a call to `thuemorse(4)` – the function keeps recursing until the stopping condition succeeds (this is called the **base case**). The base case, here `thuemorse(0)`, returns with a value, and supplies that to the function which called it, and so on, all the way back up the right-hand-side of the diagram, the string doubling in length each time, until the final string is returned as the value of the original call.

Noting that the string doubles in length each time, we can see that a call to `thuemorse(8)` for instance, will return a string of 256 characters (2 to the 8th power). This could be used as a rhythm string for 16 bars of 16th notes, or divided up into substrings.

The Towers of Hanoi

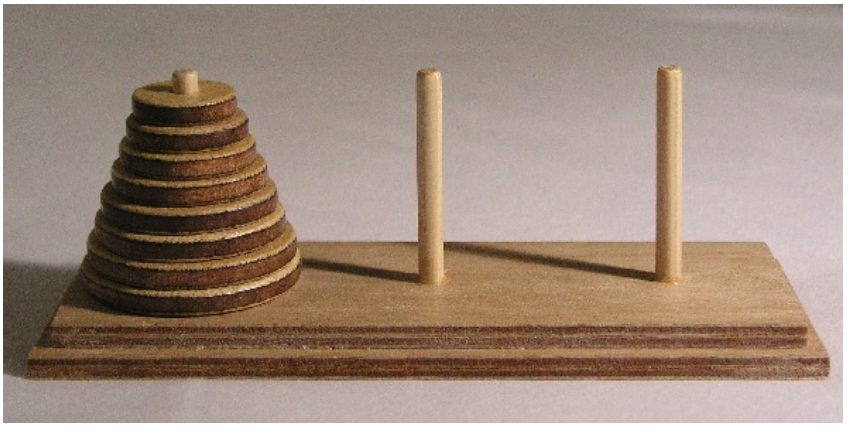


FIGURE 14-9

created by André Karwath – from Wikipedia – used with permission through Creative Commons free-use license

The “Towers of Hanoi” is a interesting puzzle that is often used to teach about recursion.

Its setup is simple:

There are three separate pegs, and a certain number of disks on the first peg. The disks are stacked in order of decreasing size from bottom to top.

The goal is to move all the disks to one of the two empty pegs, and having them end up in the same order as they began in (from smallest to largest, moving from top to bottom).

The three rules are:

Only one disk may be moved at a time.

Only the top-most disk of any peg can be moved.

A larger disk can never be placed on top of a smaller disk.

Solving the puzzle looks something like this (using three spaces instead of pegs):

FIGURE 14-10

created by Wikipedia user Evanherk – From Wikipedia – Used with permission through Creative Commons free-use license

Notice how a larger disk is never placed on top of a smaller disk.

The series of moves that is required to place all the disks in the upright tower configuration is called the solution to the puzzle.

The series of moves that is required to place all the disks in the upright tower configuration is called the solution to the puzzle.

Each move can be represented by a two-element array in JavaScript: [2, 3] means that a disk moves from peg 2 to peg 3.

We can create great music from the solution to this puzzle by choosing three audio clips from the library (one for each peg), and letting the sequence of moves from the correct solution be used to choose from the three audio clips in the same order as how the disks move from peg to peg.

This is very easy to do, because the final solution is just a list of numbers, which we can use directly to "play the clips". We will see this ahead shortly.

In the code example below, we provide a standard recursive solution to this popular puzzle.

The recursive function here is the hanoi_move() function, which solves the puzzle completely using the power of recursion.

The output of hanoi_move() is an array, where each group of two values in the array represents a move from one peg to another. For example, [2, 3, 1, 2] is an array of two moves, one from peg 2 to peg 3, and the following move from peg 1 to peg 2.

Thus all the moves that make up the solution are given in order by the array returned by hanoi_move().

This array of moves in order is used inside of the towers_of_hanoi_string() function as material to convert into a string representation that will choose one of three audio clips (given by the sequence of moves returned by hanoi_move()) according to a steady 8th-note rhythm.

We achieve this by simply using the output of towers_of_hanoi_string() as a beat-string for makeBeat().

We then split this beat-string in two halves, and use the first half to supply the beat-string for the first call to makeBeat(), and then use the second half of the string as the beat-string for a second call to makeBeat(), which uses a different sound for the second half of the musical example.

As in the previous section, listen to the audio example below, then follow the comments in the provided code to understand and learn how to use the towers_of_hanoi_string() function to create interesting music that doesn't repeat, but often makes sense on some higher level. Play around with different values, sounds, substrings and assignments as before, and create your own unique piece in EarSketch by modifying and extending the example below.

audioMedia/3_Towers_of_Hanoi.mp3

```
init();
setTempo(135);

// functions

function hanoi_move(n, frm, dest, via, sequence_of_moves) {
        // parameters: n (number of pegs), the number of the peg you're moving from...
        // the number of the peg you're moving to, the number of the other peg, and the colle
        if (n == 1) {
                // append current move (from-peg, to-peg) to sequence_of_moves array
                sequence_of_moves.push(frm);
                sequence_of_moves.push(dest);
        }
        else {
```

```
                hanoi_move(n-1, frm, via, dest, sequence_of_moves);
                hanoi_move(  1, frm, dest, via, sequence_of_moves);
                hanoi_move(n-1, via, dest, frm, sequence_of_moves);
        }
}

function towers_of_hanoi_string(n) {
        var results = []; // create an array to collect the resulting sequence of h
        // call the recursive function with n, and 0, 1, & 2 as names of the three
        // also include the presently empty "results" array to collect the moves.
        hanoi_move(n, 0, 1, 2, results);
        var beatString = ""; // create an empty string
        // for each element in the results array...
        // convert the peg number to a string, add a "+" to it (for extending a sou
        for (var i = 0; i < results.length; i++) {
                var element = results[i];
                beatString += element.toString() + "+";
        }
        return beatString; // return string of hanoi moves
}

// sounds used in the piece

var drumloop = DUBSTEP_DRUMLOOP_MAIN_001;
var soundList1 = [EIGHT_BIT_ATARI_LEAD_001, DUBSTEP_BASS_WOBBLE_001, ELECTRO_DRUM_MA
var soundList2 = [ EIGHT_BIT_ATARI_SFX_001, DUBSTEP_BASS_WOBBLE_015, ELECTRO_DRUM_MA

// code to generate piece

fitMedia(drumloop, 1, 1, 9);
// call makeBeat with the towers_of_hanoi_string function supplying the beat-string
// Try using different numbers of rings for n in towers_of_hanoi_string(n) (2 <= n <

var a = towers_of_hanoi_string(5);

makeBeat(soundList1, 2, 1, towers_of_hanoi_string(5).slice(0, 64));
makeBeat(soundList2, 2, 5, towers_of_hanoi_string(5).slice(64, 128));

finish();
```

The EarSketch API 15

Click here to open the EarSketch API.

Every Effect Explained in Detail 16

BANDPASS

audioMedia/bandpass.mp3

BANDPASS is a filter that only passes (lets through) an adjustable-sized band of frequencies. All other frequencies are suppressed. This can be used for special-effect sounds such as the "megaphone" sound popular in some modern rock music, or a telephone or small speaker sound, by greatly limiting the frequency range of the original sound (by setting BANDPASS_WIDTH to a relatively small value). By using a wider frequency range (setting BANDPASS_WIDTH to a higher value), sounds that appear "too big" for a mix may be made to sound a little smaller so that they blend better with other sounds in the mix.

Parameter	Description	Default Value	minValue	maxValue	Example
BANDPASS_FREQ	The center frequency (in Hz) of the window of frequencies to pass through.	800.0	20.0	20000.0	`setEffect(1, BANDPASS, BANDPASS_FREQ, 200)`
BANDPASS_WIDTH	The width (in Hz) of the window of frequencies to let through.	0.5	0.0	1.0	`setEffect(1, BANDPASS, BANDPASS_WIDTH, 0.3)`
MIX	The percentage of the effected sound (wet) that is mixed with the original sound (dry). At its minimum value (0.0),	0.1	0.0	1.0	`setEffect(1, BANDPASS, MIX, 0.3)`

	no effect can be heard. At its maximum value (1.0), none of the original sound is heard - it is all effect.				
BYPASS	whether the effect is "on" (1.0) or "off" (0.0). If the bypass of an effect is "on" (1.0), that means the audio going into the effect passes through, and comes out unaffected. Note that unlike other effect name/parameter pairs, the only valid values for BYPASS are 0.0 and 1.0.	0.0	0.0	1.0	`setEffect(1, BANDPASS, BYPASS, 0.0)`

CHORUS

audioMedia/chorus1.mp3

`CHORUS` creates various copies of the original sound which get varied slightly in pitch and time, and mixed back in to the sound, creating an ensemble-like effect of many voices playing together. At extreme values of parameter settings, artificial "robot-like" sounds can be heard.

Parameter	Description	Default Value	minValue	maxValue	Example
CHORUS_LENGTH	The length of time (in ms) from the original sound within which the chorus effect is activated.	15.0	1.0	250.0	`setEffect(1, CHORUS, CHORUS_LENGTH, 53.0)`
CHORUS_NUMVOICES	The number of copies of the original sound that is used. Larger values	1.0	1.0	8.0	`setEffect(1, CHORUS, CHORUS_NUMVOICES, 4.0)`

	create a bigger ensemble-like effect.				
CHORUS_RATE	The rate (in Hz) which the pitch cycles or "wobbles" at. Lower values create smoothly-cycling sounds, while higher values create more wobbly-sounding effects.	0.5	0.1	16.0	`setEffect(1, CHORUS, CHORUS_RATE, 3.0)`
MIX	The percentage of the effected sound (wet) that is mixed with the original sound (dry). At its minimum value (0.0), no effect can be heard. At its maximum value (1.0), none of the original sound is heard – it is all effect.	1.0	0.0	1.0	`setEffect(1, CHORUS, MIX, 0.5)`
CHORUS_MOD	The depth of the pitch wobbling (i.e. how much pitch cycling is used). Low settings create a more natural sound, while higher settings create a more artificial-like sound.	0.7	0.0	1.0	`setEffect(1, CHORUS, CHORUS_MOD, 0.4)`

COMPRESSOR

audioMedia/compressor.mp3

`COMPRESSOR` is a basic two-parameter compressor, which reduces the volume of the loudest sounds of the effected track, while amplifying the volume of its quietest sounds. This creates a narrower dynamic range from the original sound, and is often used to maximize the punch of the original sound, while reducing the potential for noise to be added later.

Parameter	Description	Default Value	minValue	maxValue	Example
COMPRESSOR_THRESHOLD	The amplitude (volume) level (in dB) above which the compressor starts to reduce volume.	0.0	-12.0	1.0	`setEffect(1, COMPRESSOR, COMPRESSOR_THRESHOLD, -4.0)`
COMPRESSOR_RATIO	The amount of specified gain reduction. A ratio of 3:1 means that if the original sound is 3 dB over the threshold, then the effected sound will be 1 dB over the threshold.	1.0	1.0	20.0	`setEffect(1, COMPRESSOR, COMPRESSOR_RATIO, 35.0)`
BYPASS	Whether the effect is “on” (1.0) or “off” (0.0). If the bypass of an effect is “on” (1.0), that means the audio going into the effect passes through, and comes out unaffected. Note that unlike other effectname/parameter pairs, the only valid values for BYPASS are 0.0 and 1.0.	0.0	0.0	1.0	`setEffect(1, COMPRESSOR, BYPASS, 1.0)`

DELAY

audioMedia/delay2.mp3

DELAY creates a repeated echo-like delay of the original sound. A delay effectplays back the original audio as well as a delayed, quieter version of the original that sounds like an echo. After the first echo it plays an echo of the echo (even quieter), then an echo of the echo of the echo (still quieter), and so on until the echo dies out to nothing. With the delay effect, we can control how much time passes between each echo (delay time). If we set the delay time to match the length of a beat, we can create rhythmic effects with delay.

