Most loops you write will be over lists of data. For such cases, it's simplest to use **map()** and **foreach()**.

But, occasionally, you will need to write a loop that is not scanning over a list. For that, you use the more primitive **while** and **for** looping forms.

while loops

Consider the example of a lottery program that wants to let users enter one or more numbers in the daily lottery. We want code to let a user keep entering numbers until the user says stop.

This kind of loop is traditionally written using a **while**. A **while** loop has a very simple form:

```
while (test) { statements }
```

The **while** first executes the *test*. If it returns a true value, then it executes the *statements* and goes back to try the *test*. The loop will continue until some statement does something to make the *test* false. If JavaScript, false means **false**, **0**, **null**, **undefined**, **NaN**, or an empty string.

The following code demonstrates getting lottery numbers.

We first define a function **getNumber()** to ask for a number using **prompt(** *message* **)**. **prompt()** pops up a dialog with the message, an input field, and two buttons: cancel and OK. If the user presses cancel, **null** is returned. If the user presses OK, then whatever text the user entered in the input field is returned as a string. The code for **getNumber()** will return -1 if the user presses cancel, otherwise the numeric value of the string entered. (Using **+** with a string is a shorthand way to convert strings to numbers.). This code does not try to handle if the user enters non-numeric values or negatives. In a real application, the user interface would make this impossible.

```
function getNumber(msg) {
 var answer = prompt(msg);
 return answer == null ? -1 : + answer;
}
```

Now that we have a way to read a number, we can write a **while** loop to get a list of numbers. We start by asking for the first number. Since **getNumber()** will return -1 or 0 when the user stops or cancels, the **while** loop tests that the user enter something greater than 0. Inside the **while** loop, the code saves the number entered and asks for another. When the user finally stops, the conditional checks to see whether the user canceled or simply stopped entering numbers. It returns an empty list of numbers if the user canceled, and the list of numbers entered otherwise.

```
function getNumbers() {
  var numbers = [];
  var number = getNumber('Enter a lottery number');
  while (number > 0) {
    numbers.push(number);
    number = getNumber('Enter another number, or 0 to stop, or Cancel to cancel all numbers');
  }
  if (number === -1) {
    return [];
  }
  else {
    return numbers;
  }
}
```

Notice that this code does not store the user's final entry in the list of numbers. Getting details like this correct is an important part of writing **while** loops.

for loops

The other most common kind of looping that does not involve lists is numeric iteration, i.e., doing something N times.

For example, suppose you wanted to create the HTML for a Sudoku puzzle application. Standard Sudoku uses a 9 x 9 grid. In HTML, such a grid would be 9 rows (**tr** elements), each containing 9 columns (**td** elements). While we could create these 81+ elements by hand, this seems like something better done by computer.

For doing something N time, you normally use a **for** loop. The syntax is a bit more complicated than **while**. The most commonly used pattern looks like this:

```
for (var variable = 0; variable < n; ++variable) { statements; }
```

The easiest way to understand what this form does is to translate it into the equivalent **while** version of the loop:

```
var variable = 0;
while (variable < n) {
  statements;
  ++variable;
}
```

Notice that the loop starts at 0, and stops when the variable is equal to N, i.e., it does not execute the statements when the variable is N. So, if N is 3, the statements will be executed for variable equal to 0, 1, and 2. This correctly executes the loop three times, but it does so counting from zero. Zero-based counting is standard in programming. It simplifies the arithmetic in certain situations, and it aligns with the zero-based indexed used with arrays. The only caution is that if you want to print numbers to users, you probably want to add one when printing, so that they see 1, 2, 3, ... and not 0, 1, 2, ...

Let's use **for** loops to create a 9 x 9 HTML table for a Sudoku puzzle. We'll assume that a CSS class has been defined to make the table have the size and borders we want.

```
function makeSudokuGrid() {
  var html = '<table class="sudoku-grid">';
  for (var row = 0; row < 9; ++row) {
    html += '<tr>';
    for (var col = 0; col < 9; ++col) {
      html += '<td></td>';
    }
    html += '</tr>';
  }
  html += '</table'>
  return html;
}
```

This is an example of a **nested for**. Nested **for** loops come up whenever you have two-dimensional data. The outer **for** loop creates 9 rows. The inner **for** loop creates 9 columns.