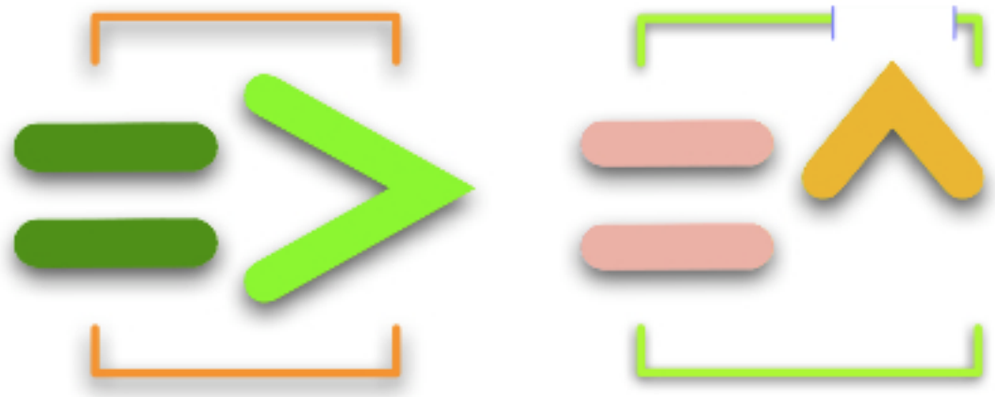
ERICA

소프트웨어학부

CSE2020 음악프로그래밍



1
Basics:
sound, waves, and ChucK programming

Properties of Sounds

- loudness
- pitch
- noise

Sound waves and waveforms

Sound consists of high and low waves of air caused by one or more vibrating objects.

Sound waves then propagate through the air, maybe bouncing off walls and other surfaces, finally reaching your ears or a microphone.

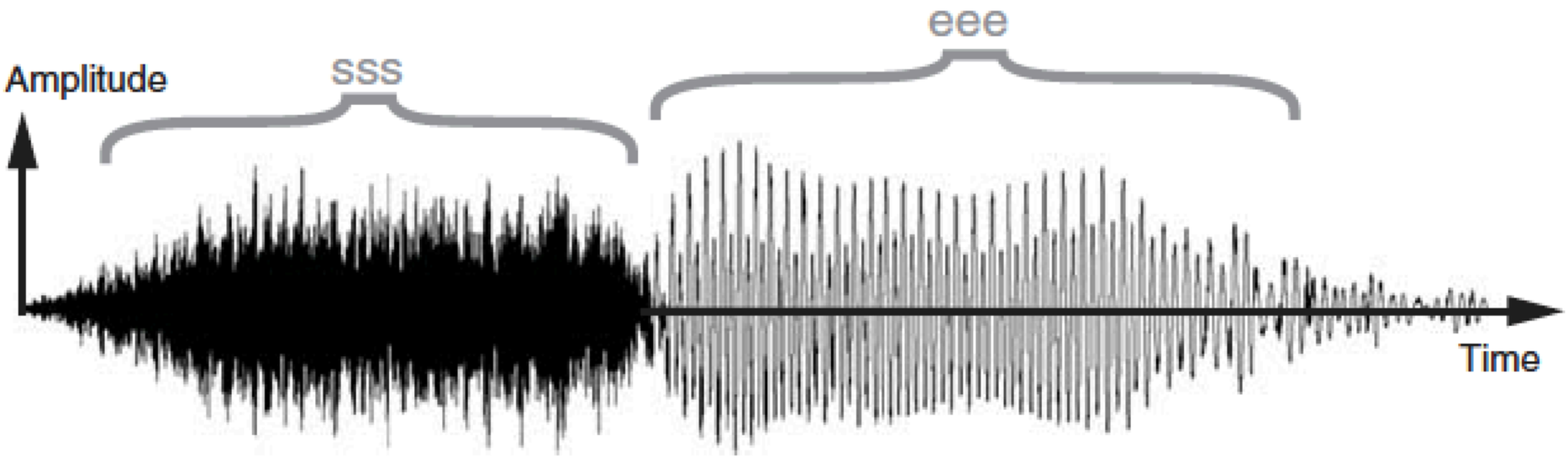


Figure 1.1 Waveform (sound amplitude versus time) of the spoken word "see"