Parameter	Description	Default Value	minValue	maxValue	Example
DELAY_TIME	The time amount in milliseconds (ms) that the original track is delayed, and the time between successive repeats of the delay.	300.0	0.0	4000.0	`setEffect(1, DELAY, DELAY_TIME, 1200.0)`
DELAY_FEEDBACK	The relative amount of repeats that the delay generates. Higher values create more “echoes”. Be careful of applying “too much” feedback!	-3.0	-120.0	-1.0	`setEffect(1, DELAY, DELAY_FEEDBACK, -20.0)`
MIX	The percentage of the effected sound (wet) that is mixed with the original sound (dry). At its minimum value (0.0), no effect can be heard. At its maximum value (1.0), none of the original sound is heard - it is all effect.	1.0	0.0	1.0	`setEffect(1, DELAY, MIX, 0.4)`
BYPASS	Whether the effect is “on” (1.0) or “off” (0.0). If the bypass of an effect is “on” (1.0), that means the audio going into the effect passes through, and comes out unaffected. Note that unlike other effect name/parameter pairs, the only valid values for BYPASS are 0.0 and 1.0.	0.0	0.0	1.0	`setEffect(1, DELAY, BYPASS, 1.0)`

DISTORTION
audioMedia/distortion2.mp3

DISTORTION creates a “dirty” or “fuzzy” sound by overdriving the original sound. This compresses or clips the sound wave, adding overtones (higher frequencies related to the original sound). It is common to distort an electric guitar sound by “overdriving” the guitar amplifier. Modern music sometimes uses distortion to add a grungy or gritty effect or feel to the composition.

Parameter	Description	Default Value	minValue	maxValue	Example
DISTO_GAIN	The amount of overdrive of the original sound.	20.0	0.0	50.0	`setEffect(1, DISTORTION, DISTO_GAIN, 25.0)`
MIX	The percentage of the effected sound (wet) that is mixed with the original sound (dry). At its minimum value (0.0), no effect can be heard. At its maximum value (1.0), none of the original sound is heard - it is all effect.	1.0	0.0	1.0	`setEffect(1, DISTORTION, MIX, 0.4)`
BYPASS	Whether the effect is “on” (1.0) or “off” (0.0). If the bypass of an effect is “on” (1.0), that means the audio going into the effect passes through, and comes out unaffected. Note that unlike other effectname/parameter pairs, the only valid values for BYPASS are 0.0 and 1.0.	0.0	0.0	1.0	`setEffect(1, DISTORTION, BYPASS, 1.0)`

EQ3BAND

audioMedia/eq3band.mp3

EQ3BAND is a three-band equalizer used for simple EQ tasks. An equalizer is used to adjust the volume of separate ranges of frequencies within an audio track. This particular effect can be used to adjust the volume of three ranges (“bands”) of frequency content, namely bass, midrange, and treble (low, mid,

high), where the upper border (`EQ3BAND_LOWFREQ`) of the low range and the center frequency of the mid range (`EQ3BAND_MIDFREQ`) may be set by the user.

Parameter	Description	Default Value	minValue	maxValue	Example
EQ3BAND_LOWGAIN	The gain (in dB) of the low range of frequencies of the EQ. Negative values lower the volume of the low frequencies, while positive values boost them.	0.0	-24.0	18.0	`setEffect(1, EQ3BAND, EQ3BAND_LOWGAIN, 5.3)`
EQ3BAND_LOWFREQ	Specifies the highest frequency (in Hz) of the low range.	200.0	20.0	20000.0	`setEffect(1, EQ3BAND, EQ3BAND_LOWFREQ, 700.0)`
EQ3BAND_MIDGAIN	The gain (in dB) of the mid range of frequencies of the EQ. Negative values lower the volume of the mid frequencies, while positive values boost them.	0.0	-24.0	18.0	`setEffect(1, EQ3BAND, EQ3BAND_MIDGAIN, -15.0)`
EQ3BAND_MIDFREQ	Specifies the center frequency (in Hz) of the mid range.	2000.0	20.0	20000.0	`setEffect(1, EQ3BAND, EQ3BAND_MIDFREQ, 1200.0)`
EQ3BAND_HIGHGAIN	The gain (in dB) of the high range of frequencies of the EQ. Negative val-	0.0	-24.0	18.0	`setEffect(1, EQ3BAND, EQ3BAND_HIGHGAIN, -15.0)`

	ues lower the volume of the high frequencies, while positive values boost them.				
EQ3BAND_HIGHFREQ	Specifies the cutoff frequency (in Hz) of the high range.	2000.0	20.0	20000.0	`setEffect(1, EQ3BAND, EQ3BAND_HIGHFREQ, 8000.0)`
MIX	The percentage of the effected sound (wet) that is mixed with the original sound (dry). At its minimum value (0.0), no effectcan be heard. At its maximum value (1.0), none of the original sound is heard - it is all effect.	1.0	0.0	1.0	`setEffect(1, EQ3BAND, MIX, 0.4)`
BYPASS	Whether the effect is “on” (1.0) or “off” (0.0). If the bypass of an effect is “on” (1.0), that means the audio going into the effectpasses through, and comes out unaffected. Note that unlike other effect name/parameter pairs, the only valid values for BY-	0.0	0.0	1.0	`setEffect(1, EQ3BAND, BYPASS, 1.0)`

	PASS are 0.0 and 1.0.				

FILTER

audioMedia/filter.mp3

`FILTER` is a standard low-pass filter with resonance. A low-pass filter effect allows low frequency audio to pass through unchanged, while lowering the volume of the higher frequencies above a cutoff frequency (the `FILTER_FREQ` parameter). This gives the audio a "darker" sound.

Parameter	Description	Default Value	minValue	maxValue	Example
FILTER_FREQ	The cutoff frequency (Hz), which means that all frequencies higher than this value are rolled-off (become lower and lower in volume the higher they are from this value).	1000.0	20.0	20000.0	`setEffect(1, FILTER, FILTER_FREQ, 3000.0)`
FILTER_RESONANCE	The amount of amplification of a narrow band of frequencies around the current `FILTER_FREQ` level. This causes the frequencies around the current `FILTER_FREQ` level to ring out more, to sound more "resonant". It effectively creates a more vibrant, ringing sound around the cutoff frequency (`FILTER_FREQ`). Higher values of resonance will make the filter "sharper" around the `FILTER_FREQ`, which accentuates the frequencies closest to the cutoff frequency. This is a subtle parameter that	0.8	0.0	1.0	`setEffect(1, FILTER, FILTER_RESONANCE, 0.0, 1.0, 0.9, 3.0)`

	helps fine-tune the sound of the filter.				
MIX	The percentage of the effected sound (wet) that is mixed with the original sound (dry). At its minimum value (0.0), no effect can be heard. At its maximum value (1.0), none of the original sound is heard - it is all effect.	1.0	0.0	1.0	`setEffect(1, FILTER, MIX, 0.4)`
BYPASS	Whether the effect is "on" (1.0) or "off" (0.0). If the bypass of an effectis "on" (1.0), that means the audio going into the effect passes through, and comes out unaffected. Note that unlike other effect name/parameter pairs, the only valid values for BYPASS are 0.0 and 1.0.	0.0	0.0	1.0	`setEffect(1, FILTER, BYPASS, 1.0)`

FLANGER

audioMedia/flanger.mp3

FLANGER is similar to a chorus effect, where various copies of the original sound are created which get varied slightly in pitch and time, and mixed back in to the sound. A flanger, however, uses a much finer range of time values, which creates an evolving "whoosh" like sound. At extreme values of parameter settings, more artificial "robot-like" sounds can be heard.

Parameter	Description	Default Value	minValue	maxValue	Example
FLANGER_LENGTH	The length of delay time (in ms) from the original sound within which the flanger effectis activated.	6.0	0.0	200.0	`setEffect(1, FLANGER, FLANGER_LENGTH, 23.0)`

FLANGER_FEEDBACK	The amount (in dB) that the effected sound is "fed back" into the effect. Higher values create more artificial-like sounds.	-50.0	-80.0	-1.0	`setEffect(1, FLANGER, FLANGER_FEEDBACK, -80.0)`
FLANGER_RATE	The rate (in Hz) which the pitch cycles or "whooshes" at. Lower values create more smoothly-cycling sounds, while higher values create more whooshing-sounding effects and sonic artifacts.	0.6	0.001	100.0	`setEffect(1, FLANGER, FLANGER_RATE, 45.0)`
MIX	The percentage of the effected sound (wet) that is mixed with the original sound (dry). At its minimum value (0.0), no effect can be heard. At its maximum value (1.0), none of the original sound is heard - it is all effect.	1.0	0.0	1.0	`setEffect(1, FLANGER, MIX, 0.4)`
BYPASS	Whether the effect is "on" (1.0) or "off" (0.0). If the bypass of an effect is "on" (1.0), that means the audio going into the effect passes through, and comes out unaffected. Note that unlike other effect name/parameter pairs, the only valid values for BYPASS are 0.0 and 1.0.	0.0	0.0	1.0	`setEffect(1, FLANGER, BYPASS, 1.0)`

PAN

audioMedia/pan2.mp3

PAN affects the audio mix between the left and right channels. For example, if you were wearing headphones, changing the panning would affect whether you heard something in the left ear or the right.

Parameter	Description	Default Value	minValue	maxValue	Example
LEFT_RIGHT	Specifies the left/right location of the original sound within the stereo field (0.0 is center, -100.0 is fully left, 100.0 is fully right).	0.0	-100.0	100.0	`setEffect(1, PAN, LEFT_RIGHT, -50.0)`
BYPASS	Whether the effect is "on" (1.0) or "off" (0.0). If the bypass of an effect is "on" (1.0), that means the audio going into the effect passes through, and comes out unaffected. Note that unlike other effect name/parameter pairs, the only valid values for BYPASS are 0.0 and 1.0.	0.0	0.0	1.0	`setEffect(1, PAN, BYPASS, 1.0)`

PHASER

audioMedia/phaser.mp3

PHASER is a sweeping-sounding effect which creates a copy of the original sound over a specified range of frequencies. This effected copy is then delayed very slightly and played against the original sound while changing its slight delay time gently back and forth. This causes some of the copied frequencies to temporarily cancel each other out by going "in and out of phase" with each other, thus creating a sweeping effect.

Parameter	Description	Default Value	minValue	maxValue	Example
PHASER_RATE	The rate (in Hz) that the slight delay time changes back and forth. Lower values create more smoothly-cycling	0.5	0.0	10.0	`setEffect(1, PHASER, PHASER_RATE, 3.0)`

	sounds, while higher values create more robotic-sounding effects and sonic artifacts.				
PHASER_RANGEMIN	The low value (in Hz) of the affected frequency range.	440.0	40.0	20000.0	`setEffect(1, PHASER, PHASER_RANGEMIN, 880.0)`
PHASER_RANGEMAX	The high value (in Hz) of the affected frequency range.	1600.0	40.0	20000.0	`setEffect(1, PHASER, PHASER_RANGEMAX, 1700.0)`
PHASER_FEEDBACK	The amount that the effected sound is “fed back” into the effect. Higher values create more artificial-like sounds.	-3.0	-120.0	-1.0	`setEffect(1, PHASER, PHASER_FEEDBACK, -1.0)`
MIX	The percentage of the effected sound (wet) that is mixed with the original sound (dry). At its minimum value (0.0), no effect can be heard. At its maximum value (1.0), none of the original sound is heard - it is all effect.	1.0	0.0	1.0	`setEffect(1, PHASER, MIX, 0.4)`
BYPASS	Whether the effect is “on” (1.0) or “off” (0.0). If the bypass of an effect is “on” (1.0), that means the audio going into the effect passes through, and comes out unaffected. Note that unlike other effect name/parameter pairs, the only valid	0.0	0.0	1.0	`setEffect(1, PHASER, BYPASS, 1.0)`

	values for BYPASS are 0.0 and 1.0.				

PITCHSHIFT

audioMedia/pitchshift1.mp3

`PITCHSHIFT` simply lowers or raises the sound by a specific pitch interval (`PITCHSHIFT_SHIFT`). It can be useful in helping multiple sound files sound better together or, contrastingly, to add a little bit of dissonance, if desired.

