

SCRIPT
JAVAS

JAVASCRIPT

“...myself and a few others saw that there was a need for a language that was approachable, that you could put directly in the web page”

BRENDAN EICH, *original creator of JavaScript*

THE NEWSTACK PODCAST, 2018

JAVAS
SCRIPT

DAILY BREAD PAYMENTS CALCULATOR

This app creates a list of all recurring payments out of your bank account per month. It stores the date, amount, and payment recipient name. Like Excel, but funner, and easily customisable.

The user inputs day of the month, and the total required from that day on till the end of the month is calculated and displayed.



GLOSSARY
OF TERMS

REPOSITORY - FULL CODE - <https://github/lavenderlens/> <PUBLIC WHEN PUBLISHED AT> [frontend-cookbook/samples/payments-calculator/](https://frontend-cookbook.com/samples/payments-calculator/)

1. Declare an empty array to store payments.

```
const payments = [];
```

We have used the keyword **const** to declare it, but that doesn't mean it will always be empty! It just means its overall container type cannot change into, say, a string, and we can't re-use the same variable name again further on down the code.

2. Define a function to create payment objects and add them to the payments array. Note the use of default parameters, so we don't end up with undefineds in the case of an invalid entry. Note it is almost a **pure function**, save for the reference to the payments array. We could pass in a payments array and it would be fully pure.



```
const createPayment = (date=1, amount=0.0, recipient="bill") => {  
  let newPayment = {  
    date,  
    amount,  
    recipient  
  }  
  payments.push(newPayment);  
}
```

3. Define a function to calculate payments remaining. It accepts a day of the month, and a payments array. Why accept a payments array when we only have one? This future-proofs the code if we later decide to transform the original array.

```
const calculatePaymentsRemaining = (todaysDate, paymentsList) => {
  let total = 0.0;
  let paymentsRemaining = paymentsList.filter(
    payment => payment.date > (todaysDate - 1)
  );
  for(let payment of paymentsRemaining){
    total += payment.amount;
  }//could write a reducer here - arguably less readable
  return total.toFixed(2);
}
```

4. Now that we have the functions to create payment objects, and calculate remaining payments, we can tie this up to DOM input from the user. This code identifies three areas in our HTML -

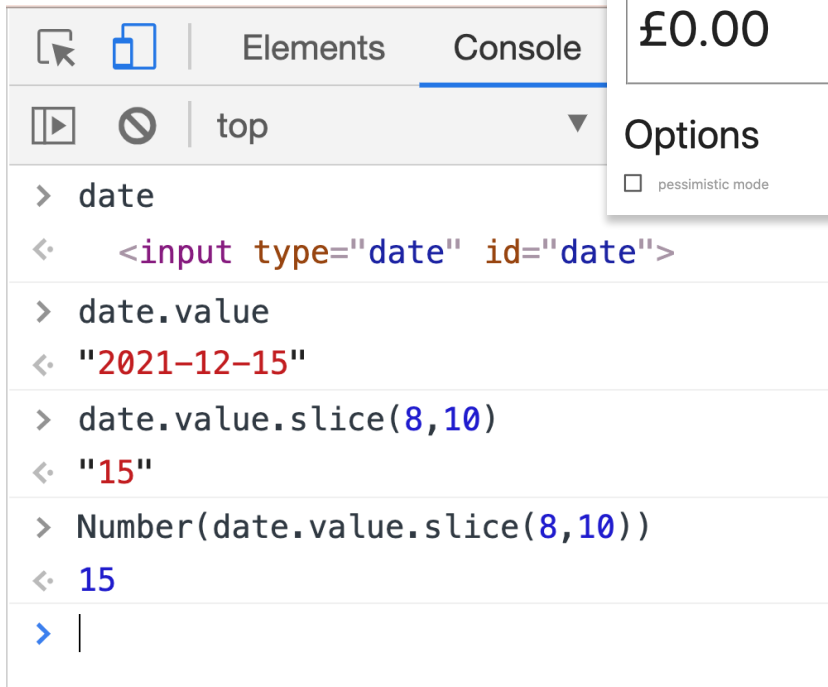
- `#date` (the date selector)
- `#filter` (the toggle pessimistic mode checkbox), and
- `#result` (the displayed result).

```
<label for="date">day of the month</label>
<input type="date" id="date"/>
<fieldset>
  <legend>AVAILABLE BALANCE REQUIRED</legend>
  <h2 id="result">£0.00</h2>
</fieldset>
<h3>Options</h3>
<p class="spaced">
  <label for="filter">
    <input type="checkbox" name="filter" id="filter"/>
    <span>pessimistic mode</span>
  </label>
</p>
```

5. The resultant DOM looks like this, with a little help from either the Materialize CDN, linked to either in the head of our page, or the Materialize distribution, linked to locally:

6. Testing the two functions in developer tools looks like this, drilling down into the element to get the value, and applying a little **type coercion** to get a number that we can do Maths to if we choose. In the console,

- **Blue is type number**
- **Orange is type string**
- **Black is output only.**



The screenshot shows the browser's developer tools console. The 'Elements' tab is active, showing a selected `<input type="date" id="date">` element. The console log shows the following steps:

