Building Blocks Of JavaScript

thruskills

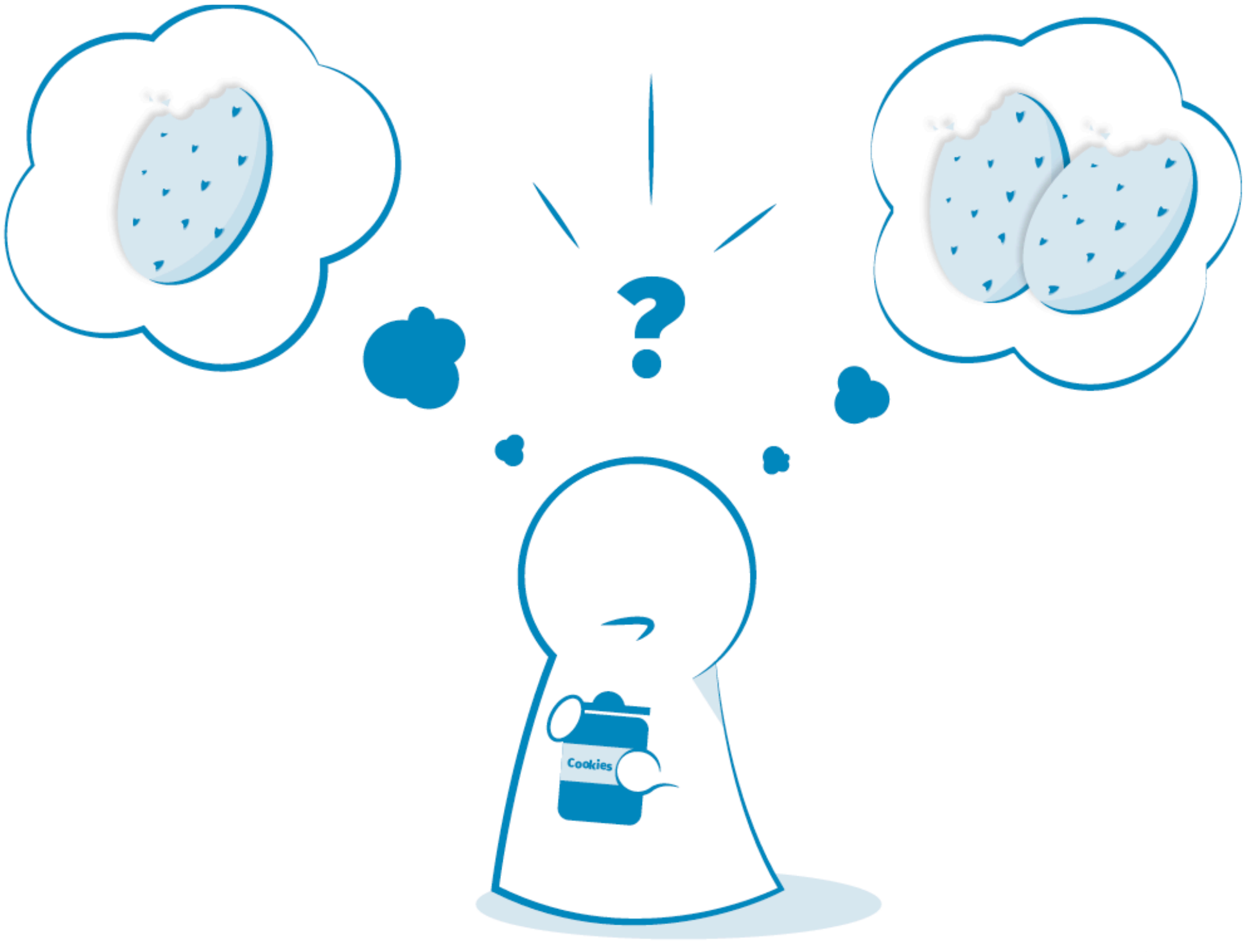
https://www.thruskills.com

+91 831 737 5392

What you will learn today

- Making decisions in your code — conditionals
- Looping code
- Functions — reusable blocks of code
- Build your own function
- Function return values
- Introduction to events