Parameter	Description	Default Value	minValue	maxValue	Example
PITCHSHIFT_SHIFT	Specifies the amount to adjust the pitch of the original sound in semitones (and fractions of a semitone, given by values after the decimal point). 12 semitones equal 1 octave.	0.0	-12.0	12.0	`setEffect(1, PITCHSHIFT, PITCHSHIFT_SHIFT, 4.0)`
BYPASS	Whether the effect is “on” (1.0) or “off” (0.0). If the bypass of an effect is “on” (1.0), that means the audio going into the effect passes through, and comes out unaffected. Note that unlike other effect name/parameter pairs, the only valid values for BYPASS are 0.0 and 1.0.	0.0	0.0	1.0	`setEffect(1, PITCHSHIFT, BYPASS, 1.0)`

REVERB

audioMedia/reverb.mp3

REVERB adds a slowly decaying ambiance to the source signal, which is similar to DELAY but is often much denser and richer. It is widely used for audio mixing and spatialization.

Parameter	Description	Default Value	min-Value	max-Value	Example
REVERB_TIME	The decaying time of the ambiance in milliseconds (ms). When modulating REVERB_TIME over time using automation curve, due to the nature of convolution-based reverb, the value is updated only at every quarter note (time=0.25) in a "stair-case" manner from the starting point of the automation. (You will, however, hardly notice that.)	1500.0	100.0	4000.0	`setEffect(1, REVERB, REVERB_TIME, 1000.0)`
REVERB_DAMPFREQ	The cutoff frequency (in Hz) of the lowpass filter applied to the ambiance. The lower the value, the darker the reverberation will sound.	10000.0	200.0	18000.0	`setEffect(1, REVERB, REVERB_DAMPFREQ, 1500.0)`
MIX	The percentage of the effected sound (wet) that is mixed with the original sound (dry). At its minimum value (0.0), no effect can be heard. At its maximum value (1.0), none of the original sound is heard - it is all effect.	0.3	0.0	1.0	`setEffect(1, REVERB, MIX, 0.4)`

BYPASS	Whether the effect is “on” (1.0) or “off” (0.0). If the bypass of an effectis “on” (1.0), that means the audio going into the effect passes through, and comes out unaffected. Note that unlike other effect name/parameter pairs, the only valid values for BYPASS are 0.0 and 1.0.	0.0	0.0	1.0	`setEffect(1, REVERB, BYPASS, 1.0)`

RINGMOD

audioMedia/ringmod.mp3

`RINGMOD` multiplies the signals from two sounds together: your original sound and a pure sine wave (that sounds like a tuning fork). The effect of this multiplication sounds different at every frequency of the original sound, which creates a completely artificial-sounding result, as this type of sound could never occur naturally. Some parameter settings for this effect will likely produce recognizable-sounding effects similar to ones used in old science-fiction movies. It is useful experimenting with since there are a wide range of sounds that can be generated from your original sound.

Parameter	Description	Default Value	minValue	maxValue	Example
RINGMOD_MODFREQ	The frequency (in Hz) of the sine wave oscillator that is being multiplied into your original sound.	40.0	0.0	100.0	`setEffect(1, RINGMOD, RINGMOD_MODFREQ, 70.0)`
RINGMOD_FEEDBACK	The amount of effected sound that is fed-back into the effect. High values create more robotic-type sounds and sonic artifacts.	0.0	0.0	100.0	`setEffect(1, RINGMOD, RING-`

					`MOD_FEED-BACK, 30.0)`
MIX	The percentage of the effected sound (wet) that is mixed with the original sound (dry). At its minimum value (0.0), no effect can be heard. At its maximum value (1.0), none of the original sound is heard - it is all effect.	1.0	0.0	1.0	`setEf-fect(1, RINGMOD, MIX, 0.4)`
BYPASS	Whether the effect is “on” (1.0) or “off” (0.0). If the bypass of an effectis “on” (1.0), that means the audio going into the effect passes through, and comes out unaffected. Note that unlike other effect name/parameter pairs, the only valid values for BYPASS are 0.0 and 1.0.	0.0	0.0	1.0	`setEf-fect(1, RINGMOD, BYPASS, 1.0)`

TREMOLO

audioMedia/tremolo.mp3

`TREMOLO` quickly changes the volume of the original sound back and forth from its original value towards silence, resulting in a wobbling-sounding effect.

Parameter	Description	De-fault Val-ue	min-Val-ue	max-Value	Example
`TREMO-LO_FREQ`	The rate (in Hz) that the volume is changed back and forth.	4.0	0.0	100.0	`setEffect(1, TREMOLO, TREMO-LO_FREQ, 10.0)`
`TREMO-LO_AMOUNT`	The amount (in dB) that the volume changes back and forth over during each cycle.	-6.0	-60.0	0.0	`setEffect(1, TREMOLO, TREMO-LO_AMOUNT, -40.0)`

MIX	The percentage of the effected sound (wet) that is mixed with the original sound (dry). At its minimum value (0.0), no effect can be heard. At its maximum value (1.0), none of the original sound is heard - it is all effect.	1.0	0.0	1.0	`setEffect(1, TREMOLO, MIX, 0.4)`
BYPASS	Whether the effect is “on” (1.0) or “off” (0.0). If the bypass of an effect is “on” (1.0), that means the audio going into the effectpasses through, and comes out unaffected. Note that unlike other effect name/parameter pairs, the only valid values for BYPASS are 0.0 and 1.0.	0.0	0.0	1.0	`setEffect(1, TREMOLO, BYPASS, 1.0)`

VOLUME

audioMedia/volume2.mp3

`VOLUME` allows you to change the volume of an audio clip.

Parameter	Description	Default Value	minValue	maxValue	Example
GAIN	Specifies the output volume level of the original sound.	0.0	-60.0	12.0	`setEffect(1, VOLUME, GAIN, -5.0)`
BYPASS	Whether the effect is “on” (1.0) or “off” (0.0). If the bypass of an effect is “on” (1.0), that means the audio	0.0	0.0	1.0	`setEffect(1, VOLUME, BYPASS, 1.0)`

	going into the effect passes through, and comes out unaffected. Note that unlike other effect name/parameter pairs, the only valid values for BYPASS are 0.0 and 1.0.				

WAH

audioMedia/wah.mp3

WAH is a resonant bandpass filter (see BANDPASS effect) that creates a "wah-wah" pedal sound when changed over time using envelopes in the setEffect() function.

Parameter	Description	Default Value	minValue	maxValue	Example
WAH_POSITION	The center frequency of the boosted fixed-width frequency range.	0.0	0.0	1.0	`setEffect(1, WAH, WAH_POSITION, 0.3)`
MIX	The percentage of the effected sound (wet) that is mixed with the original sound (dry). At its minimum value (0.0), no effect can be heard. At its maximum value (1.0), none of the original sound is heard - it is all effect.	1.0	0.0	1.0	`setEffect(1, WAH, MIX, 0.4)`
BYPASS	Whether the effect is "on" (1.0) or "off" (0.0). If the bypass of an effect is "on" (1.0), that means the audio going into the effect passes through, and comes out unaffected. Note that unlike other effect name/parameter pairs, the only valid values for BYPASS are 0.0 and 1.0.	0.0	0.0	1.0	`setEffect(1, WAH, BYPASS, 1.0)`

17 Analysis Features

This document details each of the Analysis features that can be used with the analysis functions in the EarSketch API (`analyze()`, `analyzeForTime()`, `analyzeTrack()`, and `analyzeTrackForTime()`). Each of these features can be used by using the appropriate constant (which is specified with each description). These features are possible ways to determine differences in audio samples. This difference, or timbre, is how humans are able to tell the difference between instruments. For example, it's possible to distinguish between playing a C note on a piano, from a C note on a trombone. Each of these measurements returns a value between 0.0 and 1.0, and it is encouraged to try out different features if one does not work for your particular situation.

The examples below only use the maximum and minimum examples in the `YOUNGGURU__Y04_88_BPM_F_MINOR` folder. For some of the examples, there might be other sound files the exhibit a higher or lower value than the example listed.

Spectral Centroid

constant – `SPECTRAL_CENTROID`

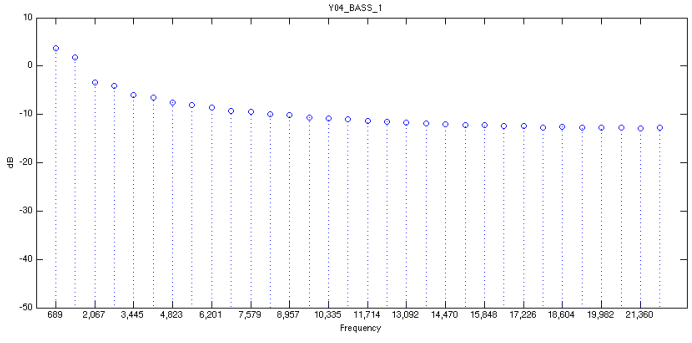
description – The average frequency of the audio. It describes the brightness of the sound. An instrument like a tuba or bass guitar will have a lower Spectral Centroid than an instrument like a violin. From the graphs below you will see that a spectrum with a lower Spectral Centroid value has more frequency content in the lower part of the spectrum, while a higher value is more spread out. In these examples, you can see that the `Y04_BASS_1` audio file's lower frequencies weight the Spectral Centroid value towards a lower value.

For more information, see this Wikipedia article: **http://en.wikipedia.org/wiki/Spectral_centroid**

Spectrum of a low Spectral Centroid value:
audioMedia/Y04-bass-1.mp3

FIGURE 17-1

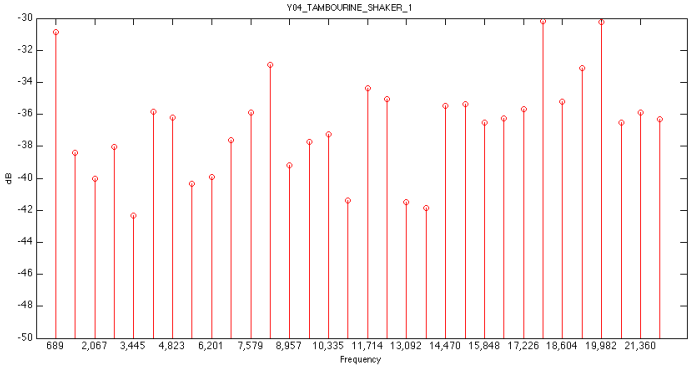
Low Spectral Centroid



Spectrum of a high Spectral Centroid value: audioMedia/Y04-Tambourine-Shaker-1.mp3

FIGURE 17-2

High Spectral Centroid



RMS Amplitude

constant – `RMS_AMPLITUDE`

description – The Root Mean Square amplitude of the audio. This is a more convenient representation of the amplitude (or loudness) of an audio signal. It is important to be careful of the sounds you analyze with this feature, because it's an average amplitude measurement. If an audio file, such as Y04_TAMBOUR-

INE_SHAKER_1 has only a small amount of non-silence in it, then the average amplitude will be small in comparison to its peak amplitude.

In the examples below, the red line indicates the RMS Amplitude value detected using the `analyze()` function. If you are more interested in the amplitude at specific times, `analyzeForTime()` might be a better function to use.

