Instrumentalmusik und Live-Elektronik

LVA 17.0078

Week 3

Intro to programming

- functions
- arrays
- loops

functions

Functions have:

- name: how you call a function (e.g. myFunction)
- parameters/ arguments: things you give to the function to compute (e.g. Math.sine(x): you are giving x as a parameter)
- return type: the type a function gives you back. Some functions compute a value for you (e.g. Math.sin(x) gives you a flaoting point value, the sine of x.)
some functions do something and you don't need a value back (e.g. setTempo(...) would return type void.

Declaring your own function:

fun *return_type function_name(param1_type param1_name, param2_type param2_name, ...)* {

//function body goes here

return *some_object_of_return_type;* //or just return; if return_type is "void"

}

functions

Calling functions:

if you call a function you must have already defined and declared your function as in previous page. Then you have to options to call it:

Syntax option 1:
function_name(param1_value, param2_value,...);

Syntax option 2:
(param1_value, *param2_value, ...) => function_name;*

If the function is associated with an object or class use the function or class name before the function and attach it with a ".".
e.g. s.freq() or Std.sin(x)

functions

Parameters and scope

The parameter names you use in your function declaration will be treated as local variables within the function. Remember learning about local variables inside loops, where something declared within a set of {} brackets isn't accessible from outside those brackets? This is the same thing, except you don't have to declare the parameters separately (the function declaration takes care of this). Of course, you're free to declare other local variables within the function.

```
fun int add(int x, int y) {
int z;
x + y => z; return z;
}
```

x, y, and z will not be accessible anywhere outside this function.

functions

Masking

What if the name of a global variable is the same as the name of a function parameter or other local variable? This is legal, but it causes the local variable to mask the global variable. (In fact, if you have nested scopes (nested {} sets), the inner-most variable of that name will be used.)

example:

```
0 => int x; {
5 => int x;
<<< x >>>; }
<<< x >>>;
```

Will print out:

5 :(int)
0 :(int)

arrays

array

An array is a list of objects of the same type (e.g., a list of ints, a list of floats, or a list of SinOscs). There may be 0 or more objects in the array.

- Declaring without assigning a value

```
//array
int x;
float y;

//array
int x[1];

float y[5];
```

arrays

- Declaring and assigning a value (instantiation) at the same time

// single variable

```
0 => int x;

5.0 => float y;
```

// array

```
[0,1,2,5,7,6]  @=> int x[];

[5.0,19.0,23.92] @=> float y[];
```

- Assigning a new value

// single variable

```
0 => int x;

10 => x;
```

// array

```
[0,1,2]  @=> int x[];

[7,6]  @=> x;

//illegal [1.2,2.5]  @=> x;
```

arrays

You can assign an array variable using @=> as above. But if you want to access the individual elements (e.g., the int values themselves), you have to use an index. This index indicates the "chunk number" of the element we want in the array, starting with 0. That is, x[0] is always the first element in the array, and x[n−1] is the last for an array of size n.

```
// single variable
SinOsc s;
50 => s.freq;
// array
[0,1,2,5,7,6]  @=> int x[];
10 + x[1] => x[2];
//use => instead of @=> for assigning primitive types
SinOsc s[2];
50 => s[0].freq;
```

arrays

You can get the size of an array by using the .size() method. For now, we're going to be dealing with static arrays, so use .size() just to get the size and not to set it.

```
[0,1,2,5,44,12]  @=> int x[];
<<< x.size() >>>; //prints out 6
100 => x[x.size()-1];//sets the last element of array x to
100 instead of 12
```

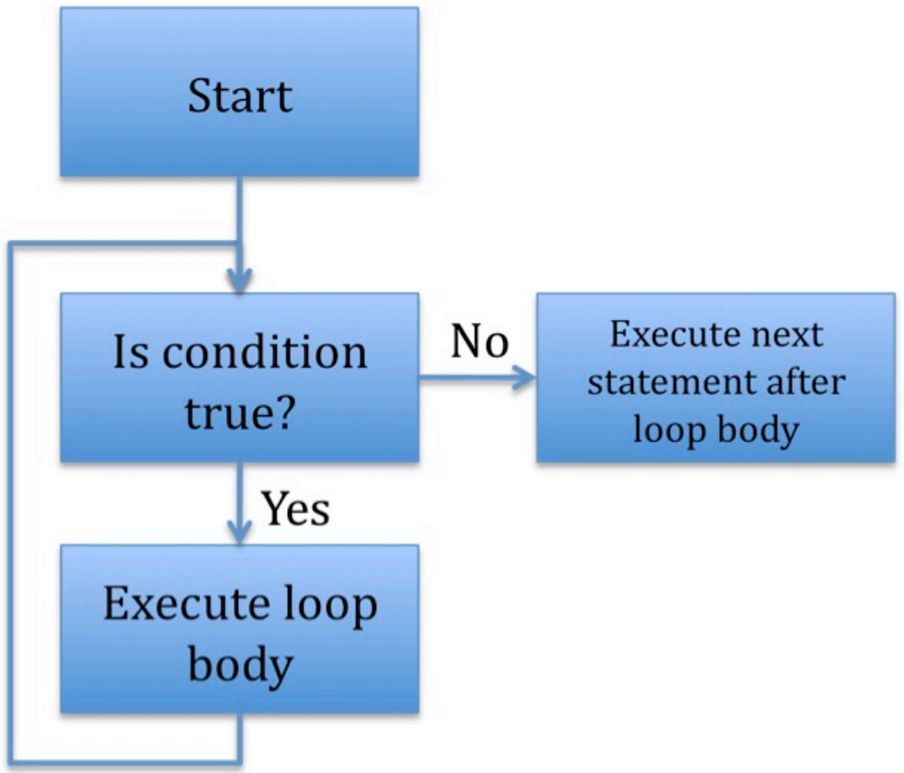
If you try to access an array element that doesn't exist (with an index that is too high), ChucK will crash and print out a message "ArrayOutOfBounds."

