JavaScript - Building Blocks

Making decisions in your code - conditionals

If you help me by going and doing the shopping, I'll give you some extra allowance so you can afford that toy you wanted." In JavaScript, we could represent this like so: if...else statements:

```
let shoppingDone = false;
let childsAllowance;

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

Also works like this:

```
let shoppingDone = false;
let childsAllowance;

// We don't need to explicitly specify 'shoppingDone === true'
if (shoppingDone) {
  childsAllowance = 10;
} else {
  childsAllowance = 5;
}
```

else if: Select the weather type today: {sunny, raining, snowing, overcast} If selects sunny: print(It is nice and sunny outside today. Wear shorts! Go to the beach, or the park, and get an ice cream.)

```
<label for="weather">Select the weather type today: </label>
<select id="weather">
  <option value="">--Make a choice--</option>
  <option value="sunny">Sunny</option>
  <option value="rainy">Rainy</option>
  <option value="snowing">Snowing</option>
  <option value="overcast">Overcast</option>
</select>

<p></p>
```

```
const select = document.querySelector("select");
const para = document.querySelector("p");

select.addEventListener("change", setWeather);

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

  if (choice === "sunny") {
    para.textContent =
      "It is nice and sunny outside today. Wear shorts! Go to the beach, or the
park, and get an ice cream.";
  } else if (choice === "rainy") {
    para.textContent =
      "Rain is falling outside; take a rain coat and an umbrella, and don't stay
out for too long.";
  } else if (choice === "snowing") {
    para.textContent =
      "The snow is coming down — it is freezing! Best to stay in with a cup of hot
chocolate, or go build a snowman.";
  } else if (choice === "overcast") {
    para.textContent =
      "It isn't raining, but the sky is grey and gloomy; it could turn any minute,
so take a rain coat just in case.";
  } else {
    para.textContent = "";
  }
}
```

- In the JavaScript, we are storing a reference to both the `<select>` and `<p>` elements, and adding an event listener to the `<select>` element so that when its value is changed, the setWeather() function is run.
- When this function is run, we first set a variable called choice to the current value selected in the `<select>` element. We then use a conditional statement to show different text inside the paragraph depending on what the value of choice is. Notice how all the conditions are tested in else if () { } blocks, except for the first one, which is tested in an if () { } block.
- The very last choice, inside the else { } block, is basically a "last resort" option — the code inside it will be run if none of the conditions are true. In this case, it serves to empty the text out of the paragraph if nothing is selected, for example, if a user decides to re-select the "--Make a choice--" placeholder option shown at the beginning.

Nesting if...else: we could update our weather forecast application to show a further set of choices depending on what the temperature is:

```
if (choice === "sunny") {
  if (temperature < 86) {
    para.textContent = `It is ${temperature} degrees outside — nice and sunny.
Let's go out to the beach, or the park, and get an ice cream.`;
  } else if (temperature >= 86) {
```

```
    para.textContent = `It is ${temperature} degrees outside — REALLY HOT! If you
want to go outside, make sure to put some sunscreen on.`;
  }
}
```

Logical operators: AND, OR, NOT

- && — AND; allows you to chain together two or more expressions so that all of them have to individually evaluate to true for the whole expression to return true.
- || — OR; allows you to chain together two or more expressions so that one or more of them have to individually evaluate to true for the whole expression to return true.
- NOT, expressed by the ! operator, can be used to negate an expression.

```
if (choice === "sunny" && temperature < 86) {
  para.textContent = `It is ${temperature} degrees outside — nice and sunny. Let's
go out to the beach, or the park, and get an ice cream.`;
} else if (choice === "sunny" && temperature >= 86) {
  para.textContent = `It is ${temperature} degrees outside — REALLY HOT! If you
want to go outside, make sure to put some sunscreen on.`;
}
```

```
if (iceCreamVanOutside || houseStatus === "on fire") {
  console.log("You should leave the house quickly.");
} else {
  console.log("Probably should just stay in then.");
}
```

```
if (!(iceCreamVanOutside || houseStatus === "on fire")) {
  console.log("Probably should just stay in then.");
} else {
  console.log("You should leave the house quickly.");
}
```

Switch statements:

Switch statements take a single expression/value as an input, and then look through several choices until they find one that matches that value, executing the corresponding code that goes along with it.

