JavaScript Essentials

Loops

Table of Contents

Lesson Objectives

- Understand the importance of loops in Programming
- Able to use loops in JavaScript to complete repetitive tasks
- Understand the difference of for, while and do…while loop
- Know when to use for loop and when to use while and do…while loop

Section 1

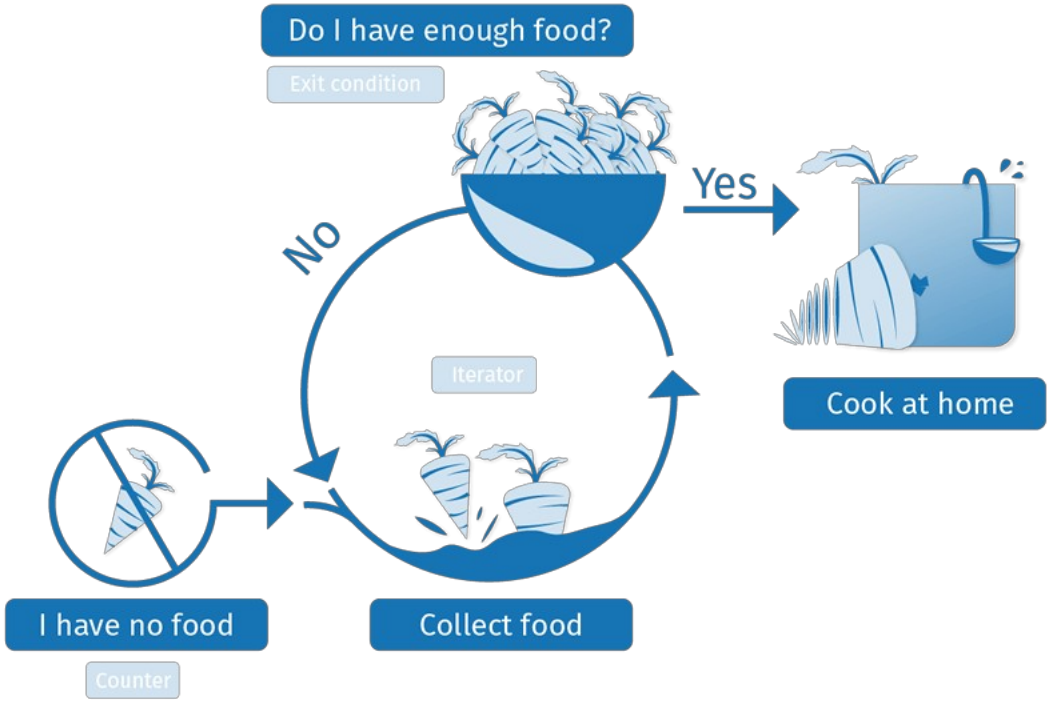
Overview

Overview

- Programming languages are very useful for rapidly completing repetitive tasks, from multiple basic calculations to just about any other situation where you've got a lot of similar items of work to complete. Here we'll look at the loop structures available in JavaScript that handle such needs.
- Loops, loops, loops. As well as being associated with popular breakfast cereals, roller coasters, and musical production , they are also a critical concept in programming. Programming loops are all to do with doing the same thing over and over again — which is termed **iteration** in programming speak.

Overview

- Let's consider the case of a farmer that is making sure he has enough food to feed his family for the week. He might use the following loop to achieve this:



Overview

A loop usually has one or more of the following features:

• A **counter**, which is initialized with a certain value — this is the starting point of the loop ("Start: I have no food", above).

•An **exit condition**, which is the criteria under which the loop stops — usually the counter reaching a certain value. This is illustrated by "Have I got enough food?", above. Let's say he needs 10 portions of food to feed his family.

•An **iterator**, which generally increments the counter by a small amount on each successive loop, until it reaches the exit condition. We haven't explicitly illustrated this above, but we could think about the farmer being able to collect say 2 portions of food per hour. After each hour, the amount of food he has collected is incremented by two, and he checks whether he has enough food. If he has reached 10 portions (the exit condition), he can stop collecting and go home.