Sound waves and waveforms

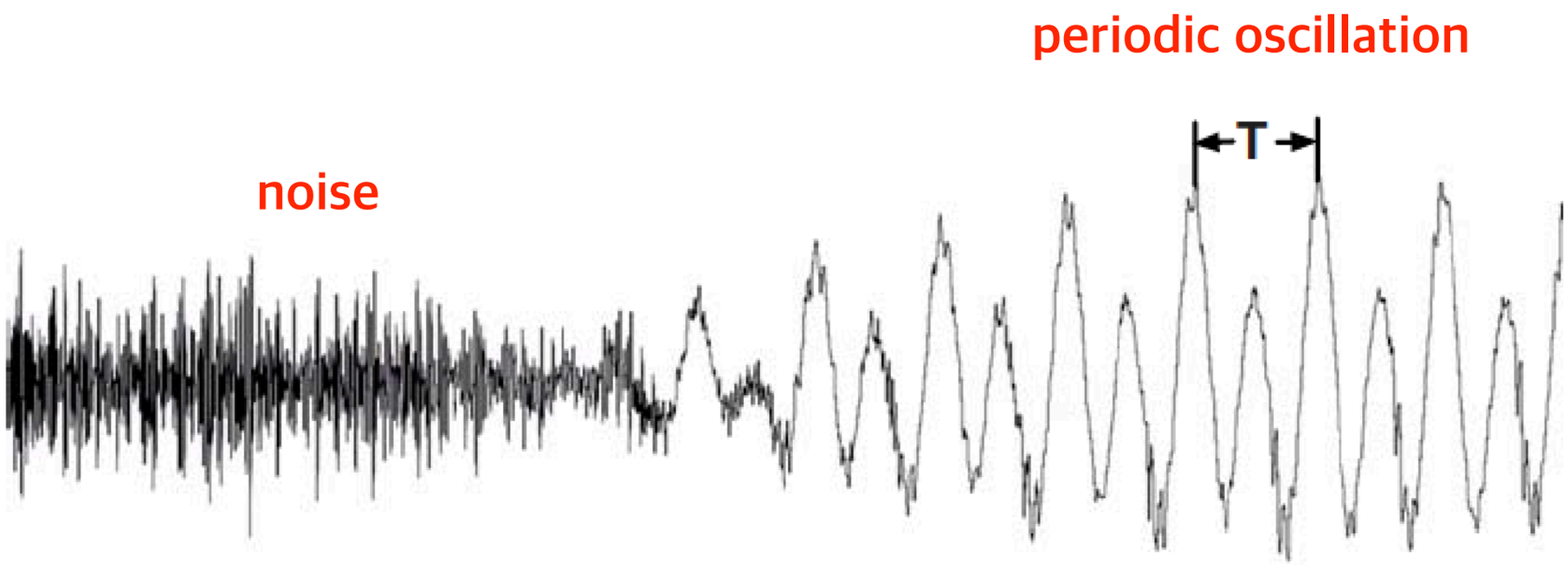


Figure 1.2 Waveform zoomed in to the transition between “sss” and “eee”

- T is the period of oscillation in seconds.
- 1/T is the **frequency** of the oscillation in cycles per second (Hz).
- Example:

  if t = 6.6 milliseconds (1/1000 seconds),

  then the frequency of oscillation is 1/0.0066 = 150 Hz

- **Pitch** is human’s perception of frequency from low to high.

Sine wave

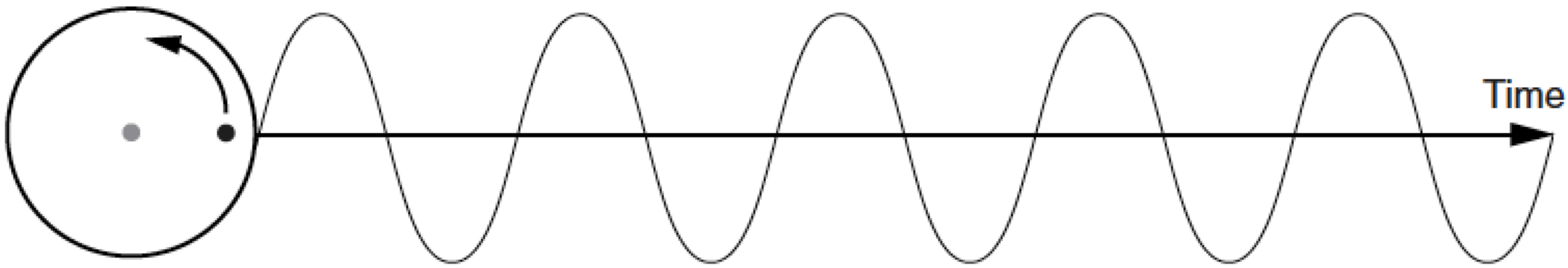


Figure 1.3 **A sine wave (right) can be generated by rotating the circle (left) counterclockwise and tracing the height of the dot as it changes in time.**

violin sound! clarinet sound!

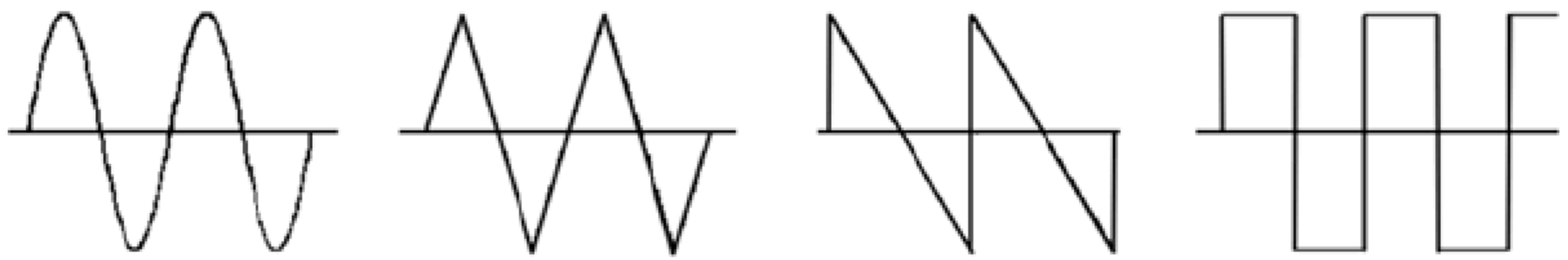


Figure 1.4 **Two cycles each of sine wave, triangle wave, sawtooth wave, and square wave**

Sound Synthesis

Complex periodic waves can be built by adding together sine waves of different

- frequencies,
- phases (delays), and
- amplitudes (loudness)