There are many ways to represent the amplitude of an audio signal, see this Wikipedia article for more information: **http://en.wikipedia.org/wiki/Amplitude**

Time vs. Amplitude plot of a low RMS Amplitude value:
audioMedia/Y04-Tambourine-Shaker-1.mp3

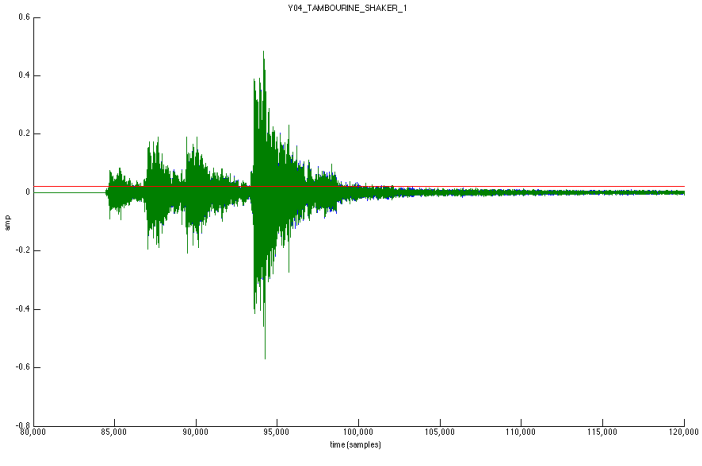


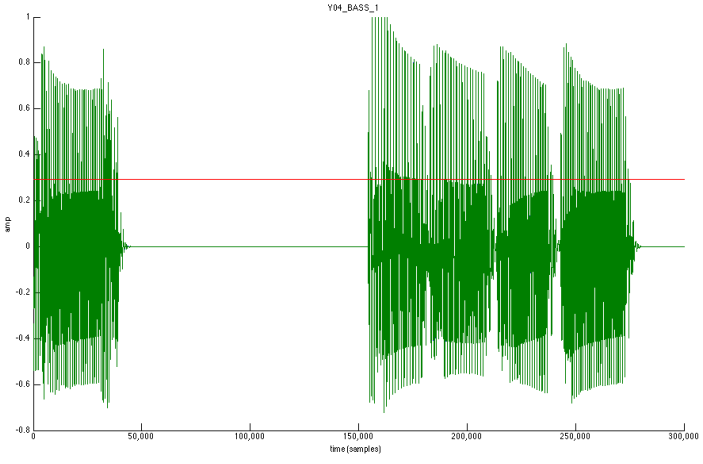
FIGURE 17-3

Low RMS Amplitude

Time vs. Amplitude plot of a high RMS Amplitude value:
audioMedia/Y04-bass-1.mp3

FIGURE 17-4

High RMS Amplitude



Creating Beats with makeBeat 18

The 16 elements of a beat string make up the 16 sixteenth notes found in one measure of 4/4 time. In creating beats with makeBeat() the style, instrument, and role of the beat should be taken into consideration in creating the rhythm pattern. This guide will provide sample rhythm patterns in the style of 4/4 Time, Hip Hop, Funk, Dub Step, and African Drum Ensemble based patterns. This will not represent a complete list of patterns, rather it will act as a guide in identifying the characteristics of percussion beats and provide string example for makeBeat().

The three elements of a percussion line

The drum set or percussion line can be divided into three elements:

1. Bass Drum or Lowest Pitched Drum: Usually provides the "center of beat." The ear will gravitate towards the lowest pitches in a drum pattern to 'center in' on the meter and feel for the music. Bass Drum beats usually emphasize beats 1 and 3 in 4/4 time.
2. Back Beat: This is usually a snare drum, clap, snap, or other mid-pitched percussion instrument. The Back Beats complement the Bass Drum Line, provide syncopation and 'pull', and emphasize beats 2 and 4 in 4/4 time. The Bass Drum and BackBeat work together to provide the bulk of the sense of style and feel in percussion lines.
3. Running 8^{th} or 16^{th} Patterns. These running lines of 8^{th} or 16^{th} notes usually are played with a high pitched metallic instrument such as a Hi Hat, Ride Cymbal, Tambourine, or other non-pitched instrument. The 'running' pattern provides a 'motor' that lays down a foundation for the Bass Drum and Back Beat lines. The running patterns also provide the listener with a clear sense of the 'microbeat' and meter for the music. Changing the timbre within the running pattern (Closed and Open Hi hat for example) can supplement the rhythmic interest of the Back Beat.

At the core level you can visualize the Bass Drum, Back Beat, and Running 8^{th} / 16^{th} in the following manner with makeBeat(). Note: the beat strings in the following tables have spaces in between each character in order to ensure

proper display in a web browser (e.g., "– – – –"). Remove the extraneous spaces when using the strings in your own code.

Bass Drum:

	Beats			
String Pattern	1	2	3	4
"0 + + + – – – – 0 + + + – – – –"	0 + + +	– – – –	0 + + +	– – – –

Back Beat:

	Beats			
String Pattern	1	2	3	4
"– – – – 0 + + + – – – – 0 + + +"	– – – –	0 + + +	– – – –	0 + + +

Running 8th:

	Beats			
String Pattern	1	2	3	4
"0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 +"	0 + 0 +	0 + 0 +	0 + 0 +	0 + 0 +

4/4 Time General Patterns (Typical of "Rock Beat" or straight 4/4 type music)

Bass Drum String Patterns:

bassBeat01 = "0 + + + – – – – 0 + + + – – – –"
bassBeat02 = "0 + + + – – 0 + 0 + + + – – – –"
"0 + + + – – 0 + 0 + + + – – 0 +"
"0 + + + 0 + + + 0 + + + 0 + + +"
"0 + + + 0 + + + 0 + 0 + 0 + + +"
"0 + + + 0 + + + 0 + + + 0 + 0 +"

Back Beat String Patterns:

"– – – – 0 + + + – – – – 0 + + +"
"– – – – 0 + + + – – – – 0 + 0 +"

“– – – – 0 + 0 + – – – – 0 + + +”

“– – – – 0 + + + – 0 + – 0 + + +”

8th Note Running Pattern:

“ 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 +”

Code Example with Fill:

```
// 4/4 Time General Pattern with Fill
// Tempo Best between 110 and 132

init();
setTempo(120);

var bass = OS_LOWTOM01;
var snare = OS_SNARE01;
var hiOpen = OS_OPENHAT01;
var hiClosed = OS_CLOSEDHAT01;
var bassBeat = "0+++--0+0+++----";
var snareBeat = "----0+++-0+-0+++";
var running = "0+0+0+0+0+0+0+0+";
var fillBass = "0+++0+++0+0+0+0+";
var fillSnare = "0+++0+++0+0+0000";

for(var measure = 1; measure < 9; measure++) {

        if(measure % 4 != 0) {
                makeBeat(hiClosed, 1, measure, running);
                makeBeat(snare, 2, measure, snareBeat);
                makeBeat(bass, 3, measure, bassBeat);
        }
        else {
        makeBeat(hiClosed, 1, measure, running);
        makeBeat(snare, 2, measure, fillSnare);
        makeBeat(bass, 3, measure, fillBass);
        }
}

finish();
```

Some Funk and Hip Hop Beats

Hip Hop and Funk both function well at tempos between 84 and 92 beats per minute. If you use a running beat of 8ths, the style will gravitate closer to Hip Hop. A running beat of 16ths will simulate a funk style.

Bass Drum Strings:

funkbassbeat1 = " 0 + 0 + – – – – 0 + 0 + – 0 + +"

funkbassbeat2 = "0 – 0 – – – – – – – 0 – – 0 – –"

Back Beat Strings:

funkbackbeat1 = "– – – – 0 – – 0 – 0 – 0 0 – – –"

funkbackbeat2 = "– – – – 0 – – 0 – 0 – 0 0 – – 0"

'Amen Beat' style strings with using a list to store snare and bass sounds. drumList = [bass, snare]

amenbeat1 = " 0 + 0 + 1 + + 1 – 1 0 0 1 + + 1"

amenbeat2 = "0 + 0 + 1 + + 1 – 1 0 0 – – 1 +"

amenbeat3 = "– 1 0 0 1 + + 1 – 1 0 + – – 1 +"

16th Beat Running using a List to store closed and Open Hi Hat Sounds:

Hats = [closed, open]

sixteenthHL1 = "0000100000001000"

sixteenthHL2 = "0000100101011000"

sixteenthHL3 = "0000100101011100"

sixteenthHL4 = "0000100101011101"

sixteenthHLFill = "0000100101011111"

Funk Beat Example with Fill:

```
// Funk Beat Example
// Best Played at 92 to 100 Beats per minute

init();
setTempo(92);

var funkbassbeat1 = "0+0+----0+0+-0++";
var funkbassbeat2 = "0-0-------0--0--" ;
var funkbackbeat1 = "----0--0-0-00---";
var funkbackbeat2 = "----0--0-0-00--0" ;
var sixteenthHL2 = "0000100101011000";
var sixteenthHLFill = "0000100101011111";

var bass = OS_LOWTOM01;
var snare = OS_SNARE01;
var hiOpen = OS_OPENHAT01;
var hiClosed = OS_CLOSEDHAT01;
var hats = [hiClosed, hiOpen];
```

```
for(var measure = 1; measure < 9; measure++) {
 if(measure % 4 != 0) {
  makeBeat(hats, 1, measure, sixteenthHL2);
  makeBeat(snare, 2, measure, funkbackbeat1);
  makeBeat(bass, 3, measure, funkbassbeat1);
 }
 else {
  makeBeat(hats, 1, measure, sixteenthHLFill);
  makeBeat(snare, 2, measure, funkbackbeat2);
  makeBeat(bass, 3, measure, funkbassbeat2);
 }
}

finish();
```

Amen Beat Example:

```
// Amen Beat Example
// Best played at 82 to 92 Beats per minute

init();
setTempo(88);

var bass = OS_LOWTOM01;
var snare = OS_SNARE01;
var hiOpen = OS_OPENHAT01;
var hiClosed = OS_CLOSEDHAT01;
var hats = [hiClosed, hiOpen];
var bassSnare = [bass, snare];

var amenbeat1 = "0+0+1++1-1001++1";
var amenbeat2 = "0+0+1++1-100--1+";
var amenbeat3 = "-1001++1-10+--1+";
var sixteenth = "0000100000001000";
var sixteenthHL2 = "0000100101011000";

for(var measure = 1; measure < 9; measure += 4) {
 makeBeat(bassSnare, 2, measure, amenbeat1);
 makeBeat(bassSnare, 2, measure+1, amenbeat2);
 makeBeat(bassSnare, 2, measure+2, amenbeat2);
 makeBeat(bassSnare, 2, measure+3, amenbeat3);
}

for(var measure = 1; measure < 9; measure++) {
 makeBeat(hats, 1, measure, sixteenthHL2);
```

```
}

finish();
```

Dub Step Style Beats:

Dub step music usually is played faster than 136 beats per minute with a 'halftime' feel using triplet style rhythms in the Bass Drum and Back Beat. Beats here will simulate the triplet style with a 3-sixteenth, 3-sixteenth, 2-sixteenth pattern. Dub step music also has longer patterns, usually extending across 4 measures, so the different beats are meant to be played in succession. Dub Step music also 'breaks' the Bass on 1 and 3 and the Back Beat on 2 and 4 rules.