```
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
    break;
}
```

The keyword switch, followed by a set of parentheses. An expression or value inside the parentheses. The keyword case, followed by a choice that the expression/value could be, followed by a colon. Some code to run if the choice matches the expression. A break statement, followed by a semicolon. If the previous choice matches the expression/value, the browser stops executing the code block here, and moves on to any code that appears below the switch statement. As many other cases (bullets 3–5) as you like. The keyword default, followed by exactly the same code pattern as one of the cases (bullets 3–5), except that default does not have a choice after it, and you don't need the break statement as there is nothing to run after this in the block anyway. This is the default option that runs if none of the choices match.

```
<label for="weather">Select the weather type today: </label>
<select id="weather">
  <option value="">--Make a choice--</option>
  <option value="sunny">Sunny</option>
  <option value="rainy">Rainy</option>
  <option value="snowing">Snowing</option>
  <option value="overcast">Overcast</option>
</select>

<p></p>
```

```
const select = document.querySelector("select");
const para = document.querySelector("p");

select.addEventListener("change", setWeather);

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

  switch (choice) {
    case "sunny":
      para.textContent =
        "It is nice and sunny outside today. Wear shorts! Go to the beach, or the
park, and get an ice cream.";
      break;
    case "rainy":
```

```
      para.textContent =
        "Rain is falling outside; take a rain coat and an umbrella, and don't stay
out for too long.";
      break;
    case "snowing":
      para.textContent =
        "The snow is coming down — it is freezing! Best to stay in with a cup of
hot chocolate, or go build a snowman.";
      break;
    case "overcast":
      para.textContent =
        "It isn't raining, but the sky is grey and gloomy; it could turn any
minute, so take a rain coat just in case.";
      break;
    default:
      para.textContent = "";
  }
}
```

Ternary operator

There is one final bit of syntax we want to introduce you to before we get you to play with some examples. The ternary or conditional operator is a small bit of syntax that tests a condition and returns one value/expression if it is true, and another if it is false — this can be useful in some situations, and can take up a lot less code than an if...else block if you have two choices that are chosen between via a true/false condition. The pseudocode looks like this:

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

Here we have a variable called isBirthday — if this is true, we give our guest a happy birthday message; if not, we give her the standard daily greeting.

```
const greeting = isBirthday
  ? "Happy birthday Mrs. Smith — we hope you have a great day!"
  : "Good morning Mrs. Smith.";
```

The ternary operator is not just for setting variable values; you can also run functions, or lines of code — anything you like. The following live example shows a simple theme chooser where the styling for the site is applied using a ternary operator.

```
<label for="theme">Select theme: </label>
<select id="theme">
  <option value="white">White</option>
  <option value="black">Black</option>
</select>
```

```
<h1>This is my website</h1>
```

```
const select = document.querySelector("select");
const html = document.querySelector("html");
document.body.style.padding = "10px";

function update(bgColor, textColor) {
  html.style.backgroundColor = bgColor;
  html.style.color = textColor;
}

select.addEventListener("change", () =>
  select.value === "black"
    ? update("black", "white")
    : update("white", "black"),
);
```

Here we've got a `<select>` element to choose a theme (black or white), plus a simple h1 to display a website title. We also have a function called update(), which takes two colors as parameters (inputs). The website's background color is set to the first provided color, and its text color is set to the second provided color.

Looping code

```
100 cicles being drawn:
const btn = document.querySelector("button");
const canvas = document.querySelector("canvas");
const ctx = canvas.getContext("2d");

document.addEventListener("DOMContentLoaded", () => {
  canvas.width = document.documentElement.clientWidth;
  canvas.height = document.documentElement.clientHeight;
});