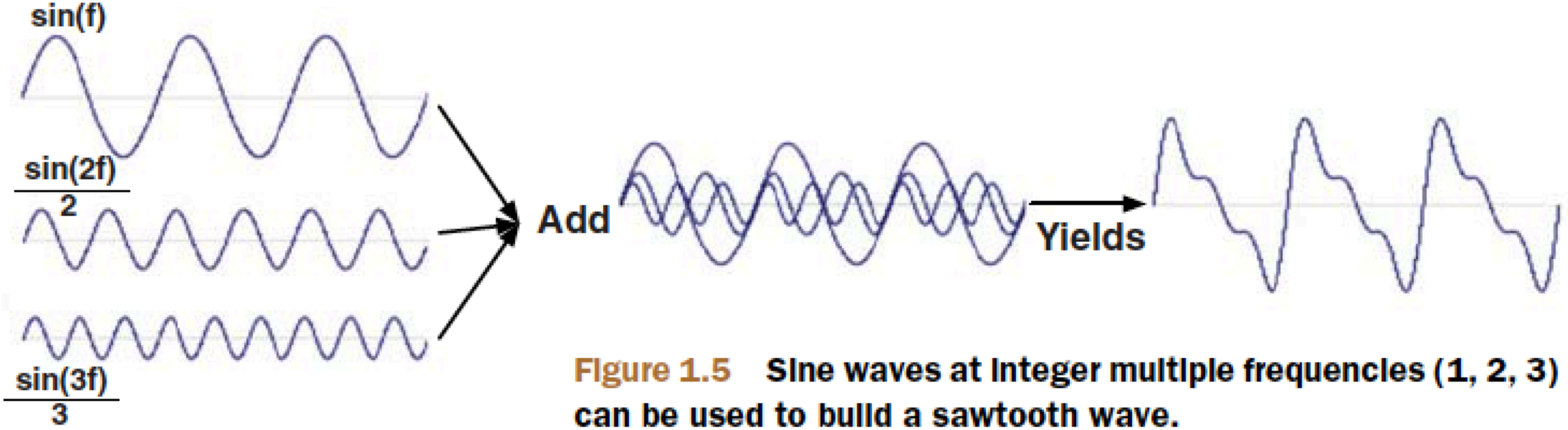


Figure 1.5 **Sine waves at integer multiple frequencies (1, 2, 3) can be used to build a sawtooth wave.**

Figure 1.6 Main windows of the miniAudicle ChucK Integrated Development Environment (IDE)

```
<<< "Hello World" >>>;
```

Hello Sine!

Listing 1.1 Hello Sine!

```
SinOsc s => dac;
0.6 => s.gain;
220 => s.freq;
second => now;
```

1. Connects (ChucKs) a sine wave
2. Sets oscillator gain to 0.6
3. Sets oscillator frequency to 220
4. Lets it play for one second

unit generator
UGen

digital-to-analog converter

duration

ChucKing

frequency
(pitch)
0~20,000 Hz

loudness
0.0~1.0

Listing 1.2 Sine wave music

```
/* Sine Music
by ChucK Programmer
January 2025          */

SinOsc s => dac;

// Play one note
220 => s.freq;
1.0 => s.gain;
0.3 :: second => now;

0.0 => s.gain;
0.3 :: second => now;

// Play another note, same pitch
1.0 => s.gain;
0.3 :: second => now;

0.0 => s.gain;
0.3 :: second => now;

// Play two more notes, higher, less loud
330 => s.freq;
0.3 => s.gain;
0.3 :: second => now;

0.0 => s.gain;
0.3 :: second => now;

0.3 => s.gain;
0.3 :: second => now;

0.0 => s.gain;
0.3 :: second => now;
```

1. A comment block; ChucK ignores this.
2. Connects sine oscillator to dac
3. Sets the gain to 1.0 and frequency to 220 Hz. Let this run for 0.3 second by ChucKing it to now (advance time).
4. Makes your sine oscillator silent for 0.3 seconds to separate it from the next note.
5. Repeats the process of blocks ❶ and ❸.
6. Repeats the same two-note pattern of blocks ❶, ❸, and ❹ but with a different frequency (pitch) and gain (loudness).

Test

WowExample.ck

```
Impulse imp => ResonZ filt => NRev rev => dac;
0.04 => rev.mix;
100.0 => filt.Q => filt.gain;

while (1)  {
    Std.mtof(Math.random2(60,84)) => filt.freq;
    1.0 => imp.next;
    100::ms => now;
}
```

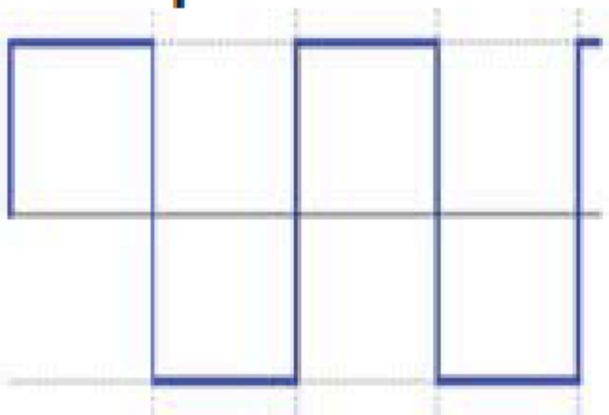
Listing3.8.ck

Listing6.15.ck

Listing4.11.ck

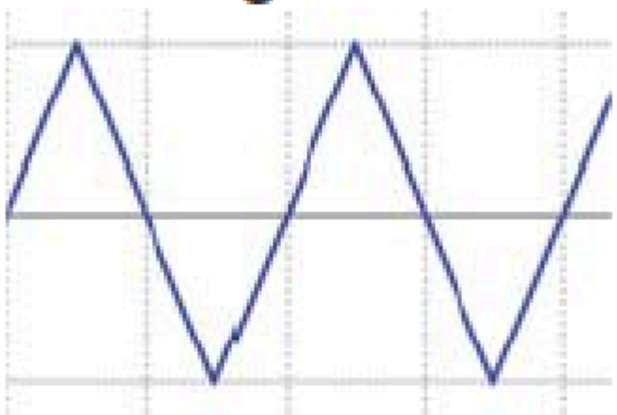
Other built-in waveforms

Square Wave



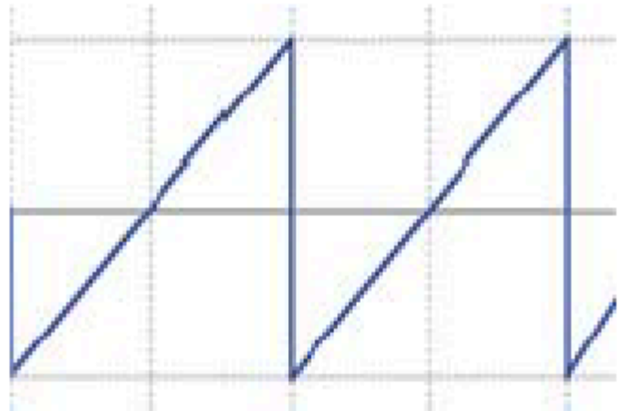
```
SqrOsc sq => dac;
```

Triangle Wave



```
TriOsc tr => dac;
```

Sawtooth Wave



```
SawOsc sw => dac;
```

Comments

```
/* Sine Music
by ChucK Programmer
January 2025          */

SinOsc s => dac;

// Play one note
220 => s.freq;
```

1. A comment block; ChucK ignores this.
2. Single line comment

Comments

Listing 1.3 Using comments to document code and control execution

```
// Author: ChucK Team
// Date: Today's date

// make a sound chain ("patch")
SinOsc s => dac;

// prints out program name
<<< "Hello Sine!" >>>;

// set volume to 0.6
.6 => s.gain;
// set frequency to 220.0
220.0 => s.freq;
// play for 1 second
second => now;

0.5 => s.gain; // set volume to 0.5
440 => s.freq; // set frequency to 440
2::second => now; // play for two seconds

// comment out this third note for now
/*
0.3 => s.gain;
330  => s.freq;
3::second => now;
*/
```

Initial comment, documents who wrote the program and when.