Dub Bass Patterns (Played in succession)

dubBass1 = "0 + + + + + + + – – – – – – 0 +"
dubBass2 = "0 + + 0 + + 0 + – – – 0 + + 0 +"
dubBass3 = "0 + + 0 + + 0 + – – – – – – 0 +"
dubBass4 = "0 0 + 0 0 + 0 + – – – – – – – –"

Dub Snare Patterns (This example only plays on measure 4 of the pattern)

dubSnare = "– – – – – – – – – – 0 0 0 + – –"

Dub Clap Patterns:

dubClap = "– – – – – – – – 0 + + + + + + +"
dubClap1 = "– – – – – – – – 0 + + + + + 0 +"

Dub Hat Patterns: (With [closed, open] list)

dubHats1 = "– – – – 0 0 0 + 1 + + + + + + +"
# Should be a triplet on beat 2
dubHats2 = "– – 0 + + 0 + + 1 + + + + + + +"
dubHats3 = "– – – – 0 0 0 + 1 + + + + + + +"
dubHats4 = "– – 0 + + 0 + + 1 + + 0 + + 0 +"

Dub Step Example:

```
// Dub Step Example
// Best played faster than 136 beats per minute

init();
setTempo(140);

var dubBass1 = "0+++++++------0+";
var dubBass2 = "0++0++0+---0++0+";
var dubBass3 = "0++0++0+------0+";
var dubBass4 = "00+00+0+--------";
```

```
var dubSnare = "----------000+--";

// Only Used on measure 4
var dubClap = "--------0+++++++";
var dubClap1 = "--------0+++++0+";
var dubHats1 = "----000+1+++++++";

// Should be a triplet on beat 2
var dubHats2 = "--0++0++1+++++++";
var dubHats3 = "----000+1+++++++";
var dubHats4 = "--0++0++1++0++0+";

var bass = OS_LOWTOM01;
var snare = OS_SNARE01;
var hiOpen = OS_OPENHAT01;
var hiClosed = OS_CLOSEDHAT01;
var hats = [hiClosed, hiOpen];

for(var measure = 1; measure < 9; measure += 4) {
 makeBeat(hats, 1, measure, dubHats1);
 makeBeat(hats, 1, measure+1, dubHats2);
 makeBeat(hats, 1, measure+2, dubHats3);
 makeBeat(hats, 1, measure+3, dubHats4);
 makeBeat(snare, 2, measure, dubClap);
 makeBeat(snare, 2, measure+1, dubClap);
 makeBeat(snare, 2, measure+2, dubClap);
 makeBeat(snare, 2, measure+3, dubClap1);
 makeBeat(snare, 3, measure+3, dubSnare);
 makeBeat(bass, 4, measure, dubBass1);
 makeBeat(bass, 4, measure+1, dubBass2);
 makeBeat(bass, 4, measure+2, dubBass3);
 makeBeat(bass, 4, measure+3, dubBass4);
}

finish();
```

African Style Drumming Patterns

These patterns seek to emulate the style of drumming ensembles and multi-layered percussion music based on African music. The patterns here are adapted from the “Unifix Patterns” as presented on the **Phil Tulga website**. The drum patterns are designed to ‘weave’ in out and each pattern complements the other. These patterns also demonstrate the use of lists.

Unifix Pattern Set 1:

ftBeat = “0 – – 0 1 – – 1 0 – – 0 1 – – 1”
tcBeat = “1 – 1 – 1 1 – 1 – 0 – 0 – 1 1 –”
guiroBeat = “1 – 0 0 1 – 0 0 – 0 – 0 1 – 0 0”
skakerBeat = “1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1”
tubeBeat = “1 – – – 0 – – – 1 – – – 0 – – –”
bottleBeat = “0 – 0 – 1 1 – 0 – 0 – 0 – 1 1 –”

High Life from Nigeria:

ftBeat = “0 – – 0 0 – 1 – 0 – – 0 0 – 1 1”
tcBeat = “0 – – 0 0 – 1 – 0 – – 0 0 – 1 1”
guiroBeat = “0 – – 1 0 – 1 – – 1 – 1 0 – 1 –”
shakerBeat = “1 0 0 1 1 – 0 1 – 1 0 1 1 – 1 0”
tubeBeat = “1 – – 1 1 – 0 – – 1 – 1 – 1 0 –”
bottleBeat = “– 1 – 0 – – – 1 – 0 – – – 1 – 0”

Fanga Beat from Liberia:

ftBeat = “0 – – 1 – 1 1 – 0 – 0 – 1 1 – –”
tcBeat = “0 – – 1 – 1 1 – 0 – 0 – 1 1 – –”
guiroBeat = “0 – 1 – – – – 0 – – 1 1 1 – – 1”
shakerBeat = “1 0 0 1 0 1 0 1 1 0 1 0 1 0 0 1”
tubeBeat = “0 – – 0 – – – – 1 – 1 – – – – 1”
bottleBeat = “– – – 1 – 1 1 – 0 – 0 – 1 1 – 0”

From Ghana:

ftBeat = “0 0 0 1 – 1 0 – 0 0 0 1 – 1 0 –”
tcBeat = “0 – – 1 – 1 0 – 0 – – 1 – 1 0 –”
guiroBeat = “0 – – 1 0 – 1 – – 1 – 1 0 – 1 –”
shakerBeat = “1 0 0 1 – 1 0 1 – 1 0 1 1 0 0 1”
tubeBeat = “0 – – 1 – – 0 – – 1 – – 1 – – 0”
bottleBeat = “0 – 0 – – – – – 0 0 – 1 – – – 0”

Example of Unifix Patterns

```
// Example of Unifix Patterns
// Best played between 92 and 110 (but can be faster)

init();

setTempo(100);

var fractionTubes = [HOUSE_BREAKBEAT_020, HIPHOP_TRAPHOP_BEAT_007];
var tinCanDrum = [OS_COWBELL01, OS_COWBELL02];
var guiro = [ELECTRO_DRUM_MAIN_BEAT_004, ELECTRO_DRUM_MAIN_BEAT_007];
var shaker = [OS_OPENHAT02, OS_OPENHAT03];
var tubeDrums = [OS_CLAP01, OS_CLAP02];
var waterBottles = [OS_SNARE01, OS_OPENHAT06];

var uniList = [fractionTubes, tinCanDrum, guiro, shaker, tubeDrums, waterBottles];
```

```
// From Ghana

var ftBeat = "0001-10-0001-10-";
var tcBeat = "0--1-10-0--1-10-";
var guiroBeat = "0--10-1--1-10-1-";
var shakerBeat = "1001-101-1011001";
var tubeBeat = "0--1--0--1--1--0";
var bottleBeat = "0-0-----00-1---0";

var ghanaList = [ftBeat, tcBeat, guiroBeat, shakerBeat, tubeBeat, bottleBeat];

for(var measure = 1; measure < 9; measure++) {
    for(var i = 0; i < ghanaList.length; i++) {
        var track = i + 1;
        makeBeat(uniList[i], track, measure, ghanaList[i]);
    }
}

finish();
```

Additional Examples 19

Effect Case Studies

Case Study 1: Volume

Crescendo and Decrescendo with a Keyboard

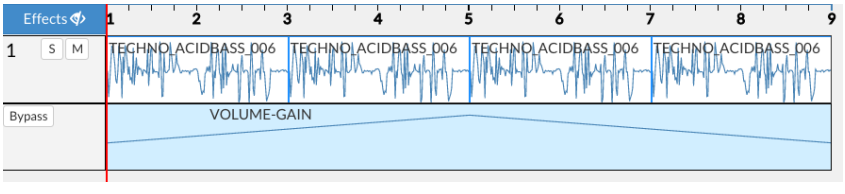


FIGURE 19-1

Crescendo is the musical term for increasing volume (amplitude) over time. Decrescendo means to decrease volume (amplitude) over time. Keyboard players often use a volume pedal to control the intensity of the sound while playing. This example shows a keyboard increasing and decreasing volume over 8 measures:

The following code generates this example:

```
// Keyboard Volume

init();

var keyboard = TECHNO_ACIDBASS_006;

fitMedia(keyboard, 1, 1, 9);

// Crescendo - Volume Pedal pressed
```

```
setEffect(1, VOLUME, GAIN, -30, 1, 5, 5);

//Decrescendo - Volume Pedal released
setEffect(1, VOLUME, GAIN, 5, 5, -30, 9);

finish();
```

Note about the Volume Effect:

The parameters for the Volume Effect range from -60 to 12 with 0 being "unity" or standard volume. Why do we use this scale? Examine this image of a volume slider from a software audio mixer:

FIGURE 19-2

The numbers indicate a logarithmic scale that increases the order of magnitude as the sound grows stronger. This measures decibels which is the same as the richter scale for earthquakes. That is why "0 dB" represents the "center" for volume. The "0 dB" value indicates that there is no change in volume from the original track. Positive numbers increase the volume, while negative numbers decrease the volume. Note that for every increase by "10 dB", the loudness of the sound is doubled. Values between -30 and about 5 work best for the volume effect. Note that a value of -60 means silence.

Case Study 2: Distortion

Guitar Distortion

Guitarists use a foot pedal to control the distortion level in real time during performance. We can simulate this with EarSketch by using a for loop to increase and decrease distortion over a given segment of the measure. Consider the following sample:

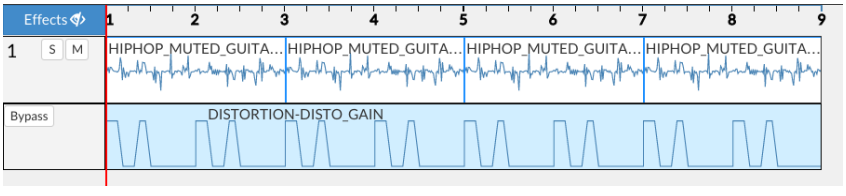


FIGURE 19-3

Here, a guitar player emphasizes beats 1 and 1.375 with an increase in distortion. The slope up and down of the "peaks" indicate where the pedal is pressed and released.

Using EarSketch, the following code generates the above example:

```
//Guitar Distortion

init();

fitMedia(HIPHOP_MUTED_GUITAR_003, 1, 1, 9);

var distortionValues = [0, 40];

var distortionString = "1+-0+-1-0+++++-";

for(var measure = 1; measure < 9; measure++) {
        rhythmEffects(1, DISTORTION, DISTO_GAIN, distortionValues, measure, distortionString);
}

finish();
```

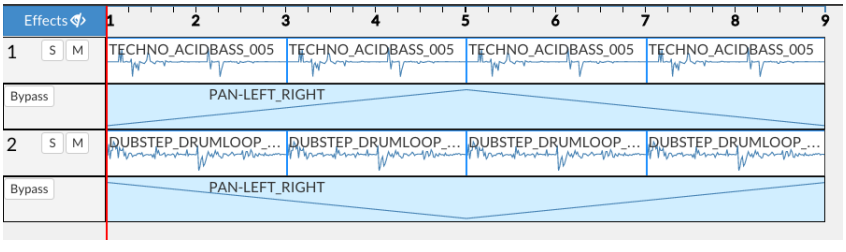
Case Study 3: Panning 1

Left and Right Stereo

The “Pan” effect directs the sound to the “Left” or “Right” speakers in a stereo system. Sweeping music from the left side to the right or from the right side to the left gives the listener a strong sense of space as they perceive the sound coming from different directions.

The following example shows two tracks panning back and forth, starting from opposite sides. The bass track pans from left to right then back again to the left, while the drum track pans from right to left then back again to the right:

FIGURE 19-4



The following code generates the example:

```
// Pan Demonstration

init();
setTempo(140);

var bass = TECHNO_ACIDBASS_005;
var drums = DUBSTEP_DRUMLOOP_MAIN_007;

fitMedia(bass, 1, 1, 9);
fitMedia(drums, 2, 1, 9);

var start = 1;
var end = 5;
var leftside = -100;
var rightside = 100;

setEffect(1, PAN, LEFT_RIGHT, leftside, start, rightside, end);
setEffect(2, PAN, LEFT_RIGHT, rightside, start, leftside, end);
```

```
start = 5;
end = 9;

setEffect(1, PAN, LEFT_RIGHT, rightside, start, leftside, end);
setEffect(2, PAN, LEFT_RIGHT, leftside, start, rightside, end);

finish();
```

Note the use of variables in lines 12 to 15 to define the start, end, left, and right sides.

Case Study 4: Panning 2

Using the pan effect on one track

You can also rapidly change sides on one track to give the sense of more than one instrument playing. Using the same flute and drum mix, note that the pan effect is used to “switch sides” for the flute track:

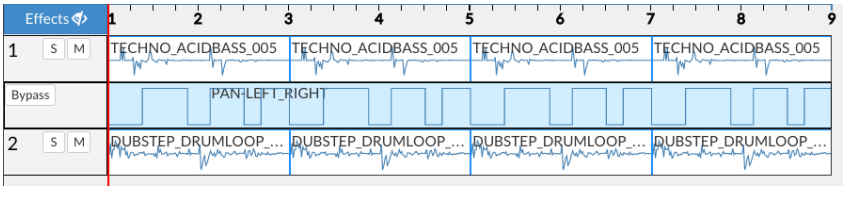


FIGURE 19-5

Code:

```
// Pan Demonstration #2

init();
setTempo(140);

var bass = TECHNO_ACIDBASS_005;
var drums = DUBSTEP_DRUMLOOP_MAIN_007;

fitMedia(bass, 1, 1, 9);
fitMedia(drums, 2, 1, 9);
```

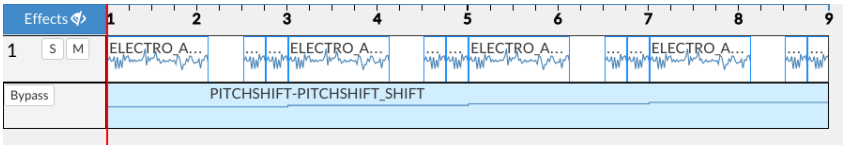
```
var panValues = [100, -100];
var panString = "1+++++0+++++++1+++0+++++1++0++++";

for(var measure = 1; measure < 9; measure += 2) {
 rhythmEffects(1, PAN, LEFT_RIGHT, panValues, measure, panString);
}

finish();
```

Case Study 5: Pitchshifting

Using Pitch Change and Modulation to Increase Tension

Modulation means to change the pitch center of a musical selection up or down. Composers use modulation in upward motion to increase tension within music. The following example uses a string passage that changes pitch upwards by 1 semitone every 2 measures.