Conditional Statements



if ... else statements

```
if (condition) {
  code to run if condition is true
} else {
  run some other code instead
}
```

```
if (condition) {
  code to run if condition is true
}

run some other code
```

```
if (condition) code to run if condition is true
else run some other code instead
```

```
var shoppingDone = false;

if (shoppingDone === true) {
  var childsAllowance = 10;
} else {
  var childsAllowance = 5;
}
```

else if

```
var select = document.querySelector('select');
var para = document.querySelector('p');

select.addEventListener('change', setWeather);

function setWeather() {
  var choice = select.value;

  if (choice === 'sunny') {
    para.textContent = 'It is nice and sunny outside today. Wear shorts! Go to the beach, or the
  } else if (choice === 'rainy') {
    para.textContent = 'Rain is falling outside; take a rain coat and a brolly, and don\'t stay
  } else if (choice === 'snowing') {
    para.textContent = 'The snow is coming down — it is freezing! Best to stay in with a cup of
  } else if (choice === 'overcast') {
    para.textContent = 'It isn\'t raining, but the sky is grey and gloomy; it could turn any minu
  } else {
    para.textContent = '';
  }
}
```

switch

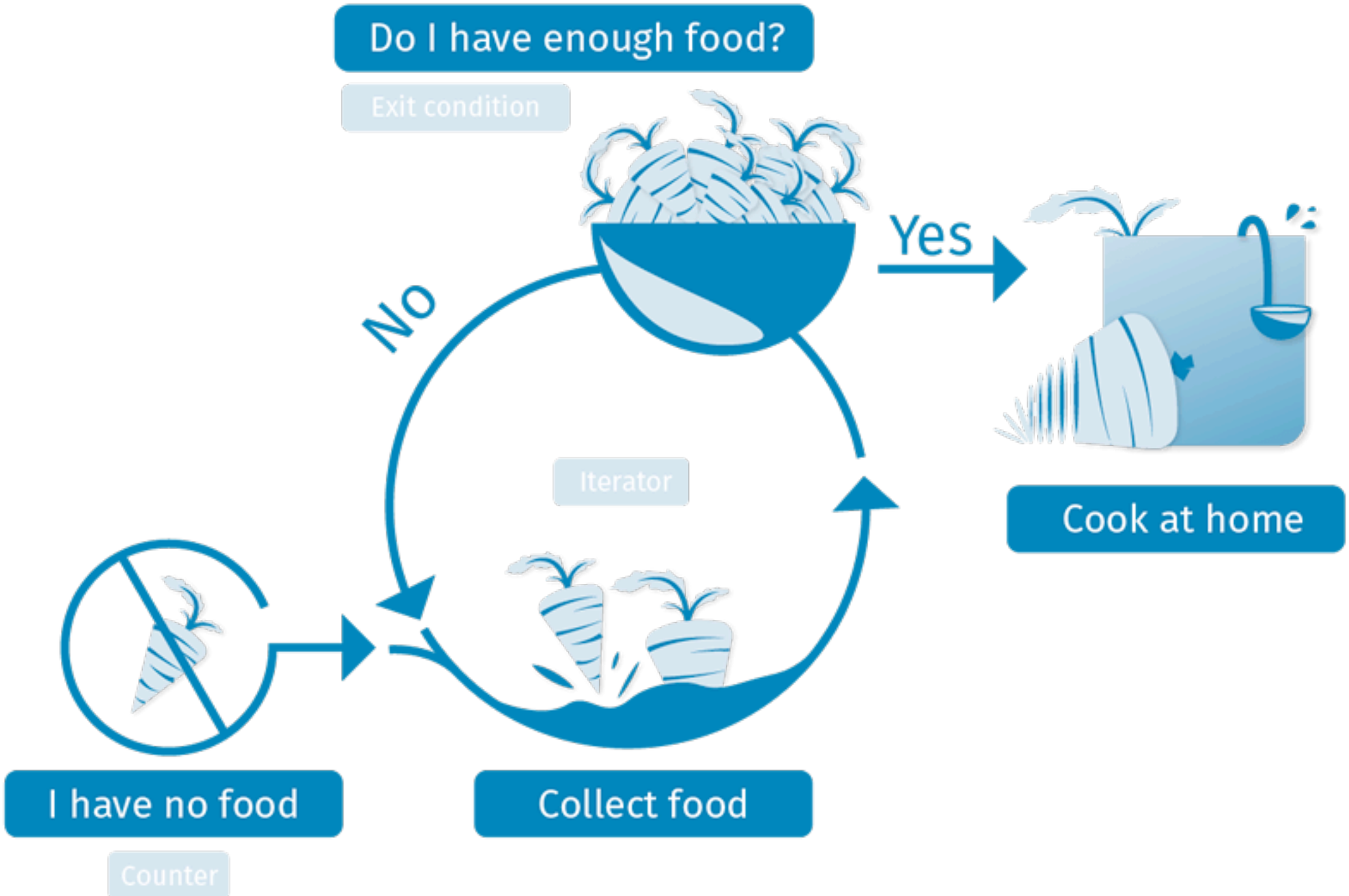
```
switch (expression) {
  case choice1:
    run this code
    break;

  case choice2:
    run this code instead
    break;

  // include as many cases as you like

  default:
    actually, just run this code
}
```