Sets up sound signal chain.

❶ Prints a greeting message.

Sets up parameters to play a note.

Plays another note.

❷ Don't play this last note right now, because we've commented it out.

Variables

Good coding practices: variable names

Because you're the programmer, you can pick any name you like for your variables, but it's almost always better to pick names that are meaningful. Names like `myPitch`, `myVolume`, and `fastTempo` are much more meaningful than `x`, `j`, and `z`. People reading your code can see those names and have an idea of their purpose. Later you'll see that some variables are temporary, used quickly in code just after they're created, and never used again. For these types of variables, `i`, `temp`, `x`, and the like are fine. But for most variables, a good name (not too long but definitely not too short) is important.

Data types and variables

Listing 1.4 Defining and using an integer variable

```
// declare an integer
int myPitch;

// store a value in it
220 => myPitch;

// print it out
<<< myPitch >>>;
```

You can also initialize integers at the time they're declared. For example, you can simultaneously declare an `int` named `myPitch` and set its value to `220`, as shown in the following listing.

Listing 1.5 Initializing an integer as it's declared

```
// another way to initialize integers
// store 220 in newly declared myPitch, all at once

220 => int myPitch;
   <<< myPitch >>>;
```

Arithmetics

Listing 1.6 Doing math with integers

```
// arithmetic with integers
220 => int myPitch;

// add or subtract
myPitch + myPitch - 110 => int anotherPitch;

// multiply
2 * myPitch => int higherPitch;

// divide
myPitch / 2 => int lowerPitch;

// print them all out
<<< myPitch, anotherPitch, higherPitch, lowerPitch >>>;
```

The code prints the following in the Console Monitor window:

```
220  330  440  110
```

Arithmetics

Listing 1.7 Shorthand math: multiply, subtract, and ChucK in one step!

```
// longhand math with integers

int myPitch;
220 => myPitch;

// multiply by 2
2 * myPitch => myPitch;

// subtraction
myPitch - 110 => myPitch;

<<< myPitch >>>;
```

```
// shorthand math with integers

220 => int myPitch;


// multiply by 2, in-place
2 *=> myPitch;  //  "times ChucK"

// in-place subtraction
110 -=> myPitch;  // "minus ChucK"

<<< myPitch >>>;
```

Both of these print the following in the Console Monitor window:

```
330  :(int)
```

Integer variables

Listing 1.8 Playing notes with integer variables

```
/* Sine Music with integer variables
by ChucK Programmer
January 2025          */

SinOsc s => dac;

220 => int myPitch;

1 => int onGain;
0 => int offGain;

// Play one note
myPitch => s.freq;
onGain => s.gain;
0.3 :: second => now;

offGain => s.gain;
0.3 :: second => now;

2 *=> myPitch;

// Play another note, with a higher pitch
myPitch => s.freq;
onGain => s.gain;
0.3 :: second => now;

offGain => s.gain;
0.3 :: second => now;
```

1. Declares and initializes an integer variable called myPitch
2. Declares and initializes two integers for controlling gain
3. Plays a note using your new integer variables
4. Turns off the sound of your oscillator to separate the notes
5. Multiplies your pitch by 2, in place
6. Sets frequency and turns on oscillator, to start your 2nd note
4. Turns off the sound of your oscillator again to end the 2nd note

Float variables

Listing 1.9 Using and manipulating `float` variables

```
220 => float twinkle;

1.5 * twinkle => float twinkle2;

1.6818 * twinkle => float lit;

1.2585 * twinkle2 => float tle;

2 * twinkle => float octave;

<<< twinkle, twinkle2, lit, tle, octave >>>;
```

1. Declares and initializes a float to hold your twinkle pitch.
2. Uses math to derive your 2nd twinkle pitch from the first one.
3. Uses more math to derive lit pitch from your base twinkle.
4. Uses more math to derive tle pitch from twinkle2.
5. Makes a new pitch an octave above twinkle.

The code produces the following console output:

```
220.000000      330.000000      369.996000      415.305000      440.000000
```

```
110 => float twinkle;
```
← an octave down

```
261.616 => float twinkle;
```
← transposition

Float variables

Listing 1.10 Twinkle with floats

```
/* Sine Twinkle Music with float variables
by ChucK Programmer
January 2025          */

SinOsc s => dac;

220.0 => float twinkle;
1.6818 * twinkle => float little;

1 => int onGain;
0 => int offGain;

// Play one note
twinkle => s.freq;
onGain => s.gain;
0.3 :: second => now;
offGain => s.gain;
0.3 :: second => now;

1.5 *=> twinkle;

// Play other note of 2nd "twinkle"
twinkle => s.freq;
onGain => s.gain;
0.3 :: second => now;
offGain => s.gain;
0.3 :: second => now;

// Play one note of "little"
little => s.freq;
onGain => s.gain;
0.3 :: second => now;
offGain => s.gain;
0.3 :: second => now;
```

You can make a float variable for your twinkle pitch and use math to compute the value of another variable to use later for little.

Turns twinkle note on (sets gain using onGain variable and advances time).

Turns off note (sets gain to offGain and advances time).