FIGURE 19-6



Code:

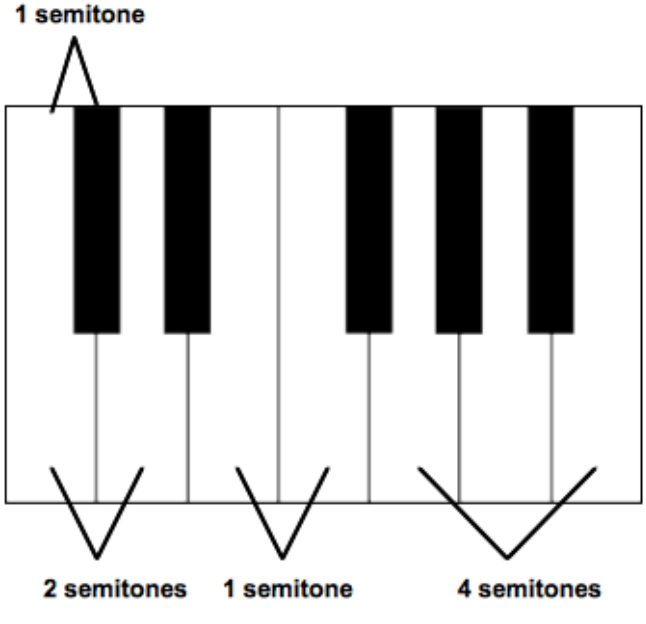
```
//Pitch Change for Modulation

init();
setTempo(182);

var synth = ELECTRO_ANALOGUE_LEAD_001;

var synthBeat = "0+++++++++++++++-----0+++0+++";

for(var measure = 1; measure < 9; measure += 2) {
 makeBeat(synth, 1, measure, synthBeat);
}
```

```
for(var pitch = 0; pitch < 4; pitch++) {
 var start = (pitch * 2) + 1;
 var end = start + 2;
 setEffect(1, PITCHSHIFT, PITCHSHIFT_SHIFT, pitch, start, pitch, end);
}

finish();
```

Notes for Pitch:

A “semitone” is a half step in music. An interval is the “distance” between pitches measured in semitones. Below you can see how semitones relate to the keys on a keyboard:

FIGURE 19-7



Common Intervals:	
Half Step	1 semitone
Whole Step	2 semitones*

Major 3rd	4 semitones
Perfect 4th	5 semitones
Perfect 5th	7 semitones
Minor 6th	8 semitones
Major 6th	9 semitones
Minor 7th	10 semitones
Major 7th	11 semitones
Octave	= 12 semitones

* A common modulation technique in popular music to to move up by 2 semitones or 1 whole step, usually on return to the chorus section after a bridge

Abstracting the remix

This example follows from the lesson on **Remixing the Rhythm** in "Randomness and Strings".

In "Remixing the Rhythm", we looked at adding some rhythmic effects to a predefined drum beat. If we want to perform this operation more than once, and generalize the parameters, we should abstract it by making a function.

Below, we have added several features. Our "random string insertion" procedure is now a function. A very similar function called `beatRepeat()` has also been added. It produces a stuttering-like sound, because it repeats the drum sound of a specified location in the `amenBreak` string. The two of them together provide some balance of repetition and randomness.

```
// javascript code
//
// script_name: Amen Remixer
//
// author: The EarSketch Team
//
// description: Two functions (insertRandom and beatRepeat) for remixing the Amen b
//
//
//

//Setup
init();
setTempo(170);
```

```
//Music

var drums = [OS_KICK05, OS_SNARE06, Y24_HI_HATS_1, Y58_HI_HATS_1, OS_OPENHAT01];

var a = "0+0-1+-1+1001+-1";
var b = "0+0-1+-1-10---1+";
var c = "-1001+-1+10---1+";
var cym1 = "2+2+2+2+2+2+2+2+";
var cym2 = "2+2+2+2+2+3+2+2+";
var cym3 = "2+2+2+2+2+4+2+2+";

// Creating a long string by concatenating shorter segments. "amenBreak" is our important str
var amenBreak = a + a + b + c; // 16 * 4 beats = 64 beats
var beat2 = cym1 + cym1 + cym2 + cym3;


// Choose a position in our beat string, and the drum at that position will be repeated for nu
// Returns a new string
function beatRepeat(track, beatPosition, numBeats, beat) {
        var newBeat = beat;
        // We need something to concatenate to, so we initialize this variable to an empty st
        var insertSection = "";
        var drumToRepeat = beat[beatPosition];

        for(var i = 0; i < numBeats; i++) {
                insertSection += drumToRepeat;
        }

        // Check if our beatPosition is too late to be useful
        if(beatPosition >= beat.length-2) return;

        // If our beatPosition is at the start, we need to make our frontSection in a differe
        if(beatPosition == 0) {
                var front = "";
        } else {
                var front = newBeat.substring(0, beatPosition);
        }

        var end = newBeat.substring(beatPosition+numBeats, beat.length-1);
        newBeat = front + insertSection + end;
        return newBeat;
}

// Choose a position in our beat string, and random drums will be inserted at that position f
// Returns a new string
function insertRandom(track, beatPosition, numBeats, beat) {
        var newBeat = beat;
        // We need something to concatenate to, so we initialize this variable to an empty st
        var insertSection = "";
```

```
        for(var i = 0; i < numBeats; i++) {
                // Concatenating random numbers using +=
                insertSection += Math.floor(Math.random() * 5);
        }

        // Check if our beatPosition is too late to be useful
        if(beatPosition >= beat.length-2) return;

        // If our beatPosition is at the start, we need to make our frontSection in
        if(beatPosition == 0) {
                var front = "";
        } else {
                var front = newBeat.substring(0, beatPosition);
        }

        var end = newBeat.substring(beatPosition+numBeats, beat.length-1);
        newBeat = front + insertSection + end;
        return newBeat;
}


// Each beatRepeat and insertRandom call returns a transformed version of beat,
// so we assign the returned value to beat.
amenBreak = beatRepeat(1, 32, 9, amenBreak);
amenBreak = beatRepeat(1, 17, 5, amenBreak);
amenBreak = insertRandom(1, 0, 16, amenBreak);
amenBreak = insertRandom(1, 7, 3, amenBreak);

makeBeat(drums, 1, 1, amenBreak);
makeBeat(drums, 2, 1, beat2);
makeBeat(drums, 1, 5, amenBreak);
makeBeat(drums, 2, 5, beat2);

setEffect(1, FILTER, FILTER_FREQ, 4000.0);

//Finish
finish();
```

When you abstract a procedure, you need to carefully consider the different kinds of input, and make sure your function can respond reliably regardless of the input. That is the reason for the conditional check in our function: when the input is 0, it must respond differently.

On your own, try making the `beatPosition` random for each call.

EarSketch Sound Library 20

To find sounds that work well together in your music, choose them from the same folder. For example, pick all your sounds from DUBSTEP_140_BPM or all of them from Y30_68_BPM_B_MINOR.

To hear a sound, click the play button next to its name. To use a sound in EarSketch, click the paste button to paste the sound constant into your script at the current cursor position. To help you find sounds, you can search by keyword and you can filter by artist, genre, and instrument.

Most of the sound folders are labeled with a tempo. Tempo is the speed of music (how slow or fast). In EarSketch, you always specify the tempo of your song using the setTempo function. For example, setTempo(120) will set the tempo of your project at 120 beats per minute. Though not required, we suggest you set the tempo to match the original tempo of the sounds you choose. To find the original tempo, click on the tags button next to the sound.

As you get more comfortable creating music with EarSketch, you may want to experiment some more by combining sounds from several different folders in the same song and by trying out different tempos. When experimenting, use your musical ear to help you decide what sounds good and what doesn't, and try a bunch of different possibilities to figure out what you like best.

Some of the sound folders also list a key for the sounds in that folder. For instance, the sounds in Y57_87_BPM_G_MINOR are in the key of G minor. In music, a key indicates a scale (like major or minor) and a particular "home" note (called the tonic) from which most of the pitches are drawn. So if you want to combine sounds from two different folders (other than just drums), they are more likely to sound good together if they are in the same key.

The sounds in the EarSketch library come from several different sources. **Richard Devine** (a well-known sound designer and electronic musician) and **Young Guru** (a Grammy-nominated DJ and audio engineer best known for his work with Jay-Z) each created about 2000 sounds specifically for EarSketch. Some additional sounds in the MAKEBEAT folder contains a collection of single drum hits suitable for use with EarSketch's makebeat function. These were created by Thom Jordan, a member of the EarSketch research team.

To add your own sounds to EarSketch, follow the instructions on our web site for **Chapter 22**. Your sounds will then be displayed in the USER_SOUNDS folder here.

Programming Reference 21

Online JavaScript Interpreter

Codecademy Labs: Run JavaScript code in your browser for instant feedback. You can share your code snippets with others, and download them to your computer. *Note: You cannot use any modules from the EarSketch API in this editor.*

External Help

Official JavaScript Tutorials: These tutorials contain many useful explanations and examples to help you learn to code in JavaScript.

What is Programming?

Programming is the process of writing instructions that a computer uses to perform tasks. These instructions are written in a very specific format called **code**. When a programmer writes a program in code, the computer executes the code one line at a time until it reaches the end. With good programming skills and knowledge, you can write code that can create almost anything imaginable.

Computers can only understand problems in a very specific way. In order to produce a program that a computer will be able to understand and run, a problem needs to be broken down into many small steps. In turn, these steps must be written in code of the appropriate language and in the explicit order in which the computer should complete them.

There are many different types of programming languages that consist of particular and unique terms, phrases and conventions. The set of rules that define the combination of symbols that are used within a certain language is called that language's **syntax**. Every language, including English, has a different syntax and, therefore, must be written differently to accomplish correct form. For example, in English, we capitalize proper nouns and add punctuation to the

ends of our sentences as a part of the syntax of the language. There are different and specific syntax rules for all of the different types of programming languages. Some of the most popular programming languages used today are Java, JavaScript, Python, and C.

Programming Terms

Boolean

A variable data type that stores a value that is either `true` or `false`

Commenting

Comments are sections of code that the computer does not execute. These sections are made for the programmer to explain portions of code in order to make his or her program more readable to himself or herself and to other people. In JavaScript, comments are denoted with `//`. Any text after a`//` is considered to be a comment. Multi-line comments act the same way as single line comments, but must begin with `/*` and end with `*/`. Example:

```
// This is a comment

/*
this is a multi-line comment
it extends across multiple lines
*/
```

Conditional

A type of statement that checks to see if an argument is true or false before executing some amount of code. Types of conditionals are if-statements, else-if-statements and else-statements. For example,

```
if(x < 0) {
 println("Negative.");
} else if (x == 0) {
 println("Neither positive nor negative");
} else {
 println("Positive.");
}
```

This code reads as: if the variable x (defined elsewhere in code) is less than zero, print negative, however, if the parameter x equals zero, it is neither positive nor negative. Otherwise, x is positive.