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

function draw() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  for (let i = 0; i < 100; i++) {
    ctx.beginPath();
    ctx.fillStyle = "rgb(255 0 0 / 50%)";
    ctx.arc(
      random(canvas.width),
      random(canvas.height),
      random(50),
```

```
      0,
      2 * Math.PI,
    );
    ctx.fill();
  }
}

btn.addEventListener("click", draw);
```

random(x): returns a whole number between 0 and x-1. 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.

map() and filter():

You can use map() to do something to each item in a collection and create a new collection containing the changed items

```
function toUpper(string) {
  return string.toUpperCase();
}

const cats = ["Leopard", "Serval", "Jaguar", "Tiger", "Caracal", "Lion"];

const upperCats = cats.map(toUpper);

console.log(upperCats);
// [ "LEOPARD", "SERVAL", "JAGUAR", "TIGER", "CARACAL", "LION" ]
```

You can use filter() to test each item in a collection, and create a new collection containing only items that match:

```
function lCat(cat) {
  return cat.startsWith("L");
}

const cats = ["Leopard", "Serval", "Jaguar", "Tiger", "Caracal", "Lion"];

const filtered = cats.filter(lCat);

console.log(filtered);
// [ "Leopard", "Lion" ]
```

This looks a lot like map(), except the function we pass in returns a boolean: if it returns true, then the item is included in the new array. Our function tests that the item starts with the letter "L", so the result is an array containing only cats whose names start with "L"

for loop:

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

An initializer — this is usually a variable set to a number, which is incremented to count the number of times the loop has run. It is also sometimes referred to as a counter variable. A condition — this defines when the loop should stop looping. This is generally an expression featuring a comparison operator, a test to see if the exit condition has been met. A final-expression — this is always evaluated (or run) each time the loop has gone through a full iteration. It usually serves to increment (or in some cases decrement) the counter variable, to bring it closer to the point where the condition is no longer true.

```
const results = document.querySelector("#results");

function calculate() {
  for (let i = 1; i < 10; i++) {
    const newResult = `${i} x ${i} = ${i * i}`;
    results.textContent += `${newResult}\n`;
  }
  results.textContent += "\nFinished!\n\n";
}

const calculateBtn = document.querySelector("#calculate");
const clearBtn = document.querySelector("#clear");

calculateBtn.addEventListener("click", calculate);
clearBtn.addEventListener("click", () => (results.textContent = ""));
```

This code calculates squares for the numbers from 1 to 9, and writes out the result. The core of the code is the for loop that performs the calculation.

break

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 on to 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.

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

<p></p>
```

```
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", () => {
  const searchName = input.value.toLowerCase();
  input.value = "";
  input.focus();
  para.textContent = "";
  for (const contact of contacts) {
    const splitContact = contact.split(":");
    if (splitContact[0].toLowerCase() === searchName) {
      para.textContent = `${splitContact[0]}'s number is ${splitContact[1]}.`;
      break;
    }
  }
  if (para.textContent === "") {
    para.textContent = "Contact not found.";
  }
});
```

First of all, we have some variable definitions — we have an array of contact information, with each item being a string containing a name and phone number separated by a colon. Next, we attach an event listener to the button (btn) so that when it is pressed some code is run to perform the search and return the results. We store the value entered into the text input in a variable called searchName, before then emptying the text input and focusing it again, ready for the next search. Note that we also run the toLowerCase() method on the string, so that searches will be case-insensitive. Now on to the interesting part, the for...of loop: Inside the loop, we first split the current contact at the colon character, and store the resulting two values in an array called splitContact. We then use a conditional statement to test whether splitContact[0] (the contact's name, again lower-cased with toLowerCase()) is equal to the inputted searchName. If it is, we enter a string into the paragraph to report what the contact's number is, and use break to end the loop. After the loop, we check whether we set a contact, and if not we set the paragraph text to "Contact not found.".