ternary operator

```
( condition ) ? run this code : run this code instead
```

var greeting = (isBirthday) ? 'Happy birthday Mrs. Smith — we hope you have a great day!' : 'Good morning Mrs. Smith.';

Keep me in the loop...



pseudocode

```
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
  }
}
```

The standard for loop

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

```
var cats = ['Bill', 'Jeff', 'Pete', 'Biggles', 'Jasmin'];
var info = 'My cats are called ';
var para = document.querySelector('p');

for (var i = 0; i < cats.length; i++) {
  info += cats[i] + ', ';
}

para.textContent = info;
```

```
var num = input.value;

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

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

for … each

```
for each (variable in object) {
  statement
}
```

```
var sum = 0;
var obj = {prop1: 5, prop2: 13, prop3: 8};

for each (var item in obj) {
 sum += item;
}

console.log(sum); // logs "26", which is 5+13+8
```

while and do ... while

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

  final-expression
}
```

```
var i = 0;

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

  i++;
}
```

```
initializer
do {
  // code to run

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

break & continue

```
1  var i = 0;
2
3  while (i < 6) {
4    if (i === 3) {
5      break;
6    }
7    i = i + 1;
8  }
9
10 console.log(i);
```

```
1  var text = "";
2
3  for (var i = 0; i < 10; i++) {
4    if (i === 3) {
5      continue;
6    }
7    text = text + i;
8  }
9
10 console.log(text);
```

Functions

- Built-in browser functions
- Functions versus methods
- Custom functions
- Invoking functions
- Anonymous functions
- Function parameters
- Function scope and conflicts
- Functions inside functions

Built-in browser functions

```
var myText = 'I am a string';
var newString = myText.replace('string', 'sausage');
console.log(newString);
// the replace() string function takes a string,
// replaces one substring with another, and returns
// a new string with the replacement made
```

```
var myArray = ['I', 'love', 'chocolate', 'frogs'];
var madeAString = myArray.join(' ');
console.log(madeAString);
// the join() function takes an array, joins
// all the array items together into a single
// string, and returns this new string
```

```
var myNumber = Math.random();
// the random() function generates a random
// number between 0 and 1, and returns that
// number
```

Functions versus methods

One thing we need to clear up before we move on — technically speaking, built in browser functions are not functions — they are **methods**. This sounds a bit scary and confusing, but don't worry — the words function and method are largely interchangeable, at least for our purposes, at this stage in your learning.

The distinction is that methods are functions defined inside objects. Built-in browser functions (methods) and variables (which are called properties) are stored inside structured objects, to make the code more efficient and easier to handle.

You don't need to learn about the inner workings of structured JavaScript objects yet — you can wait until our later module that will teach you all about the inner workings of objects, and how to create your own. For now, we just wanted to clear up any possible confusion of method versus function — you are likely to meet both terms as you look at the available related resources across the Web.

Custom functions