Object

A collection of key-value pairs that maps from keys to values (often called a map or dictionary in other programming languages).Example:

```
var greetings = {
     "hello":["hola"],
     "goodbye":["adios"]
};


println(greetings.hello);
println(greetings.goodbye);
```

`hello` and `goodbye` are keys whereas `hola` and `adios` are values. To get a value from one of the object's keys (called a "property" in JavaScript), you write the object's name followed by a dot and the key: `object.key`

Function

A named sequence of statements that performs some useful operation. Functions may or may not take in parameters. Each call you make to the EarSketch API is called a function call. This calls for the code that defines that function to execute.

```
//defining a function
function myFunction(string) {
        println(string);
}

//calling a function
myFunction("hello world");

//the result is that 'hello world' gets printed to the EarSketch console
```

Index

An integer variable or value that indicates an element of an array. The first element of an array has an index of zero.

```
// an example array of colors
var list = ["red", "blue", "green", "yellow"];

//call the value at index 2
println( list[2] );

//'green'

//assign a new value to index 2
list[2] = "orange";

//print index 2 again
println( list[2] );

//'orange'
```

Keyword

A reserved word that is used by JavaScript to parse the program; you cannot use keywords as variable names. (A list of JavaScript keywords are available after this section).

Array

A named collection of objects, where each object is identified by an index. The items within an array can be of any data type (e.g. int, str, float). Example:

```
var array = [1, 1.5, 2, 2.5, 3];
```

- the value of array[0] is 1
- the value of array[1] is 1.5
- the value of array[4] is 3

After being created, it is possible to change elements of an array by reassigning the different indices to different values. For example, if you have the array above, you could change the first element at index 0 in the following way:

```
array[0] = 5;
```

Now the first element is no longer 1. It is 5.

Loop

A statement or group of statements that execute repeatedly until a terminating condition is satisfied. There are two main types of loops: a for loop and a while loop. For loop:

```
for(var i = 0; i < 10; i++) {
        println("Hello");
}
```

The above code prints “Hello” once for each number in the range between 1 and 9. For loops are useful if there is a specific number of items that you wish to iterate over. While loop:

```
var n = 5;

while(n > 0) {
 println(n);
 n = n-1;
}
```

The above code first checks to see if n is greater than zero. If this is true, n is printed and then decremented by 1. The loop continues to execute until the condition that n is greater than zero is false. While loops are useful if you do not have a specific set of items to iterate through. In any situation, a for loop can be written as a while loop and vice versa. It is ultimately up to the programmer to choose which will work best for any given situation.

Number

A JavaScript data type that consists of both integers and floating-point numbers. Floating-point numbers have decimal components. Note that integers and floating-point numbers are not different data types in JavaScript, as they are in many other languages (they are called “literals”). Examples:

```
//Floating point numbers
var a = 0.5;
```

```
var b = -2.0;
var c = 6.67

// Integers
var d = 50;
var e = 0;
var f = -12;
```

Parameter

A name used inside a function to refer to the value passed as an argument. For example, in the function `setTempo(tempo)`, `tempo` is the parameter.

return

The return keyword signals to the program that the end of a function has been reached. When a return statement is executed, the currently running function will terminate. The function can simply return with no output, or it can return a value. A function will automatically return after executing all of its statements, even if `return` is not written. For example:

```
function add(x,y) {
        var result = x+y;
        return result;
}
```

The above code will return the sum of x and y.

String

A JavaScript data type that holds characters. A character can be a letter, number, space, punctuation or symbol.Examples:

```
var myString = "This is a string!"
var myOtherString = "$tring..."
```

Recording & Uploading Sounds 22

EarSketch allows you to record or upload your own sounds, to use as clips in your EarSketch projects. Using your own sounds and clips in an EarSketch project is a fun way to make your music more personalized. You can access this feature in the Sound Browser, by clicking the "plus" sign left of the search bar.

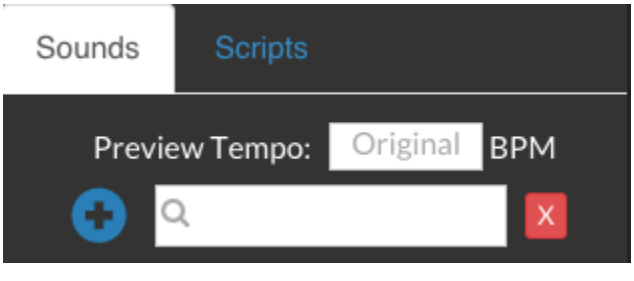
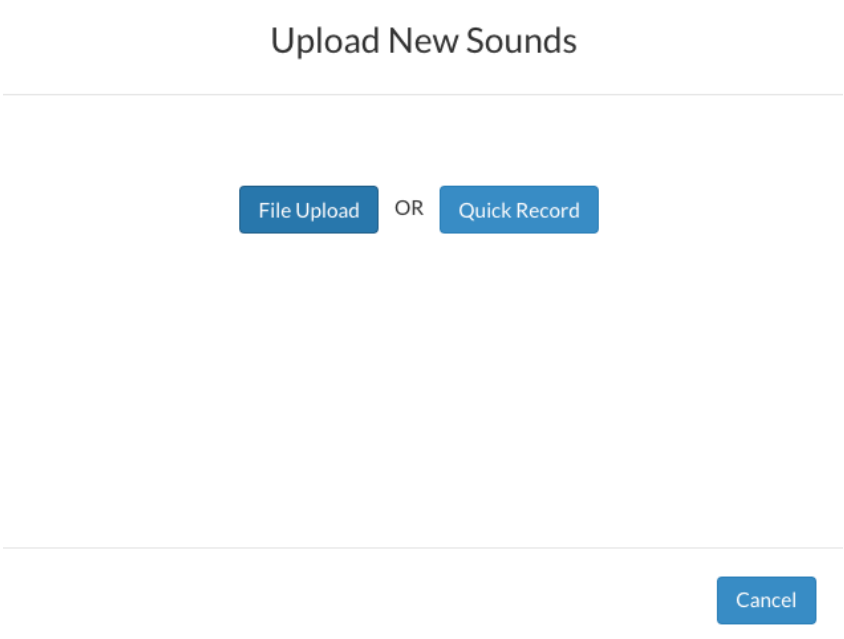


FIGURE 22-1

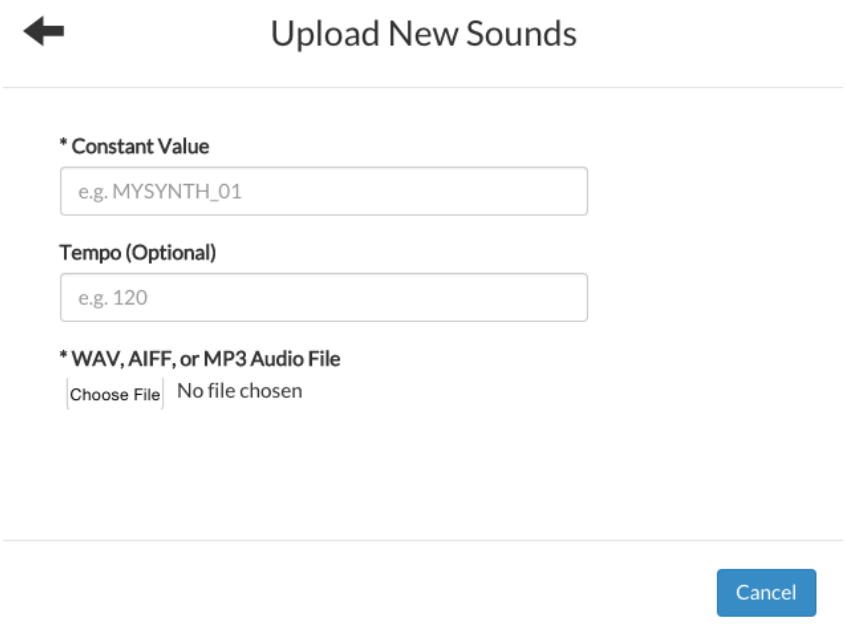
Press the plus sign to upload or record

This brings you to a menu, where you can choose to either upload or record sounds:



Uploading audio files to use in EarSketch

EarSketch only currently supports sound files uploaded in most common audio file formats. Once you've found an audio file you want to use in an EarSketch project, you can click "File Upload" to upload it from your computer. Select the audio file you want to upload and give it a name.



If you know the tempo (BPM) of your sound, you should specify it in the dialog box. Once EarSketch knows the tempo of your sound, it will automatically speed it up or slow it down to match whatever tempo you specify in your EarSketch script using `setTempo()`. Your sound will then sync up rhythmically with the rest of your tracks. If you don't know the tempo, leave the tempo box blank. EarSketch will not speed up or slow down your sound to match `setTempo()`, but this is fine if your sound is a sound effect or a drum hit or something else that doesn't need to sync rhythmically with other tracks. It is also fine for when you are using your sound with `makeBeat()`.

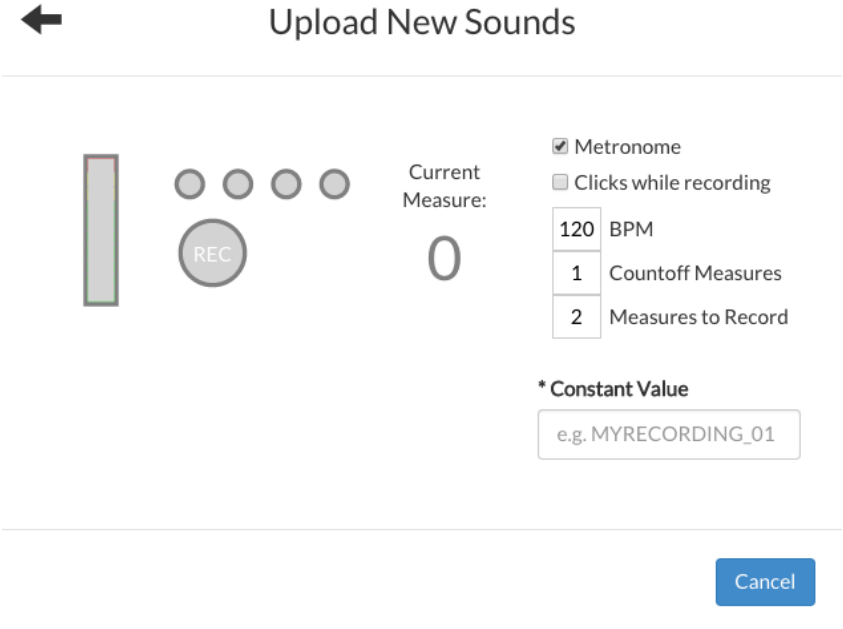
Recording your own sounds

If you want to record directly into EarSketch, click on "Quick Record" in the "Upload New Sounds" screen (see above). Your browser will probably ask you for permission to use the microphone, like below. Press "allow" or "share".

On the next screen, we have the recorder. Choose the number of measures you want to record, set the tempo for the clip, specify the number of countoff measures (this gives you 4 clicks per measure before recording so you know how fast to play), then press REC to start recording! When you're done, give your clip a name (bottom right) and press "Upload the Sound File".

FIGURE 22-2

The EarSketch Recorder



Using your uploaded/recorded sounds

Once your sound is uploaded or recorded, you can find it in the Sound Browser. To display your sounds, click "Artists", then click your username (shown below as "EARSKETCH_USER") to display only your sounds. Upload as many clips as you want and start personalizing your EarSketch projects with original sounds!