continue

Skipping iterations with continue The continue statement works similarly 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).

```
<label for="number">Enter number: </label>
<input id="number" type="number" />
<button>Generate integer squares</button>

<p>Output:</p>
```

```
const para = document.querySelector("p");
const input = document.querySelector("input");
const btn = document.querySelector("button");

btn.addEventListener("click", () => {
  para.textContent = "Output: ";
  const num = input.value;
  input.value = "";
  input.focus();
  for (let i = 1; i <= num; i++) {
    let sqRoot = Math.sqrt(i);
    if (Math.floor(sqRoot) !== sqRoot) {
      continue;
    }
    para.textContent += `${i} `;
  }
});
```

In this case, the input should be a number (num). The for loop is given a counter starting at 1 (as we are not interested in 0 in this case), an exit condition that says the loop will stop when the counter becomes bigger than the input num, and an iterator that adds 1 to the counter each time. Inside the loop, we find the square root of each number using Math.sqrt(i), then check whether the square root is an integer by testing whether it is the same as itself when it has been rounded down to the nearest integer (this is what Math.floor() does to the number it is passed). If the square root and the rounded down square root do not equal one another (!==), it means that the square root is not an integer, so we are not interested in it. In such a case, we use the continue statement to skip on to the next loop iteration without recording the number anywhere. If the square root is an integer, we skip past the if block entirely, so the continue statement is not executed; instead, we concatenate the current i value plus a space at the end of the paragraph content.

While and do...while

while:

```
const cats = ["Pete", "Biggles", "Jasmine"];

let myFavoriteCats = "My cats are called ";
```

```
let i = 0;

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

  i++;
}

console.log(myFavoriteCats); // "My cats are called Pete, Biggles, and Jasmine."
```

while...do:

```
const cats = ["Pete", "Biggles", "Jasmine"];

let myFavoriteCats = "My cats are called ";

let i = 0;

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

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

console.log(myFavoriteCats); // "My cats are called Pete, Biggles, and Jasmine."
```

Functions - reusable blocks of code

Another essential concept in coding is functions, which allow you to store a piece of code that does a single task inside a defined block, and then call that code whenever you need it using a single short command — rather than having to type out the same code multiple times.

Built-in browser functions: myText.replace("string", "sausage") : takes a source string and replaces it with a target string. myArray.join(" ") : takes an array and joins all the array items into a single string Math.random() : generates a random number between 0 and up to, but not including, 1

In fact, some of the code you are calling when you invoke (a fancy word for run, or execute) a built-in browser function couldn't be written in JavaScript — many of these functions are calling parts of the background

browser code, which is written largely in low-level system languages like C++, not web languages like JavaScript.

Functions vs. Methods

Functions that are part of objects are called methods. The built-in code we've made use of so far comes in both forms: functions and methods.

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
```

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.

Anonymous functions and arrow functions

An anonymous function has no name. You'll often see anonymous functions when a function expects to receive another function as a parameter. In this case, the function parameter is often passed as an anonymous function.

```
(function () {
  alert("hello");
});
```

example: addEventListener() parameters:

- the name of the event to listen for, which in this case is keydown
- a function to run when the event happens.

```
function logKey(event) {
  console.log(`You pressed "${event.key}".`);
}

textBox.addEventListener("keydown", logKey);
```

can also be written with an anonymous function:

```
textBox.addEventListener("keydown", function (event) {
  console.log(`You pressed "${event.key}".`);
});
```

arrow function:

```
textBox.addEventListener("keydown", (event) => {
  console.log(`You pressed "${event.key}".`);
});
```

If the function only takes one parameter, you can omit the parentheses around the parameter:

```
textBox.addEventListener("keydown", event => {
  console.log(`You pressed "${event.key}".`);
});
```

Finally, if your function contains only one line that's a return statement, you can also omit the braces and the return keyword and implicitly return the expression. In the following example, we're using the map() method of Array to double every value in the original array:

```
const originals = [1, 2, 3];

const doubled = originals.map(item => item * 2);

console.log(doubled); // [2, 4, 6]
```

Arrow function live sample: gives a box that prints("you pressed "{letter}") each time you press a letter.