Modifies twinkle pitch using math, so you can do the 2nd, higher twinkle.

```
0.3 => float myDur;
myDur :: second => now;
```

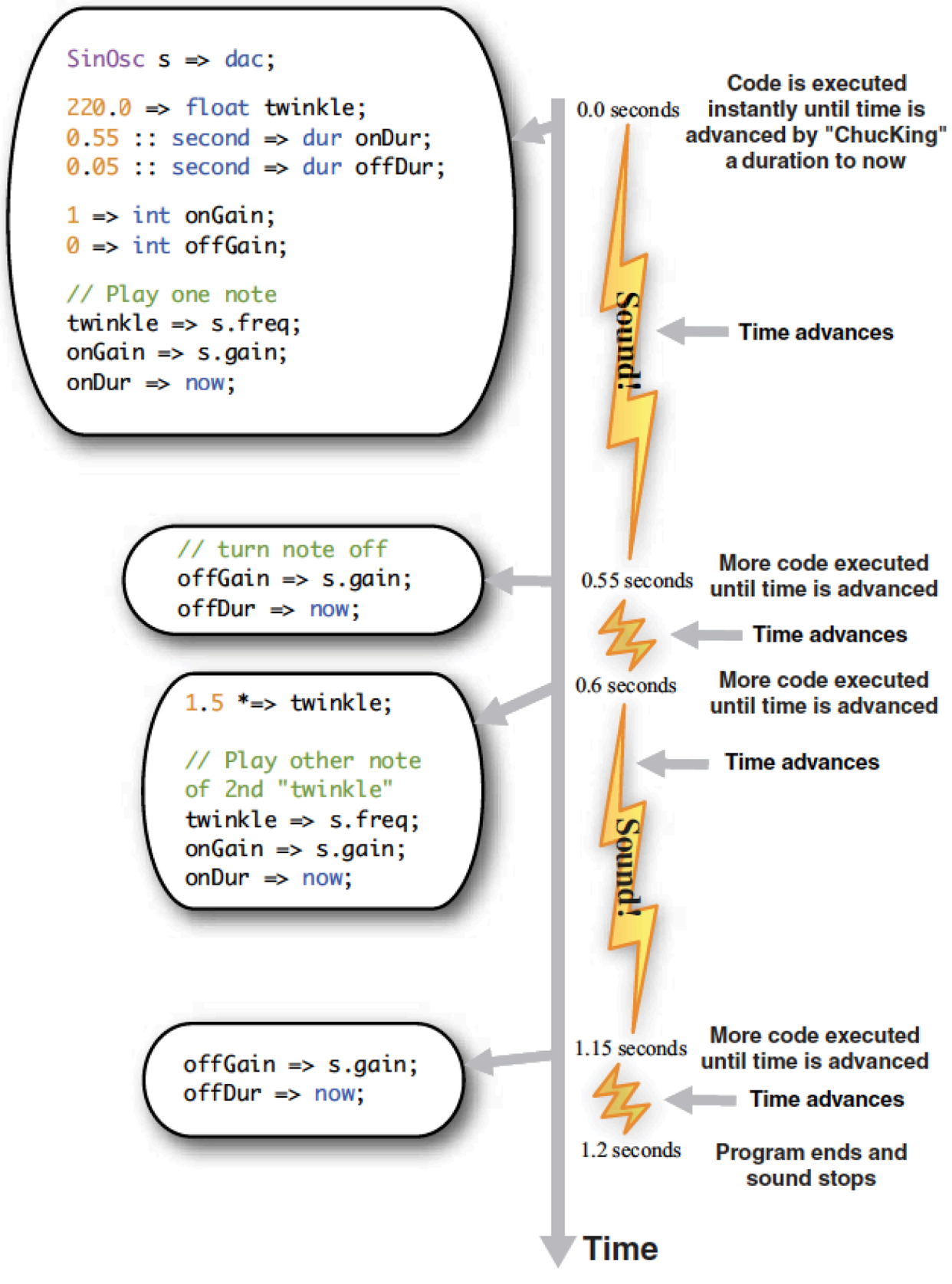
Time in ChucK

Listing 1.11 Twinkle with dur variables

```
/* Sine Music using dur variables
by ChucK Programmer
January 2025          */

SinOsc s => dac;

220.0 => float twinkle;
0.55 :: second => dur onDur;
0.05 :: second => dur offDur;

1 => int onGain;
0 => int offGain;

// Play one note
twinkle => s.freq;
onGain => s.gain;
onDur => now;

offGain => s.gain;
offDur => now;

1.5 *=> twinkle;

// Play other note of 2nd "twinkle"
twinkle => s.freq;
onGain => s.gain;
onDur => now;

offGain => s.gain;
offDur => now;
```

1. Defines note durations as variables
2. Waits while note sounds...
3. ...then waits for space between notes
4. Next note frequency
5. Sets it and plays another note

Time in ChucK



Time in ChucK

Table 1.1 Special ChucK duration keywords and their corresponding durations

Keyword	Duration
`samp`	1 digital sample in ChucK time (usually 1/44100 of a second)
`ms`	1 millisecond
`second`	1 second
`minute`	1 minute
`hour`	1 hour
`day`	1 day
`week`	1 week

ChucK data types

Table 1.2 ChucK data types

Data Type	Description	Example	Comment
`int`	Integer	`3, 3541`	No decimal point
`float`	Floating point	`2.23, 3.14159, 22.0`	Decimal point
`string`	Description	`"hello", "data/a.wav"`	Comment or text
`dur`	Distance between times	`1:: second, ms, 3*day`	Duration
`time`	ChucKianTime	`22050.0`	Time in samples
`void`	No type		

```
0.8::second => dur quarter;  // quarter note duration

      60 / 0.8 = 75 BPM(beats per minutes)

4::quarter => dur whole;

4 * quarter => dur whole;

whole / 2 => dur half;

quarter / 2 => dur eighth;
```

now

`now` is a special keyword, and is of type `time`.

- `now` is ChucK's master clock. When read, `now` holds the current logical ChucK time.
- Even though `now` is a variable, you cannot change its value directly.

no negative duration to `now`

Control structures

Listing 1.12 Random triangle wave music

```
/* Random Triangle Wave Music
   by ChucK Programmer */

  TriOsc t => dac;

// infinite loop runs forever
while (true)
{
    // randomly choose frequency from 30 to 1000
    Math.random2(30,1000) => t.freq;

    // random choose duration from 30 to 1000 ms
    Math.random2f(30,1000) :: ms => now;
}
```

Uses a triangle wave for variety.