```
function draw() {
  ctx.clearRect(0,0,WIDTH,HEIGHT);
  for (var 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();
  }
}
```

```
draw();
```

```
function random(number) {
  return Math.floor(Math.random()*number);
}
```

Invoking functions

You are probably clear on this by now, but just in case ... to actually use a function after it has been defined, you've got to run — or invoke — it. This is done by including the name of the function in the code somewhere, followed by parentheses.

```
function myFunction() {
  alert('hello');
}

myFunction()
// calls the function once
```

Anonymous functions

```
1 function myFunction() {
2   alert('hello');
3 }
```

```
1 function() {
2   alert('hello');
3 }
```

```
1 var myButton = document.querySelector('button');
2
3 myButton.onclick = function() {
4   alert('hello');
5 }
```

```
1 myButton.onclick = function() {
2   alert('hello');
3   // I can put as much code
4   // inside here as I want
5 }
```

Function parameters

Some functions require parameters to be specified when you are invoking them — these are values that need to be included inside the function parentheses, which it needs to do its job properly.

```
var myNumber = Math.random();
```

```
var myText = 'I am a string';
var newString = myText.replace('string', 'sausage');
```

```
var myArray = ['I', 'love', 'chocolate', 'frogs'];
var madeAString = myArray.join(' ');
// returns 'I love chocolate frogs'
var madeAString = myArray.join();
// returns 'I,love,chocolate,frogs'
```

Function scope and conflicts

Let's talk a bit about **scope** — a very important concept when dealing with functions. When you create a function, the variables and other things defined inside the function are inside their own separate scope, meaning that they are locked away in their own separate compartments, unreachable from inside other functions or from code outside the functions.

The top level outside all your functions is called the **global** scope. Values defined in the global scope are accessible from everywhere in the code.

JavaScript is set up like this for various reasons — but mainly because of security and organization. Sometimes you don't want variables to be accessible from everywhere in the code — external scripts that you call in from elsewhere could start to mess with your code and cause problems because they happen to be using the same variable names as other parts of the code, causing conflicts. This might be done maliciously, or just by accident.

Function scope and conflicts ...

```
<!-- Excerpt from my HTML -->
<script src="first.js"></script>
<script src="second.js"></script>
<script>
  greeting();
</script>
```

```
// first.js
var name = 'Chris';
function greeting() {
  alert('Hello ' + name + ': welcome to our company.');
}
```

```
// second.js
var name = 'Zaptec';
function greeting() {
  alert('Our company is called ' + name + '.');
}
```

Functions inside functions

Keep in mind that you can call a function from anywhere, even inside another function. This is often used as a way to keep code tidy — if you have a big complex function, it is easier to understand if you break it down into several sub-functions:

```
1  function myBigFunction() {
2    var myValue;
3
4    subFunction1();
5    subFunction2();
6    subFunction3();
7  }
8
9  function subFunction1() {
10   console.log(myValue);
11 }
12
13 function subFunction2() {
14   console.log(myValue);
15 }
16
17 function subFunction3() {
18   console.log(myValue);
19 }
```

Function return values

What are return values?

Return values are just what they sound like — values returned by the function when it completes. You've already met return values a number of times, although you may not have thought about them explicitly. Let's return to some familiar code:

```
var myText = 'I am a string';
var newString = myText.replace('string', 'sausage');
console.log(newString);
// the replace() string function takes a string,
// replaces one substring with another, and returns
// a new string with the replacement made
```

Events

- Built-in browser functions
- Functions versus methods
- Custom functions
- Invoking functions
- Anonymous functions
- Function parameters
- Function scope and conflicts
- Functions inside functions

A series of fortunate events

Events

Events are actions or occurrences that happen in the system you are programming — the system will fire a signal of some kind when an event occurs, and also provide a mechanism by which some kind of action can be automatically taken (e.g. some code running) when the event occurs.

For example in an airport when the runway is clear for a plane to take off, a signal is communicated to the pilot, and as a result they commence piloting the plane.

In the case of the Web, events are fired inside the browser window, and tend to be attached to a specific item that resides in it — this might be a single element, set of elements, the HTML document loaded in the current tab, or the entire browser window.

Events

There are a lot of different types of event that can occur, for example:

- The user clicking the mouse over a certain element, or hovering the cursor over a certain element.
- The user pressing a key on the keyboard.
- The user resizing or closing the browser window.
- A web page finishing loading.
- A form being submitted.
- A video being played, or paused, or finishing play.
- An error occurring.

Events… a simple example

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Test Event</title>
</head>
<body>
    <button>Change color</button>
</body>
</html>
```

```
var btn = document.querySelector('button');

function random(number) {
  return Math.floor(Math.random()*(number+1));
}

btn.onclick = function() {
  var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random
  (255) + ')';
  document.body.style.backgroundColor = rndCol;
}
```

Events... its not just web pages

Another thing worth mentioning at this point is that events are not particular to JavaScript — most programming languages have some kind of event model, and the way it works will often differ from JavaScript's way. In fact, the event model in JavaScript for web pages differs from the event model for JavaScript as it is used in other environments.

For example, Node.js is a very popular JavaScript runtime that enables developers to use JavaScript to build network and server-side applications. The Node.js event model relies on listeners to listen for events and emitters to emit events periodically — it doesn't sound that different, but the code is quite different, making use of functions like on() to register an event listener, and once() to register an event listener that unregisters after it has run once.

```
const EventEmitter = require('events');
const util = require('util');

function MyEmitter() {
  EventEmitter.call(this);
}
util.inherits(MyEmitter, EventEmitter);

const myEmitter = new MyEmitter();
myEmitter.on('event', () => {
  console.log('an event occurred!');
});
myEmitter.emit('event');
```

Event handler properties

There are a number of different ways in which you can add event listener code to web pages so that it will be run when the associated event fires.

```
var btn = document.querySelector('button');

btn.onclick = function() {
  var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';
  document.body.style.backgroundColor = rndCol;
}
```

```
var btn = document.querySelector('button');

function bgChange() {
  var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';
  document.body.style.backgroundColor = rndCol;
}

btn.onclick = bgChange;
```

Inline event handlers — don't use these

The earliest method of registering event handlers found on the Web involved event handler HTML attributes (aka inline event handlers) like the one shown above — the attribute value is literally the JavaScript code you want to run when the event occurs. The above example invokes a function defined inside a <script> element on the same page, but you could also insert JavaScript directly inside the attribute, for example:

```
<button onclick="bgChange()">Press me</button>
```

```
function bgChange() {
  var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';
  document.body.style.backgroundColor = rndCol;
}
```

```
<button onclick="alert('Hello, this is my old-fashioned event handler!');">Press me</button>
```

```
var buttons = document.querySelectorAll('button');

for (var i = 0; i < buttons.length; i++) {
  buttons[i].onclick = bgChange;
}
```

addEventListener() & removeEventListener()

The newest type of event mechanism is defined in the Document Object Model (DOM) Level 2 Events Specification, which provides browsers with a new function — addEventListener(). This functions in a similar way to the event handler properties, but the syntax is obviously different. We could rewrite our random color example to look like this:

```
var btn = document.querySelector('button');

function bgChange() {
  var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';
  document.body.style.backgroundColor = rndCol;
}

btn.addEventListener('click', bgChange);
```

```
btn.addEventListener('click', function() {
  var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';
  document.body.style.backgroundColor = rndCol;
});
```

```
btn.removeEventListener('click', bgChange);
```

```
myElement.onclick = functionA;
myElement.onclick = functionB;
```

```
myElement.addEventListener('click', functionA);
myElement.addEventListener('click', functionB);
```

What mechanism should I use?