```
<input id="textBox" type="text" />
<div id="output"></div>
```

```
const textBox = document.querySelector("#textBox");
const output = document.querySelector("#output");
```

```
textBox.addEventListener("keydown", (event) => {
  output.textContent = `You pressed "${event.key}".`;
});
```

Function scope

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 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.

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

```
// first.js
const name = "Chris";
function greeting() {
  alert(`Hello ${name}: welcome to our company.`);
}
```

```
// second.js
const name = "Zaptec";
function greeting() {
  alert(`Our company is called ${name}.`);
}
```

Both functions you want to call are called greeting(), but you can only ever access the first.js file's greeting() function (the second one is ignored). In addition, an error results when attempting (in the second.js file) to assign a new value to the name variable — because it was already declared with const, and so can't be reassigned.

Introduction to events

Events are things that happen in the system you are programming, which the system tells you about so your code can react to them. For example, if the user clicks a button on a webpage, you might want to react to that action by displaying an information box.

- The user selects, clicks, or hovers the cursor over a certain element.
- The user chooses a key on the keyboard.
- The user resizes or closes the browser window.
- A web page finishes loading.
- A form is submitted.
- A video is played, paused, or ends.
- An error occurs.

To react to an event, you attach an event handler to it. This is a block of code (usually a JavaScript function that you as a programmer create) that runs when the event fires. When such a block of code is defined to run in response to an event, we say we are registering an event handler. Note: Event handlers are sometimes called event listeners — they are pretty much interchangeable for our purposes, although strictly speaking, they work together. The listener listens out for the event happening, and the handler is the code that is run in response to it happening.

example: handling a click event

```
<button>Change color</button>
```

```
const btn = document.querySelector("button");

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

btn.addEventListener("click", () => {
  const rndCol = `rgb(${random(255)} ${random(255)} ${random(255)})`;
  document.body.style.backgroundColor = rndCol;
});
```

The HTML `<button>` element will fire an event when the user clicks the button. So it defines an addEventListener() function, which we are calling here. We're passing in two parameters:

the string "click", to indicate that we want to listen to the click event. Buttons can fire lots of other events, such as "mouseover" when the user moves their mouse over the button, or "keydown" when the user presses a key and the button is focused. a function to call when the event happens. In our case, the function generates a random RGB color and sets the background-color of the page `<body>` to that color.

click can be changed by:

- focus and blur — The color changes when the button is focused and unfocused; try pressing the tab to focus on the button and press the tab again to focus away from the button. These are often used to display information about filling in form fields when they are focused, or to display an error message if a form field is filled with an incorrect value.
- dblclick — The color changes only when the button is double-clicked.

- mouseover and mouseout — The color changes when the mouse pointer hovers over the button, or when the pointer moves off the button, respectively.

remove event listeners:

```
btn.removeEventListener("click", changeBackground);
```

Event handlers can also be removed by passing an AbortSignal to addEventListener() and then later calling abort() on the controller owning the AbortSignal. For example, to add an event handler that we can remove with an AbortSignal:

```
const controller = new AbortController();

btn.addEventListener("click",
  () => {
    const rndCol = `rgb(${random(255)} ${random(255)} ${random(255)})`;
    document.body.style.backgroundColor = rndCol;
  },
  { signal: controller.signal } // pass an AbortSignal to this handler
);
```

Then the event handler created by the code above can be removed like this:

```
controller.abort(); // removes any/all event handlers associated with this
controller
```

Event handler properties: Objects (such as buttons) that can fire events also usually have properties whose name is on followed by the name of the event. For example, elements have a property onclick. This is called an event handler property. To listen for the event, you can assign the handler function to the property.

For example, we could rewrite the random-color example like this:

```
const btn = document.querySelector("button");

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

btn.onclick = () => {
  const rndCol = `rgb(${random(255)} ${random(255)} ${random(255)})`;
  document.body.style.backgroundColor = rndCol;
};
```

You can also set the handler property to a named function:

```
const btn = document.querySelector("button");

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

function bgChange() {
  const rndCol = `rgb(${random(255)} ${random(255)} ${random(255)})`;
  document.body.style.backgroundColor = rndCol;
}

btn.onclick = bgChange;
```

Event objects

Sometimes, inside an event handler function, you'll see a parameter specified with a name such as event, evt, or 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:

```
const btn = document.querySelector("button");

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

function bgChange(e) {
  const rndCol = `rgb(${random(255)} ${random(255)} ${random(255)})`;
  e.target.style.backgroundColor = rndCol;
  console.log(e);
}

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

Here you can see we are including an event object, e, in the function, and in the function setting a background color style on e.target — which is the button itself. The target property of the event object is always a reference to the element the event occurred upon. So, in this example, we are setting a random background color on the button, not the page.

keydown event: The keydown event fires when the user presses a key. Its event object is a KeyboardEvent, which is a specialized Event object with a key property that tells you which key was pressed:

```
<input id="textBox" type="text" />
<div id="output"></div>
```

```
const textBox = document.querySelector("#textBox");
const output = document.querySelector("#output");
textBox.addEventListener("keydown", (event) => {
  output.textContent = `You pressed "${event.key}".`;
});
```

a simple HTML form that requires you to enter your first and last name and complains when it is empty:

```
<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>
```

```
const form = document.querySelector("form");
const fname = document.getElementById("fname");
const lname = document.getElementById("lname");
const para = document.querySelector("p");

form.addEventListener("submit", (e) => {
  if (fname.value === "" || lname.value === "") {
    e.preventDefault();
    para.textContent = "You need to fill in both names!";
  }
});
```

Event bubbling

Event bubbling describes how the browser handles events targeted at nested elements.

Setting a listener on a parent element Consider a web page like this:

```
<div id="container">
  <button>Click me!</button>
</div>
<pre id="output"></pre>
```

Here the button is inside another element, a `<div>` element. We say that the

element here is the parent of the element it contains. What happens if we add a click event handler to the parent, then click the button?

```
const output = document.querySelector("#output");
function handleClick(e) {
  output.textContent += `You clicked on a ${e.currentTarget.tagName} element\n`;
}

const container = document.querySelector("#container");
container.addEventListener("click", handleClick);
```

You'll see that the parent fires a click event when the user clicks the button. This makes sense: the button is inside the `<div>`, so when you click the button you're also implicitly clicking the element it is inside.

```
<body>
  <div id="container">
    <button>Click me!</button>
  </div>
  <pre id="output"></pre>
</body>
```

```
const output = document.querySelector("#output");
function handleClick(e) {
  output.textContent += `You clicked on a ${e.currentTarget.tagName} element\n`;
}

const container = document.querySelector("#container");
const button = document.querySelector("button");

document.body.addEventListener("click", handleClick);
container.addEventListener("click", handleClick);
button.addEventListener("click", handleClick);
```

In this case:

the click on the button fires first followed by the click on its parent (the `<div>` element) followed by the `<div>` element's parent (the `<body>` element). We describe this by saying that the event bubbles up from the innermost element that was clicked.

Video player example In this example our page contains a video, which is hidden initially, and a button labeled "Display video". We want the following interaction:

When the user clicks the "Display video" button, show the box containing the video, but don't start playing the video yet. When the user clicks on the video, start playing the video. When the user clicks anywhere in the box outside the video, hide the box. The HTML looks like this:

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

<div class="hidden">
  <video>
    <source
      src="https://interactive-examples.mdn.mozilla.net/media/cc0-
videos/flower.webm"
      type="video/webm" />
    <p>
      Your browser doesn't support HTML video. Here is a
      <a href="rabbit320.mp4">link to the video</a> instead.
    </p>
  </video>
</div>
```

It includes:

a `<button>` element a `<div>` element which initially has a class="hidden" attribute a `<video>` element nested inside the `<div>` element. We're using CSS to hide elements with the "hidden" class set.