1 Infinite loop runs forever (more on this shortly).

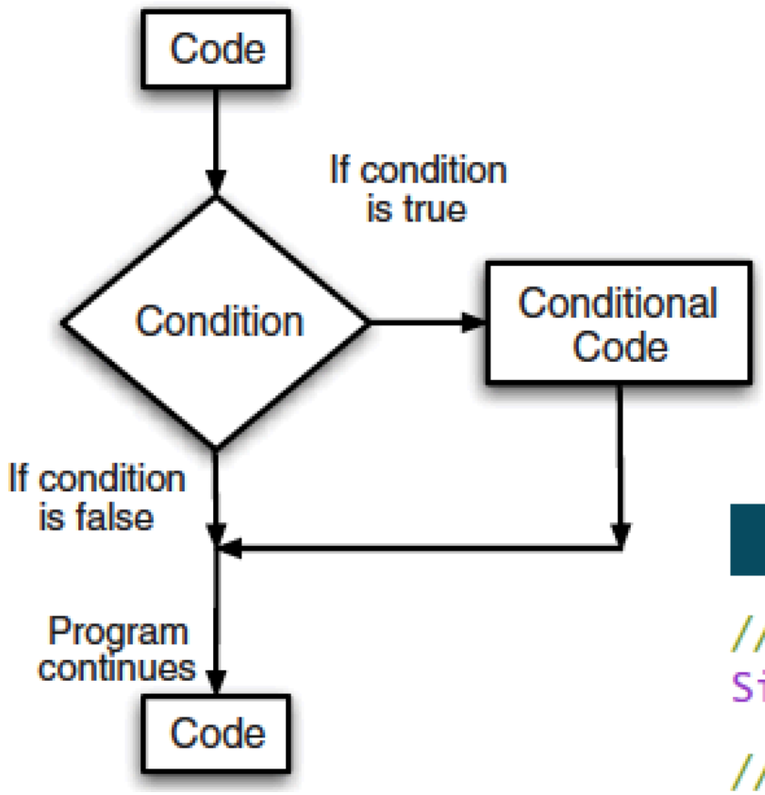
2 Generates a random number between 30 and 1000 and use that for your pitch.

3 Advances time by ChucKing a random number between 30 ms and 1 second to now.

Try this

Don't click Replace Shred each time you make a change to the code of listing 1.12. Instead, click Add Shred. Now you hear more magic of ChucK, which is that it can run multiple programs, called shreds, at the same time! You can add more running random sine programs, almost as many as you like, and hear that ChucK is happily generating many sounds at the same time. You'll use this power of ChucK and learn a lot more about it later, but for now, isn't it amazing how much sound/music you can make with just a little bit of code? We promised that your ratio of sound to typing would increase.

Control structures



Listing 1.13 `If` statement example

```
// sound chain
SinOsc s => dac;

// set frequency
220.0 => s.freq;
// set volume
0.6 => s.gain;

// chance logical variable
1 => int chance;

if (chance == 1)
{
    // sound will play only if chance == 1
    1 :: second => now;

}

// set new frequency
330.0 => s.freq;
1 :: second => now;
```

The usual sound patch.

Sets frequency and gain.

1 Defines an integer named chance to use as a logical variable (will have a value of either 1 or 0).

2 if statement. If the value in the parentheses is equal to 1, then do what's in the { }; otherwise skip that

3 Advances time and lets sound happen if true.

Sets new frequency for a different note…

4 …then plays this new note.

Logical conditionals

Table 1.3 Logical conditionals

Symbol	Meaning in words	Example use
==	is equal to	`if (x == 0)`
!=	is not equal to	`if (x != 0)`
<	is less than	`if (x < y)`
>	is greater than	`if (x > y)`
>=	is greater than or equal to	`if (x >= y)`
<=	is less than or equal to	`if (x <= y)`

```
<<< true, false >>>;
```

you'll see that ChucK prints

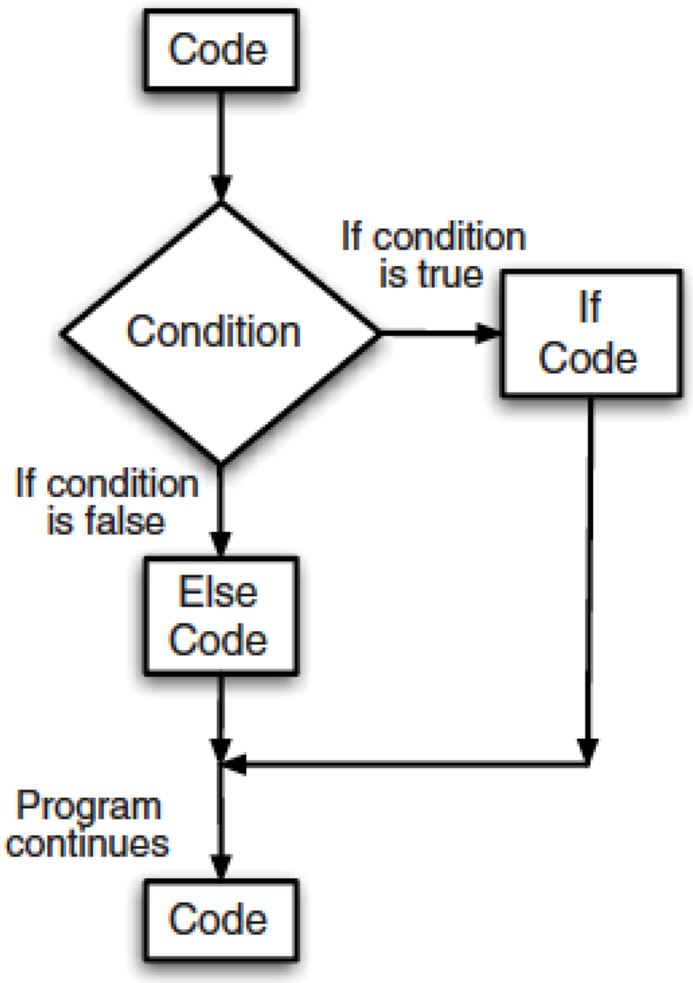
```
1   0
```

```
<<< -3 < 0, true==1, true==false, 1 > 10 >>>;
```