Overview

In pseudocode, this would look something like the following:

```
loop(food = 0; foodNeeded = 10) {
  if (food >= foodNeeded) {
    exit loop;
    // We have enough food; let's go home
  } else {
    food += 2; // Spend an hour collecting 2 more food
    // loop will then run again
  }
}
```

Overview – Why loop?

- At this point, you probably understand the high-level concepts behind loops, but you are probably thinking "OK, great, but how does this help me write better JavaScript code?" As we said earlier, **loops are all to do with doing the same thing over and over again**, which is great for **rapidly completing repetitive tasks**.
- Often, the code will be slightly different on each successive iteration of the loop, which means that you can complete a whole load of tasks that are similar but slightly different — if you've got a lot of different calculations to do, you want to do each different one, not the same one over and over again!

Overview – Why loop?

Let's look at an example to perfectly illustrate why loops are such a good thing. Let's say we wanted to draw 100 random circles on a <canvas> element (press the *Update* button to run the example again and again to see different random sets):

Overview – Why loop?

You don't have to understand all the code for now, but let's look at the part of the code that actually draws the 100 circles:

```
for (let i = 0; i < 100; i++) {
  ctx.beginPath();
  ctx.fillStyle = 'rgba(255,0,0,0.5)';
  ctx.arc(random(WIDTH), random(HEIGHT), random(50), 0, 2 * Math.PI);
  ctx.fill();
}
```

Overview – Why loop?

- You should get the basic idea — we are using a loop to run 100 iterations of this code, each one of which draws a circle in a random position on the page. The amount of code needed would be the same whether we were drawing 100 circles, 1000, or 10,000. Only one number has to change.
- If we weren't using a loop here, we'd have to repeat the following code for every circle we wanted to draw:

```
ctx.beginPath();
ctx.fillStyle = 'rgba(255,0,0,0.5)';
ctx.arc(random(WIDTH), random(HEIGHT), random(50), 0, 2 * Math.PI);
ctx.fill();
```

Overview – Summary

- Loops are all to do with doing the same thing over and over again (which is termed **iteration** in programming speak)
- Loops are essentials for rapidly completing repetitive tasks
- Every time you do a same tasks over and over think about loops
- Remember the mind of Programmer: Don’t repeat – be lazy

Section 2

Famous for loop

Famous for loop

- Let's start exploring some specific loop constructs. The first, which you'll use most of the time, is the for loop — this has the following syntax:

```
for (initializer; exit-condition; final-expression) {
  // code to run
}
```

Famous for loop

- Let's look at a real example so we can visualize what these do more clearly.

```
1  const cats = ['Bill', 'Jeff', 'Pete', 'Biggles', 'Jasmin'];
2  let info = 'My cats are called ';
3  const para = document.querySelector('p');
4
5  for (let i = 0; i < cats.length; i++) {
6    info += cats[i] + ', ';
7  }
8
9  para.textContent = info;
```

Famous for loop

- This shows a loop is used to iterate over the items in an array and do something with each of them — a very common pattern in JavaScript. Here:

1. The counter variable (sometimes known as an initializer or an iteration variable), i, starts at 0 (let i = 0).
2. The loop has been told to run until i is no longer smaller than the length of the cats array. This is important — the exit condition shows the condition under which the loop will still run. So in this case, while i < cats.length is still true, the loop will still run.
3. Inside the loop, we concatenate the current loop item (cats[i] , which is cats[whatever i is at the time]) along with a comma and space, onto the end of the info variable. So:
    1. During the first run, i = 0, therefore cats[0] + ', ' (which is equal to Bill,) will be concatenated onto info.
    2. During the second run, i = 1, so cats[1] + ', ' (which is equal to Jeff,) will be concatenated onto info.
    3. And so on. After each time the loop has run, 1 will be added to i (i++), then the process will start again.
4. When i becomes equal to cats.length (in this case, 5), the loop will stop, and the browser will move on to the next bit of code below the loop.