Of the three mechanisms, you definitely shouldn't use the HTML event handler attributes — these are outdated, and bad practice, as mentioned above.

The other two are relatively interchangeable, at least for simple uses:

- Event handler properties have less power and options, but better cross-browser compatibility (being supported as far back as Internet Explorer 8). You should probably start with these as you are learning.
- DOM Level 2 Events (addEventListener(), etc.) are more powerful, but can also become more complex and are less well supported (supported as far back as Internet Explorer 9). You should also experiment with these, and aim to use them where possible.

Event objects

Sometimes inside an event handler function, you might see a parameter specified with a name such as event, evt, or simply e. This is called the event object, and it is automatically passed to event handlers to provide extra features and information. For example, let's rewrite our random color example again slightly:

```
function bgChange(e) {
  var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';
  e.target.style.backgroundColor = rndCol;
  console.log(e);
}

btn.addEventListener('click', bgChange);
```

```
var divs = document.querySelectorAll('div');

for (var i = 0; i < divs.length; i++) {
  divs[i].onclick = function(e) {
    e.target.style.backgroundColor = bgChange();
  }
}
```

Preventing default behaviour

Sometimes, you'll come across a situation where you want to stop an event doing what it does by default. The most common example is that of a web form submit.

```
<form>
  <div>
    <label for="fname">First name: </label>
    <input id="fname" type="text">
  </div>
  <div>
    <label for="lname">Last name: </label>
    <input id="lname" type="text">
  </div>
  <div>
    <input id="submit" type="submit">
  </div>
</form>
<p></p>
```

```
var form = document.querySelector('form');
var fname = document.getElementById('fname');
var lname = document.getElementById('lname');
var submit = document.getElementById('submit');
var para = document.querySelector('p');

form.onsubmit = function(e) {
  if (fname.value === '' || lname.value === '') {
    e.preventDefault();
    para.textContent = 'You need to fill in both names!';
  }
}
```

Event bubbling and capture

The final subject to cover here is something that you'll not come across often, but it can be a real pain if you don't understand it. Event bubbling and capture are two mechanisms that describe what happens when two handlers of the same event type are activated on one element. Let's look at an example to make this easier — open up the show-video-box.html example in a new tab (and the source code in another tab.) It is also available live below:

```
<button>Display video</button>

<div class="hidden">
  <video>
    <source src="rabbit320.mp4" type="video/mp4">
    <source src="rabbit320.webm" type="video/webm">
    <p>Your browser doesn't support HTML5 video. Here is a <a href="rabbit320.mp4">link to the video</a> inste
  </video>
</div>
```

```
btn.onclick = function() {
  videoBox.setAttribute('class', 'showing');
}
```

```
videoBox.onclick = function() {
  videoBox.setAttribute('class', 'hidden');
};

video.onclick = function() {
  video.play();
};
```

Bubbling and capturing explained

When an event is fired on an element that has parent elements (e.g. the <video> in our case), modern browsers run two different phases — the capturing phase and the bubbling phase.