yielding

```
1   1   0   0
```

Control structures



Listing 1.14 `if/else` code example

```
 // sound chain
SinOsc s => dac;

// set frequency
220.0 => s.freq;
// set volume
0.5 => s.gain;

// chance logical variable
3 => int chance;

if (chance == 1)
{
    // play first "twinkle" note if chance == 1
    1 :: second => now;
}
else
{
    // otherwise, play other, higher twinkle note
    330.0 => s.freq;
    // and play it for a much longer time
    3::second => now;
}

// set and play one note, an octave above "twinkle"
440.0 => s.freq;
1 :: second => now;
```

Sine to dac, set frequency and gain.

1. New integer named chance, initialized to value of 3.
2. If chance is equal to 1...
3. ...do this block...
4. ...otherwise (else), set sine frequency to something different and play for a longer time.

After the if/else, play another, much higher note

Logical operators and conditions

Listing 1.15 More complex logical conditions

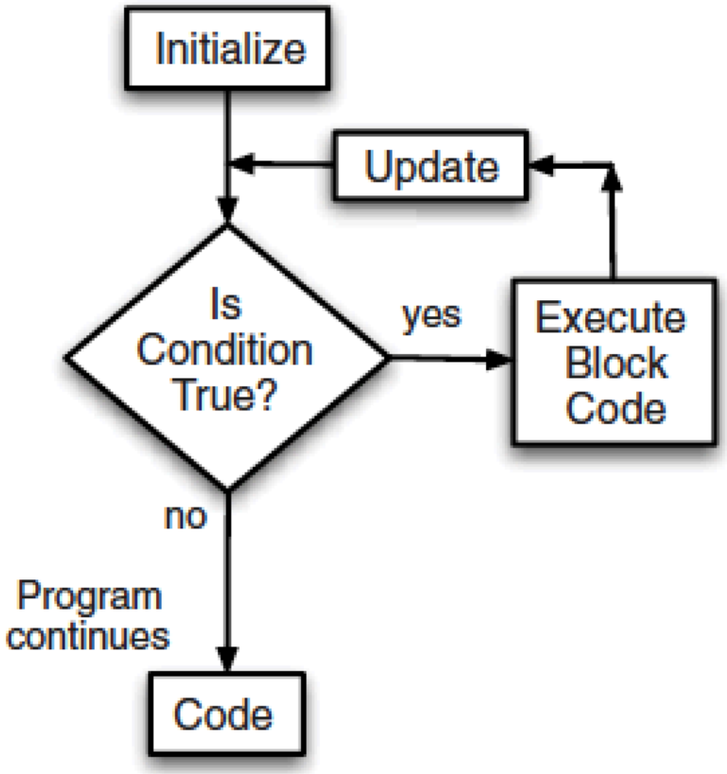
```
//conditional statements

if ( (chance == 1) || (chance == 5) )
{
    //code goes here
}

if ( (chance < 2) && (chance2 >= 6))
{
    //code goes here
}
```

Or condition. If chance is equal to 1 or 5, then the condition is true.

If chance is less than 2, and chance2 is greater than or equal to 6, then the condition is true.

Control structures

for loop

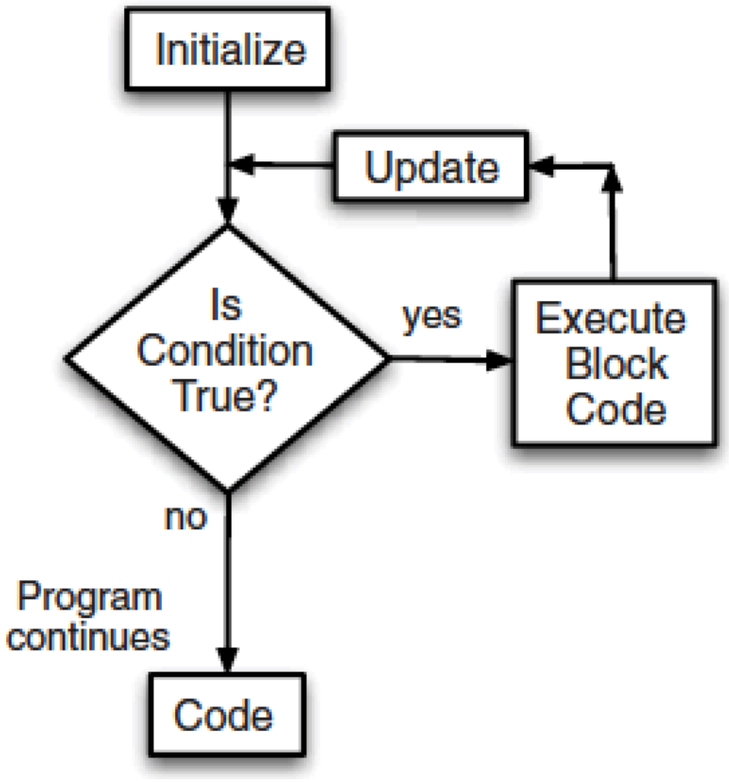


Listing 1.16 The for loop

Begin for structure

```
// For loop

for (0 => int i; i < 4; i++)
{
    <<< i >>>; // print value of i
    second => now; // advance time
}
```

1. Initial setup, usually declare and initialize a counter variable
2. Conditional test, end condition
3. Update thing(s) to do each time around the loop, usually increment your counter
4. Block of code to do each time, as long as conditional test is true



Control structures

for loop

Listing 1.17 Musical use of a for loop

```
// set up our patch
SinOsc s => dac;

// loop
for (20 => int i; i < 400; i++)
{
    <<< i >>>;   // print loop counter value
    i => s.freq; // set freq to i
    10::ms => now; // advance time
}
```