loops

While loops:

All while loops have the form:

```
while (condition) {
        do stuff
}
```



condition is a boolean expression that evaluates to true or false.

```
[0]  @=> int i;
while (i< 10) {
        <<< i >>>;
        i++;
}
```

loops

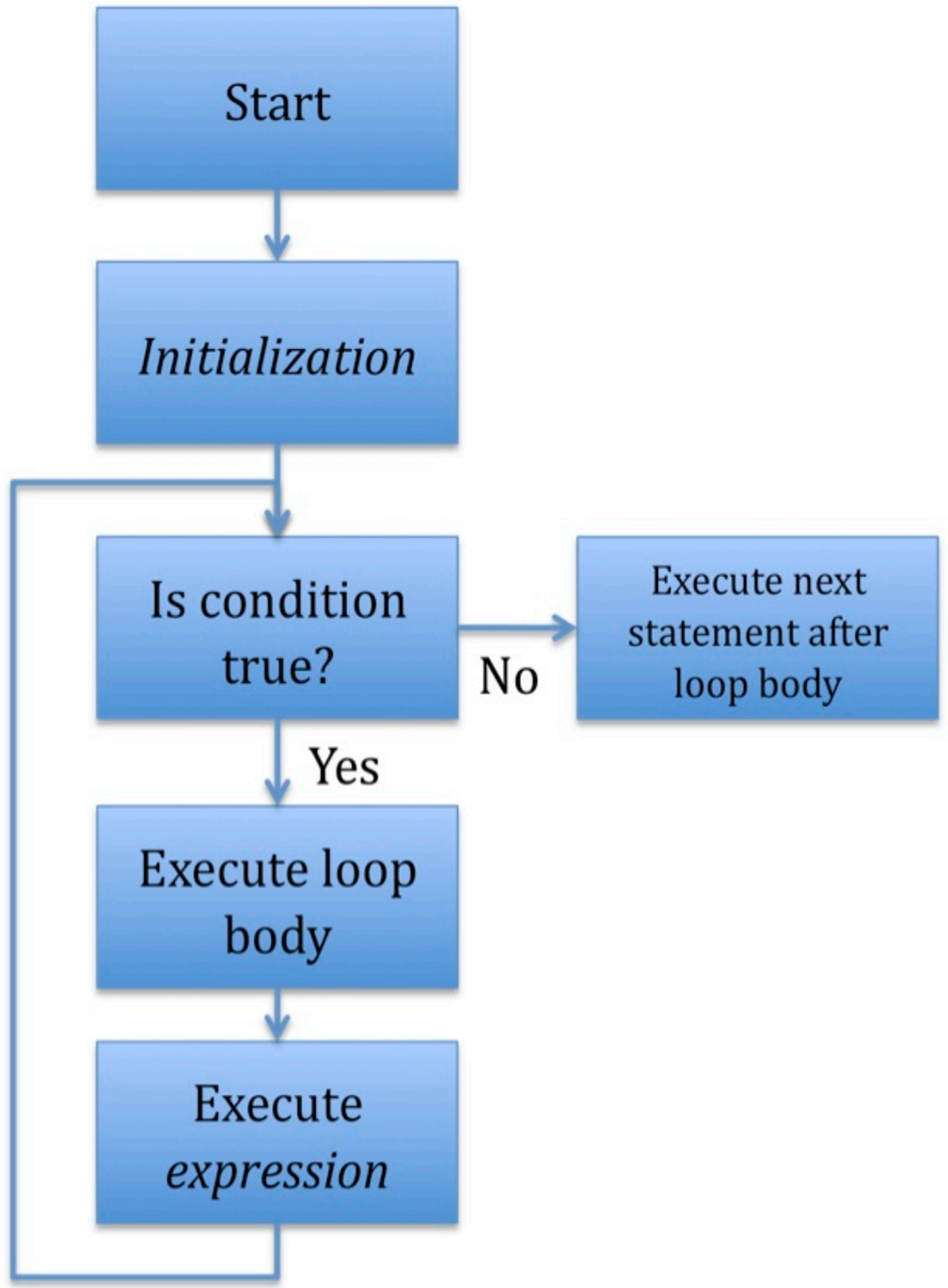
for loops:

All while loops have the form:

for (*initialization; condition; expression*) {

    do stuff

}



loops

The condition and loop body play the same rolls as in while loops. Here, though, the initialization is typically used to initialize a "counter" variable before the first iteration of the loop, and the expression is typically used to increment this counter after each loop iteration. For−loops can accomplish exactly the same set of tasks as while−loops, but they're most useful (and most often used) to execute something a set number of times. (Leaving out the initialization and expression is equivalent to using a while loop.)

```
[10,20,30,40]  @=> int x[];
for (0 => int i; i< x.size(); i++) {
      <<< x[i] >>>;
}
```