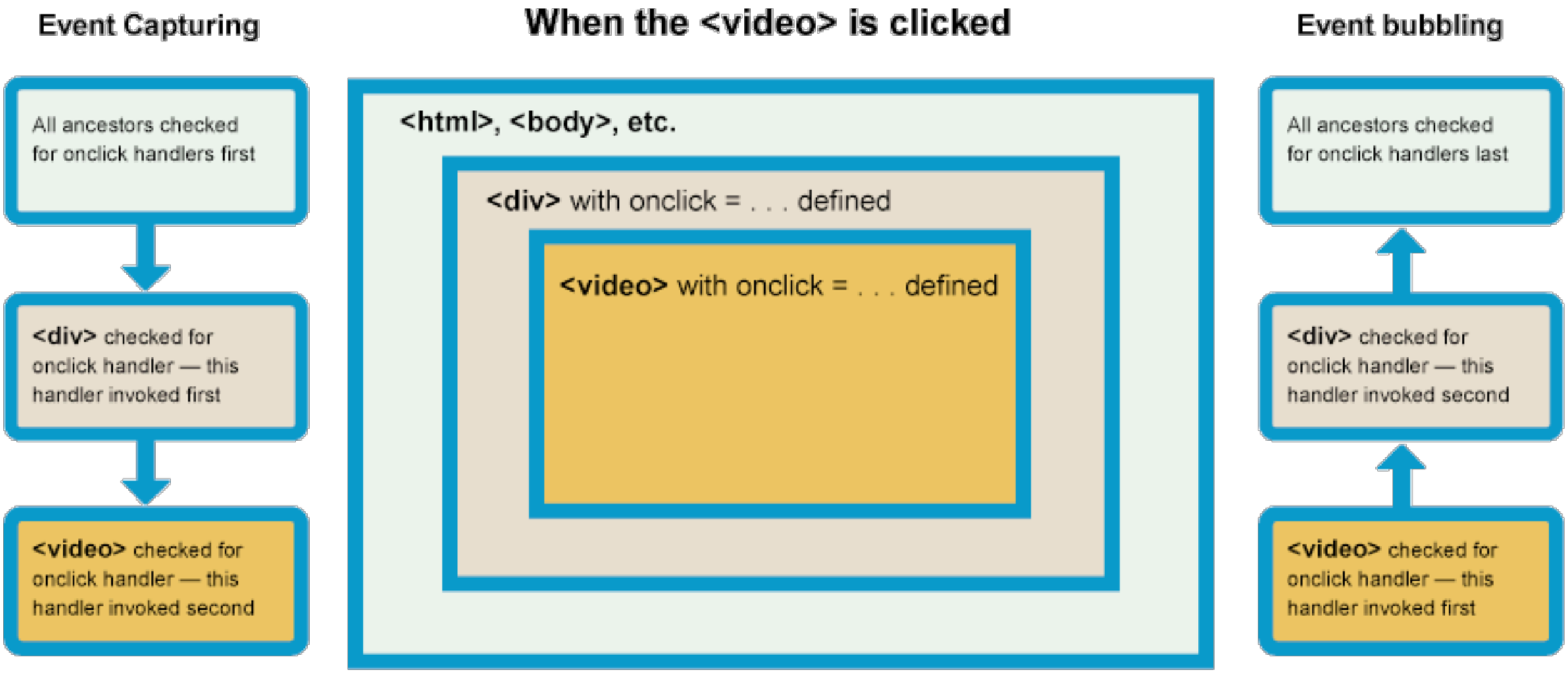
In the capturing phase:

- The browser checks to see if the element's outer-most ancestor (<html>) has an onclick event handler registered on it in the capturing phase, and runs it if so.
- Then it moves on to the next element inside <html> and does the same thing, then the next one, and so on until it reaches the element that was actually clicked on.

In the capturing phase:

- The browser checks to see if the element that was actually clicked on has an onclick event handler registered on it in the bubbling phase, and runs it if so.
- Then it moves on to the next immediate ancestor element and does the same thing, then the next one, and so on until it reaches the <html> element.

Bubbling and capturing explained



Fixing the problem with stopPropagation()

This is annoying behavior, but there is a way to fix it! The standard event object has a function available on it called stopPropagation(), which when invoked on a handler's event object makes it so that handler is run, but the event doesn't bubble any further up the chain, so no more handlers will be run.

```
1 video.onclick = function(e) {
2   e.stopPropagation();
3   video.play();
4 };
```

Event delegation

Bubbling also allows us to take advantage of event delegation — this concept relies on the fact that if you want some code to run when you click on any one of a large number of child elements, you can set the event listener on their parent and have the effect of the event listener bubble to each child, rather than having to set the event listener on every child individually.

A good example is a series of list items — if you want each one of them to pop up a message when clicked, you can set the click event listener on the parent <ul>, and it will bubble to the list items.

Any Questions

Thank You

https://www.thruskills.com

+91 831 737 5392

thruskills