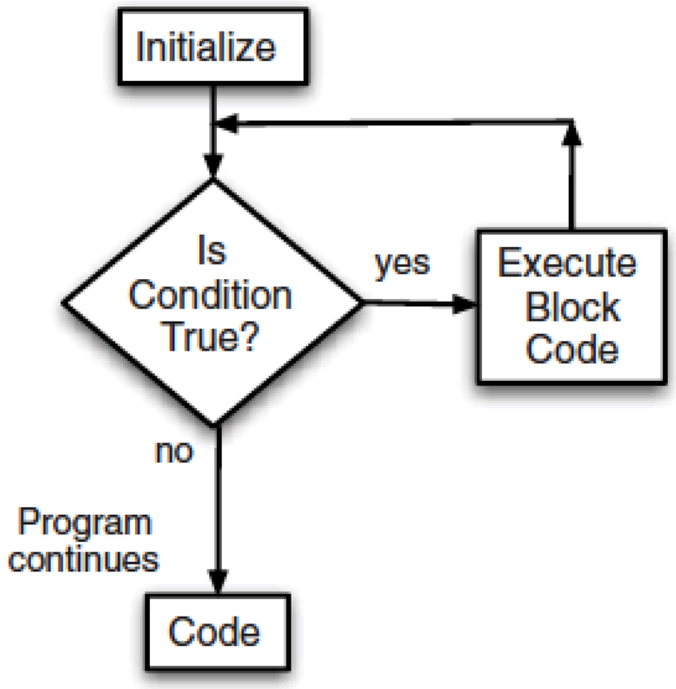
Sine wave to dac

A musical for loop

1. For loop from 20 to 399
2. Prints out current value of i
3. Sets frequency
4. Waits 10ms, then loops again

Control structures

while loop



Listing 1.18 Musical while loop

```
SinOsc s => dac;

// initialize variable i
20 => int i;

// while loop (instead of for loop)
while ( i < 400)
{
    <<< i >>>;
    i => s.freq;
    10::ms => now;
    i++;    // update counter (very important!!)
}
```

1. Initializes the counter/ frequency to 20.
2. The while loop only has a conditional test.
3. Block of code to execute, including…
4. …increment counter

Multiple Oscillators

Listing 1.19 Using more than one oscillator

```
// connect two oscillators to the dac
SinOsc s => dac;
SinOsc s2 => dac;

// set their frequencies and gains
220 => s.freq;
1030 => s2.freq;
0.5 => s.gain;
0.5 => s2.gain;

second => now;
```

Your usual sine wave.

1 Another sine wave oscillator.

2 Sets frequency of first sine.

3 Sets frequency of second sine.

4 Sets gains to 1/2.

Lets time pass so you can hear the sound.

mixing

- total gain = s.gain + s2.gain = 0.5 + 0.5 = 1.0
- A good rule of thumb is to scale their gains so that they add up to about 1.0

Example: Twinkle

Listing 1.20a Putting "Twinkle" all together, with two waves!

```
// Twinkle, with two oscillators!
SinOsc s => dac;
TriOsc t => dac;

// our main pitch variable
110.0 => float melody;

// gain control for our triangle wave melody
0.3 => float onGain;

// we'll use this for our on and off times
0.3 :: second => dur myDur;
```

1. Sine wave oscillator.
2. Another oscillator (triangle wave).
3. Sets initial pitch.
4. Gains for note on.
5. Notes duration.

Listing 1.20b Putting "Twinkle" all together (part B, Sweeping Upward)

```
// only play t at first, sweeping pitch upward
onGain => t.gain;
0 => s.gain;
while (melody < 220.0) {
    melody => t.freq;
    1.0 +=> melody;
    0.01 :: second => now;
}
```

6. Turns on triangle oscillator.
7. Turns on sine osc.
8. Loops until pitch reaches 220.
9. Steps up pitch by 1 Hz.
10. Every 1/100 of a second.

Example: Twinkle

Listing 1.20c Putting "Twinkle" all together, first "Twinkle"

```
// turn both on, set up the pitches
0.7 => s.gain;
110 => s.freq;

// play two notes, 1st "Twinkle"
for (0 => int i; i < 2; i++)  {
    onGain => t.gain;
    myDur => now;
    0 => t.gain;
    myDur => now;
}
```

11 Now turn on sin osc too.

12 ...and initialize its pitch.

13 Use a for loop to play two notes.

14 Turn on triangle.

15 Let note play.

16 Turn off triangle.

17 Silence to separate notes.

Listing 1.20d Putting "Twinkle" all together , the second "twinkle"

```
// new pitches!
138.6 => s.freq;
1.5*melody => t.freq;

// two more notes, 2nd "twinkle"
for (0 => int i; i < 2; i++)  {
    onGain => t.gain;
    myDur => now;
    0 => t.gain;
    myDur => now;
}
```

18 Sets up next "twinkle" frequency.

19 Plays that twice (for loop).

Example: Twinkle

Listing 1.20e Putting "Twinkle" all together, playing "little" and "star"

```
// pitches for "little"
146.8 => s.freq;
1.6837 * melody => t.freq;

// play 2 notes for "little"
for (0 => int i; i < 2; i++)  {
    onGain => t.gain;
    myDur => now;
    0 => t.gain;
    myDur => now;
}

// set up and play "star!"
138.6 => s.freq;
1.5*melody => t.freq;

onGain => t.gain;
second => now;
```

(20) Sets up next frequency for "little".

(21) Plays that twice (for loop).

(22) Sets up next frequency for "star".

(23) Plays that note...

(24) ...for a second.

Listing 1.20f Putting "Twinkle" all together

```
// end by sweeping both oscillators down to zero
for (330 => int i; i > 0; i--)  {
    i => t.freq;
    i*1.333 => s.freq;
    0.01 :: second => now;
}
```

(25) Uses a for loop to sweep down from 330 Hz.

(26) Updates every 1/100 of a second.

Exercise

Try changing the number of times the notes are played by changing the conditions of the `for` loops (try `i < 3` or 4, or more). Try changing the increment ❾ (listing 1.20b) and decrement ❷❺ amounts (listing 1.20f) for your pitch-sweeping loops. Experiment!