- `> date` (black text)
- `< <input type="date" id="date">` (black text)
- `> date.value` (black text)
- `< "2021-12-15"` (orange text)
- `> date.value.slice(8,10)` (black text)
- `< "15"` (orange text)
- `> Number(date.value.slice(8,10))` (black text)
- `< 15` (blue text)
- `> |` (black text)

Recurring Payments

enter current day of the month - find out available funds needed in account - till payday 🍷 - toggle pessimistic mode to be really sure 🍷 - it allows for payments taken up to 3 days later than usual

day of the month

dd/mm/yyyy



AVAILABLE BALANCE REQUIRED

£0.00

Options

pessimistic mode

7. This code identifies our 3 areas in the DOM, and either gets data from them or returns it to display. We could add a button at this stage, and listen for click events, but I have added change listeners to both the date selector and an optional

checkbox toggle which takes 3 days off the date entered, so they BOTH trigger the callback `getAndDisplayTotal`. It's much more lively and saves clicks. This is all the DOM-related code, together. It's worth noting that:

- You need to either place this at the end of your HTML elements that it references, basically, a last child of your `<body>`, just before the closing body tag `</body>`, OR
- Do something more clever with it to delay it running before the DOM is built.

```
const date = document.querySelector('#date');
const filter = document.querySelector('#filter');
const getDateAndDisplayTotal = () => {
  let day = Number(document.querySelector('#date').
value.slice(8,10));
  if(filter.checked){
    document.querySelector('#result').innerHTML =
`£${calculatePaymentsRemaining(day - 3, payments)}`
  } else {
    document.querySelector('#result').innerHTML =
`£${calculatePaymentsRemaining(day, payments)}`
  }
}
date.addEventListener("change", function() {
  getDateAndDisplayTotal()
});
filter.addEventListener("change", function() {
  getDateAndDisplayTotal()
});
```

8. Populate payments by calling `createPayment` repeatedly with different data. Start off by hardcoding your own expenses. Refactor later to accept ALL payments from ANY user.

```
createPayment(9, 22.00, "Sky");
createPayment(10, 90.75, "HME/Argos card");//
createPayment(10, 18.49, "Apple");
createPayment(10, 34, "GoDaddy DNH");
createPayment(10, 11, "Netflix");
```

9. OK, so now we have the basic app working and looking reasonably neat, we can add extra functionality. Suppose we decided it would be nice to show the expenses while taking into account payment holidays on one or more credit or store cards.



10. Functional Programming, or FP, design patterns encourage leaving the original data, in this case our array of payment objects, unchanged, or “immutable”.

11. FP transformations of the data return a COPY of the original which may be assigned its own unique pointer, and so leaving the original unchanged.

12. We can use the “recipient” field of each payment object to search for its corresponding entry in the payments array.

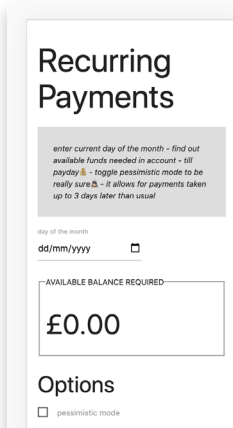
13. We can then use `Array.prototype.filter()`, repeatedly, to filter out payment object from the collection, one-by-one. In code, it would look something like this:

```
const paymentsWithHolidays = payments
  .filter(payment =>
    !(payment.recipient.includes("Virgin")))
    // returns new array without Virgin card
  .filter(payment =>
    !(payment.recipient.includes("Marbles")))
    // returns new array without Marbles card
  .filter(payment =>
    !(payment.recipient.includes("Argos")))
    // returns new array without Argos card
    // expression finishes evaluating
    // a reference to this expression is stored
    // now variable name points to third and final array
```



What’s fully going on here? The filter function TAKES A FUNCTION as an argument. It’s a tiny algorithm usually written inline as an arrow function but doesn’t have to be, which tells filter the criterion to filter on. In this case it’s allow any payment that doesn’t (!) contain the name passed in as part of its recipients property.

14. Let’s test the logic of our filter algorithm in the console: [over]



```

> payments
< (33) [{"recipient": "Virgin"}, {"recipient": "Marbles"}, {"recipient": "Argos"}]
> const paymentsWithHolidays = payments
  .filter(payment =>
    !(payment.recipient.includes("Virgin")))
  .filter(payment =>
    !(payment.recipient.includes("Marbles")))
  .filter(payment =>
    !(payment.recipient.includes("Argos")))
< undefined
> paymentsWithHolidays
(30) [{"recipient": "Virgin"}, {"recipient": "Marbles"}, {"recipient": "Argos"}]
    
```

CHALLENGES

15. **Modify the UI** to take payment holiday data from the user and **modify the function** to accept a search term.
16. If you want to keep the loading view of the app nice and clean, you can **make the filter function optional**, say, by using another checkbox toggle.
17. This checkbox, when checked, would **render a new text input and search button** which would **call the filter function**. Output the result in the same place - don't confuse the user by changing the UI too much! **HINT:** use `document.createElement()` to create the new elements, `document.appendChild()` to place them in the DOM.
18. At the moment, the app will only remember transformations on the payments array as long as the browser tab is kept open. If we want to persist state between uses of the app we can leverage the HTML5 **localStorage API**. This is like a halfway house to connecting with a database on the server.

NOTES

Can you think of other questions you would like to be able to ask your data?