Famous for loop – Common Mistakes

- We have made the exit condition i < cats.length, not **i <= cats.length**, because computers count from 0, not 1 — we are starting i at 0, and going up to i = 4 (the index of the last array item). cats.length returns 5, as there are 5 items in the array, but we don't want to get up to i = 5, as that would return undefined for the last item (there is no array item with an index of 5). So, therefore, we want to go up to 1 less than cats.length (i <), not the same as cats.length (i <=).
- A common mistake with **exit conditions** is making them use "equal to" **(===) rather than say "less than or equal to" (<=).** If we wanted to run our loop up to i = 5, the exit condition would need to be i <= cats.length. If we set it to i === cats.length, the loop would not run at all because i is not equal to 5 on the first loop iteration, so it would stop immediately.

Famous for loop

- One small problem we are left with is that the final output sentence isn't very well-formed
- Ideally, we want to change the concatenation on the final loop iteration so that we haven't got a comma on the end of the sentence. Well, no problem — we can quite happily insert a conditional inside our for loop to handle this special case:

```
1 for (let i = 0; i < cats.length; i++) {
2   if (i === cats.length - 1) {
3     info += 'and ' + cats[i] + '.';
4   } else {
5     info += cats[i] + ', ';
6   }
7 }
```

Famous for loop - Important

- With for — as with all loops — you must make sure that the initializer is incremented or, depending on the case, decremented, so that it eventually reaches the exit condition. If not, the loop will go on forever, and either the browser will force it to stop, or it will crash. This is called an **infinite loop**

Famous for loop - Overview

- Syntax for for loop is: for (initializer; exit-condition; final-expression) { // code to run }
- For loop is commonly used when you know exactly the number of iteration for example loop every item of an array (n iteration where n is the length of the array)
- Remember Computer starts a 0 so last item when you loop an array is n – 1 (where n is the length of the array)
- Take care of the loop condition or you will get an **Infinite loops** (which is very bad)

Section 3

Break the loops

Break the loops

- If you want to exit a loop before all the iterations have been completed, you can use the break statement.
- We already met this in the previous article when we looked at switch statements — when a case is met in a switch statement that matches the input expression, the break statement immediately exits the switch statement and moves onto the code after it.
- It's the same with loops — a break statement will immediately exit the loop and make the browser move on to any code that follows it.

Break the loops - Example

- Say we wanted to search through an array of contacts and telephone numbers and return just the number we wanted to find? First, some simple HTML — a text <input> allowing us to enter a name to search for, a <button> element to submit a search, and a <p> element to display the results in:

```
<label for="search">Search by contact name: </label>
<input id="search" type="text">
<button>Search</button>

<p></p>
```

Break the loops - Example

- Now on to the JavaScript:

```
const contacts = ['Chris:2232322', 'Sarah:3453456', 'Bill:7654322', 'Mary:9998769', 'Dianne:9384975
const para = document.querySelector('p');
const input = document.querySelector('input');
const btn = document.querySelector('button');

btn.addEventListener('click', function() {
  let searchName = input.value.toLowerCase();
  input.value = '';
  input.focus();
  for (let i = 0; i < contacts.length; i++) {
    let splitContact = contacts[i].split(':');
    if (splitContact[0].toLowerCase() === searchName) {
      para.textContent = splitContact[0] + '\'s number is ' + splitContact[1] + '.';
      break;
    } else {
      para.textContent = 'Contact not found.';
    }
  }
});
```

Break the loops - Overview

- A break statement will immediately exit the loop and make the browser move on to any code that follows it.
- Use break when you want to loop a partition of the iterations for example: loop first half of an array
- Use break outside of loops does not make sense

Section 4

Skip iterations with continue

Skip iterations with continue

- The continue statement works in a similar manner to break, but instead of breaking out of the loop entirely, it skips to the next iteration of the loop. Let's look at another example that takes a number as an input, and returns only the numbers that are squares of integers (whole numbers).