```
const btn = document.querySelector("button");
const box = document.querySelector("div");
const video = document.querySelector("video");

btn.addEventListener("click", () => box.classList.remove("hidden"));

video.addEventListener("click", (event) => {
  event.stopPropagation();
  video.play();
});

box.addEventListener("click", () => box.classList.add("hidden"));
```

This adds three 'click' event listeners:

one on the `<button>`, which shows the `<div>` that contains the `<video>` one on the `<video>`, which starts playing the video one on the `<div>`, which hides the video

You should see that when you click the button, the box and the video it contains are shown. But then when you click the video, the video starts to play, but the box is hidden again! (we don't want this behavior and avoid it using the stopPropagation())

The video is inside the `<div>` — it is part of it — so clicking the video runs both the event handlers, causing this behavior.

The Event object has a function available on it called stopPropagation() which, when called inside an event handler, prevents the event from bubbling up to any other elements.

Event capture

An alternative form of event propagation is event capture. This is like event bubbling but the order is reversed: so instead of the event firing first on the innermost element targeted, and then on successively less nested elements, the event fires first on the least nested element, and then on successively more nested elements, until the target is reached.

Event capture is disabled by default. To enable it you have to pass the capture option in addEventListener().

```
<body>
  <div id="container">
    <button>Click me!</button>
  </div>
  <pre id="output"></pre>
</body>
```

```
const output = document.querySelector("#output");
function handleClick(e) {
  output.textContent += `You clicked on a ${e.currentTarget.tagName} element\n`;
}

const container = document.querySelector("#container");
const button = document.querySelector("button");

document.body.addEventListener("click", handleClick, { capture: true });
container.addEventListener("click", handleClick, { capture: true });
button.addEventListener("click", handleClick);
```

In this case, the order of messages is reversed: the `<body>` event handler fires first, followed by the `<div>` event handler, followed by the `<button>` event handler:

You clicked on a BODY element You clicked on a DIV element You clicked on a BUTTON element

Event delegation

In the last section, we looked at a problem caused by event bubbling and how to fix it. Event bubbling isn't just annoying, though: it can be very useful. In particular, it enables event delegation. In this practice, when we want some code to run when the user interacts with any one of a large number of child elements, we set the event listener on their parent and have events that happen on them bubble up to their parent rather than having to set the event listener on every child individually.

Let's go back to our first example, where we set the background color of the whole page when the user clicked a button. Suppose that instead, the page is divided into 16 tiles, and we want to set each tile to a random color when the user clicks that tile.

Here's the HTML:

```
<div id="container">
  <div class="tile"></div>
  <div class="tile"></div>
  <div class="tile"></div>
  <div class="tile"></div>
  <div class="tile"></div>
  <div class="tile"></div>
  <div class="tile"></div>
  <div class="tile"></div>
  <div class="tile"></div>
  <div class="tile"></div>
  <div class="tile"></div>
  <div class="tile"></div>
  <div class="tile"></div>
  <div class="tile"></div>
  <div class="tile"></div>
  <div class="tile"></div>
</div>
```

css:

```
.tile {
  height: 100px;
  width: 25%;
  float: left;
}
```

JavaScript:

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

function bgChange() {
  const rndCol = `rgb(${random(255)} ${random(255)} ${random(255)})`;
```

```
  return rndCol;
}

const container = document.querySelector("#container");

container.addEventListener("click", (event) => {
  event.target.style.backgroundColor = bgChange();
});
```

It's not just web pages Events are not unique to JavaScript — most programming languages have some kind of event model, and the way the model works often differs 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. The HTTP connect event docs provide a good example.

You can also use JavaScript to build cross-browser add-ons — browser functionality enhancements — using a technology called WebExtensions. The event model is similar to the web events model, but a bit different — event listeners' properties are written in camel case (such as onMessage rather than onmessage), and need to be combined with the addListener function. See the runtime.onMessage page for an example.