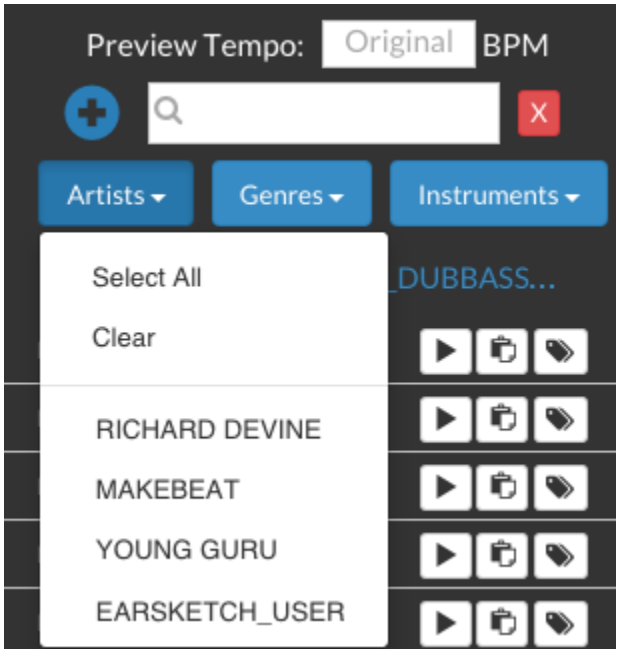


FIGURE 22-3

Finding your clips with the "Artists" menu

Copyright 23

What is Copyright?

When you own your car, or your cell phone, or your watch, that ownership is pretty simple: if you paid for it, then it's yours, and if someone takes it from you without your permission then you no longer have it and they've stolen it. But what about things like a movie or an invention or a song - things that you can't see or touch? Someone can't steal a song that you wrote in the same way that they can steal your watch, but that song still has value. So there can still be ownership in these things as well - we call that **intellectual property**. There are different types of intellectual property that deals with other sorts of things like inventions or brand names, but what's important for music is **copyright**. Copyright is the part of the law that covers ownership in creative work. Whether you've made a movie or written a newspaper article or created a song, copyright tells us both what other people can't and what they can do with it. As musicians, copyright is important because: (a) it keeps people from using your work in ways you wouldn't want (for example, someone else selling it without your permission); and (b) it lets you know when and how you can use other people's work when creating your own (for example, remixing and sampling).

Credit: **copyright_001.jpg** / **Olivia Hotshot** / **CC BY-NC 2.0**

Copyright Basics

Here's the first thing you should know about copyright: You probably already have one! All you have to do to get a copyright in something is to create it in some form outside your head. When you doodle pictures in the margins of your notebook during class? Copyrighted. When you write down song lyrics? Copyrighted. When you lay down an original drum beat? Copyrighted. It's a little more complicated than this since you have to be sure that there's enough of something and that it's original enough to be copyrightable (maybe not that 140-character Twitter status), but as a general rule, if you make something creative and new then you have a copyright automatically. So what does that give you? In the United States, having a copyright gives you six exclusive rights over what you create:

- to make copies
- to make derivative works (a new work based on the original - like a movie adaptation of a book)
- to distribute copies

- to perform publicly
- to display publicly
- to digitally transmit

Copyright infringement is when anyone other than you does any of those things with your work without your permission - unless it's covered by some exception (like fair use, which we'll cover in a later module). A really common example of copyright infringement these days is illegally downloading music. If you upload a song without permission onto the Internet so that other people can download it, then you've violated both the right to make copies and the right to distribute copies. If you download a song that someone else put up illegally, then you've violated the right to make copies as well. If you create an original piece of music and put it on the Internet, even if it's not for sale, then if someone copies it without your permission, then they've violated your copyright. A lot of times you might not mind - in fact, you might want people to share your music! When you read about licenses you'll learn some ways to signal that this is the case. Remember that copyright isn't all about getting people into trouble. It's even in the Constitution, which says its purpose is to "promote science and the useful arts." Copyright should help us make and share more art, not less. When you're creating things in EarSketch, we want you to think less about *stealing* and more about *sharing*. EarSketch works because artists have shared their work with you, resulting in the library of samples you use to make new music - using the social media site or letting other students remix your code is a way of paying that forward and helping to put new art into the world.

Music and Sampling

One thing you should know about copyright is that it can get pretty complicated, especially when you consider laws from a lot of different countries. This module gives you a basic idea of how music copyright works in the United States, and how it applies to sampling.

Credit: **Rocketship Music (#96639)** / **mark sebastian** / **CC BY-SA 2.0**

Copyrights in Music

When you hear a song on the radio, or download one from iTunes, there are actually two copyrights involved, not just one - the rights to the song and the rights to the sound recording. So when you hear, for example, the song "Hurt" by Nine Inch Nails, there are two pieces of the pie to carve up: (1) ownership in the song (the composition) by Trent Reznor, because he wrote it; and (2) ownership in the sound recording most likely by A&M because they're the record company. And then when you hear "Hurt" by Johnny Cash, there are still two pieces of the pie: (1) Trent Reznor still owns the composition; and (2) Universal Music Group owns the sound recording because they were Cash's record label. What this means is that when someone needs permission to use a song, then they might have to get two pieces of permission, not just one.

Part of this distinction in the two copyrights covers who gets money when. For example, for record sales, much of the royalties go to the owner of the sound recording copyright (usually, the record label), but royalties from public performances (which includes radio play) go to the songwriter. Because it would be pretty difficult for a songwriter to go around and collect royalties from every restaurant or hotel or radio station where someone is performing her mu-

sic, there are some organizations that do this for them. ASCAP (American Society of Composers, Authors and Publishers) is the biggest, and it negotiates and collects fees from all of these places and divvies it up among songwriters.

The important thing to know here is that the composition of a song and the recording of a song have different copyrights. If you're using samples, then you're probably dealing with the sound recording copyright in deciding whether you have permission to use something or not. If you create music yourself but it's a cover of an existing song, then the copyright at issue is the composition.

Legal Issues with Sampling

Sampling means taking part of a sound recording and using it in a new piece of music. All of the sounds in the EarSketch library are samples. You're not creating sounds from scratch, but instead combining and using them in new ways.

The use of samples in music has been causing legal controversy for a long time. In order to sample a piece of music you have to copy that small part of it, which could be a violation of the copyright in that sound recording.

In the very early days of sampling - when it involved manipulation with tape recorders - no one really asked for permission. However, by the 1980's, it was not uncommon for copyright owners to complain about samples, and so many artists using sampling began to license instead - that is, to get permission to use part of a song.

Sometimes it is hard to identify a sample in a song because it is such a small part. Could just a few notes really be copyright infringement? According to a court case from 2005, even as little as three notes could be infringement. This is why most artists "clear" the samples before using them - they pay for a license, or permission, from the owner of the original recording. Of course, copyright law contains some exceptions where certain kinds of uses are not infringement - you will learn about **fair use** in the next module.

Fair Use

You probably have some notion that copyright law has some exceptions. After all, if you couldn't ever copy anything, then how could you quote a line from a book in a book review or parody a television show on *Saturday Night Live*? Copyright is important, but so is free speech and creativity, and so in the United States **fair use** is the part of the law that acts as a sort of "safety valve" to keep copyright from going too far. It allows for some uses of copyrighted content under certain conditions.

What are those conditions? That is somewhat complicated. Fair use is decided on a case-by-case basis by a judge, and so there aren't any "bright line"

rules about what is a fair use. Instead, there are a set of four factors to be weighed to make the determination:

(1) The purpose and character of the use. What are some of the characteristics of the new use? If you are using part of a copyrighted work for educational purposes, or for critical commentary on the original, then it is more likely to be fair use. It is also more likely to be fair use if you are using something noncommercially - that is, you're not getting paid for it. Also, how transformative is the use - is it very different than the original? This is also the part of fair use that covers things like parody and satire.

(2) The nature of the copyrighted work. Basically, fair use is less likely if the original work is fiction rather than nonfiction, and if it is published rather than unpublished. Note that putting something on the Internet counts as being published.

(3) Amount used. The more of an original work you use, the less likely it is to be fair use. So quoting one line from a book in a book review is probably fair use. Copying an entire book but changing one word and republishing it is probably not.

(4) Market harm. Is the new work harming the market for the original? Think about it this way: Does the existence of this new work mean that people are less likely to buy the original? A use is also less likely to be fair use if it eats into a potential market for the original. For example, if you make a movie from a book without paying for the rights, then that means that the book's copyright owner might not be able to make their own movie.

So is sampling fair use? Unfortunately, there isn't a clear answer to that. A lot of very smart legal scholars and lawyers have come to completely different conclusions on that question. Because fair use is determined on a case-by-case basis, you don't know for sure until someone sues someone else and it goes to court. Though what we do know is that there are some very well known sampling artists - GirlTalk is one example - who aren't being sued, and one possible conclusion is that the copyright owners think that this work might be fair use.

You might also see a lot of stuff on the Internet that seems to be copyright infringement but isn't getting anyone in trouble - some of this might be fair use as well. Think about all of the image memes that you see passed around online - there is probably a copyright in the original image being used. However, since the use is noncommercial, it is transformative of the original, and it isn't harming the market from the original, it is probably fair use.

So fair use is one way that people are allowed to make new, transformative works based on copyrighted material. Unfortunately, the law is kind of confusing and it is hard to know exactly when a use is fair or not. This is another reason why most artists get permission for the samples they use. You'll learn about **licenses** in the next module.

Licensing and Free Culture

To **license** is to give permission. If you own a copyright in something and you want to let someone else use it, you typically don't just sign over your copyright - instead you give them permission for that use. For most songs that you hear on the radio with samples of other songs, the artist of the new song negotiated a license with the original song's owner to use that sample. However, licenses don't only work with a specific person in mind - you can also put a license on your work that will let anyone use it.

For example, all of the samples in EarSketch are licensed so that you are allowed to use them however you like. This means that all the music you create in EarSketch is totally yours, and you have the copyright, and you can do what you want with it.

Open, permissive licenses like this are sometimes called **free culture** licenses. This means that by putting your work out there so that other people can use it, you're contributing to a pool of art and culture that is freely available and in turn inspires more art. Just think: Right now you have all of this great material to work with. The musicians who created the samples in EarSketch have licensed it to us so that you can use it completely freely. Something you should consider as you continue to make music is whether you want to pay this forward and encourage other people use your work as well.

In EarSketch, we encourage you to remix one another's work. If you put your music up on the social media site, then other EarSketch users can remix it and make new and awesome stuff, and it gives you credit on the site so that everyone knows where the source material came from.

Outside of EarSketch, what if you think that it's great if someone remixes your music, but you want to make sure that you get credit for it, or that no one else makes money from it? There's a license for that, too!

Creative Commons licenses let creators specify what rights they keep and what rights they give away. By default, copyright is "all rights reserved" - that is, you keep all of those rights that we listed in the first module. Creative Commons (CC) lets you say "some rights reserved" - for when you want to let others use your art in some ways. Here are the possible parts of a CC license.

"You can use this work however you like, EXCEPT..."

- "... you have to put my name on it." - Attribution (BY)
- "... you can't change it at all." - No Derivatives (ND)
- "... you can't make money from it." - Non-Commercial (NC)
- "... you have to share whatever new thing you make under the same license." - Share-Alike (SA)

By default, all licenses include attribution (BY) but otherwise, you can mix-and-match. So the most permissive license you can use is CC-BY but you could

also use something like CC-BY-ND-NC (Attribution-No Derivative-Noncommercial).

How do you use a Creative Commons license? Really, all you have to do is use it - just put it on your work if you release it on the Internet, for example. However, **their website** also has a license chooser where you can go through and get HTML for whatever license you pick.

Also, the great news for music remixers is that Creative Commons means that you can find other people's work to use as well. One great resource is **ccMixter** which has tons of CC-licensed music that you're free to use and sample and remix so long as you abide by whatever CC license the creator puts on their music. Music also isn't the only media where there is a lot of CC-licensed work, and someday when you need to make an album cover you should have a look at **Flickr**,which lets you search photographs by license. That is also where all the photos in these modules come from! You can click on the links in the captions to see the original photograph and the license that the photographer used.

HOW TO: Free Culture / CC BY-NC 2.0

As remixers, you have a lot of options when it comes to your own work and finding other people's work that you can use. The best advice is to be cautious when using copyrighted work without permission. However, the more open everyone is - maybe starting with you! - then the easier that will be in the future.

Curriculum PDF 24

Click here to download a PDF version of the EarSketch Curriculum.

Teacher Materials 25

To access teaching materials for EarSketch, **click this link**. A password is required for access. If you are a teacher and you do not yet have the password, please **contact us**.