Skip iterations with continue

- The HTML is basically the same as the last example — a simple text input, and a paragraph for output. The JavaScript is mostly the same too, although the loop itself is a bit different:

```
let num = input.value;

for (let i = 1; i <= num; i++) {
  let sqRoot = Math.sqrt(i);
  if (Math.floor(sqRoot) !== sqRoot) {
    continue;
  }

  para.textContent += i + ' ';
}
```

Skip iterations with continue - Summary

- To skips to the next iteration of the loop use continue
- continue is commonly used with conditionals statement such as if…else
- Use continue outside of loop makes no sense

Section 5

while and do…while

while and do…while

- for is not the only type of loop available in JavaScript.
- There are actually many others and, while you don't need to understand all of these now, it is worth having a look at the structure of a couple of others so that you can recognize the same features at work in a slightly different way.

while and do…while

- First, let's have a look at the while loop. This loop's syntax looks like so:

```
initializer
while (exit-condition) {
  // code to run

  final-expression
}
```

while and do…while

- This works in a very similar way to the for loop, except that the initializer variable is set before the loop, and the final-expression is included inside the loop after the code to run — rather than these two items being included inside the parentheses.
- The exit-condition is included inside the parentheses, which are preceded by the while keyword rather than for.

while and do…while

- The same three items are still present, and they are still defined in the same order as they are in the for loop
- This makes sense, as you still have to have an initializer defined before you can check whether it has reached the exit-condition;
- The final-condition is then run after the code inside the loop has run (an iteration has been completed), which will only happen if the exit-condition has still not been reached.

while and do…while

- Let's have a look again at our cats list example, but rewritten to use a while loop:

```
let i = 0;

while (i < cats.length) {
  if (i === cats.length - 1) {
    info += 'and ' + cats[i] + '.';
  } else {
    info += cats[i] + ', ';
  }

  i++;
}
```

while and do…while

- The do...while loop is very similar, but provides a variation on the while structure:

```
initializer
do {
  // code to run

  final-expression
} while (exit-condition)
```

while and do…while

- In this case, the initializer again comes first, before the loop starts.
- The keyword directly precedes the curly braces containing the code to run and the final expression.
- The differentiator here is that the exit-condition comes after everything else, wrapped in parentheses and preceded by a while keyword.
- In a do...while loop, the code inside the curly braces is always run once before the check is made to see if it should be executed again (in while and for, the check comes first, so the code might never be executed).

while and do…while

- Let's rewrite our cat listing example again to use a do...while loop:

```
let i = 0;

do {
  if (i === cats.length - 1) {
    info += 'and ' + cats[i] + '.';
  } else {
    info += cats[i] + ', ';
  }

  i++;
} while (i < cats.length);
```

while and do…while - Overview

- while and do…while have very similar syntax
- The differentiator here is that the exit-condition comes after everything else, wrapped in parentheses and preceded by a while keyword
- With while and do...while — as with all loops — you must make sure that the initializer is incremented or, depending on the case, decremented, so that it eventually reaches the exit condition. If not, the loop will go on forever, and either the browser will force it to stop, or it will crash. This is called an **infinite loop**

Section 6

Which loop should i use?

Which loop should i use?

- For basic uses, for, while, and do...while loops are largely interchangeable.
- They can all be used to solve the same problems, and which one you use will largely depend on your personal preference — which one you find easiest to remember or most intuitive.

Which loop should i use?

- First for:

```
for (initializer; exit-condition; final-expression) {
  // code to run
}
```

Which loop should i use?

- while:

```
initializer
while (exit-condition) {
  // code to run

  final-expression
}
```

Which loop should i use?

- do…while:

```
initializer
do {
  // code to run

  final-expression
} while (exit-condition)
```

Which loop should i use?

- We would recommend for, at least to begin with, as it is probably the easiest for remembering everything — the initializer, exit-condition, and final-expression all have to go neatly into the parentheses, so it is easy to see where they are and check that you aren't missing them.

Which loop should i use? - Overview

- For basic uses, **for**, **while**, and **do...while** loops are largely interchangeable.
- They can all be used to solve the same problems, and which one you use will largely depend on your personal preference — which one you find easiest to remember or most intuitive.
- **for** is the recommended loop for beginner

Thank you

Q&A