ADVANCED JAVASCRIPT FOR WEB SITES AND WEB APPLICATIONS

SESSION 5 - PATTERN EXAMPLE - BUBBLING AND DELEGATION

Pattern example: shopping basket

Event handlers
Bubbling
Delegation

PATTERN EXAMPLE: SHOPPING BASKET

Build a simple application to manage a shopping basket.

Yon can add items to the basket, remove an item from the basket and update the quantity of an item.

You can also do a quick check to return the number of items in the basket and add the total amount for all items.

Each item needs the following information:

- unique id (can be a string)
- price
- description
- name
- quantity
- url

How can we do this?

How do we represent the:

- basket?
- the items?

What functions do we need?

- removeFromBasket
- addToBasket
- updateBasket
- countItems
- basketTotal

`removeFromBasket`, `addToBasket` and `updateBasket` will all need to loop through the basket and find an item.

We can create another function called `findInBasket` to do this.

findinbasket

The functions needs to loop through the basket find an item, passed as an argument.

It needs to return the position of the item in the array, if it finds one, or `null`.

It's a good idea to add comments to function like this, that indicate what the function takes and what it outputs.

There's different methods and conventions of doing this, we'll just use a simple commet for now:

```
// input: the type of input for the function
// output: what the function ouputs
```

```
findInBasket = function (item) {

  // input: basket array, an item object
  // output: the index of item in array, or null

  for (var i = 0; i < this.basket.length; i++) {
    if (this.basket[i].id === item.id) {
      return i
    }
  }

  return null

},
```

`countitems`

This function needs to loop through the array and add all quantities together.

It returns a number.

```
countItems = function () {

  // input: basket array
  // output: the number of item

  var total = 0;

  for (var i = 0; i < this.basket.length; i ++) {
    total = total + this.basket[i].quantity;
  }

  return total

},
```

addtobasket

This functions takes the item to add as an argument.

It needs to loop through the basket and search for the item (using `findInBasket`).

If it finds an item, it needs to increment its quantity by one, the item doesn't exist in the basket, it needs to push to the array.

Remember that `findInBasket` returns `null` or a position.

```
addToBasket = function (item) {

  // input: basket array, an item object

  // item object looks like:
  // {id: "item id", price: 100, description: "string", url: "url",quanti

  var itemFound = findInBasket(item);

  if (itemFound === null) {
    this.basket.push(item);
  }
  else {
    this.basket[itemFound].quantity ++;
  }

},
```

removefrombasket

This function takes an item (as object) as argument.

It needs to loop through the basket, look for the item (using findInBasket) and remove the item if it exists or do nothing if it doesn't.

```
removeFromBasket = function (item) {

  // input: basket array, an item object

  var itemFound = findInBasket(item);

  if (itemFound !== null) {
    //removes element from basket
    this.basket.splice(itemFound, 1);
  }

},
```

updatebasket

This function needs to update the quantity of an item.

It is used when the quantity of an item is changed.

Again, it takes an item and a quantity as argument.

It needs to loop through the basket, and if it finds the item, update its quantity.

```
uupdateBasket = function (item, newQuantity) {

  // input: basket array, an item object, a quantity as a number

  var itemFound = findInBasket(item);

  if (itemFound !== null) {
    this.basket[itemFound].quantity = newQuantity;
  }

},
```

baskettotal

This function needs to loop through the basket and add the total price of all items.

```
basketTotal = function (basket) {

  // input: basket array
  // output: the total amount, as a number

  var total = 0;

  for (var i = 0; i < this.basket.length; i++) {
    total = total + this.basket[i].price * this.basket[i].quantity;
  }

  return total
}
```

EVENT HANDLERS

You can register an event handler the following way:

```
node.addEventListener(
    event_type,
    function_to_run
);
```

```
<a id="link" href="">Link to click</a>
```

```
var link = document.getElementById("link");

link.addEventListener(
    "click",
    function (event) {
      console.log(event)
    }
);
```

Every event handler declared gets an event object assigned to its argument handler function. In the previous example, I called it `event`.

BUBBLING

Remember that in the DOM, when an event is triggered, it fires first on the most downwards element that triggered it (the last element that doesn't have any children) and then goes up the tree, triggering the same event for the parents and grand-parents and etc. of that element.

This means that all event handlers that have been declared on elements up the tree will fire if any children or grand-children are firing the event.

Bubbling Example

See it live

You can prevent bubbling if needed by calling the `stopPropagation` method on the event object:

```
event.stopPropagation();
```

In this case, an event will not bubble up to its parents or grand-parents.

EXERCISE 1

Can you create a small test case for an event that does not bubble up the tree? For example, mouse leave does not bubble up.

Use the html in exercise1.html and create a simple test case for this.

```
var parent = document.getElementById("parent");
var child = document.getElementById("child");


parent.addEventListener(
  "mouseleave",
  function () {
    alert("parent mouse leave event!");
  }
);

child.addEventListener(
  "mouseleave",
  function () {
    alert("child mouse leave event!");
  }
);
```

DELEGATION

The event object is interesting because it has a `target` property, that returns the first most nested element that triggered the event.

It also has a `currentTarget` property, that returns the item the current event handler was attached to.

Remember that the event object is automatically assigned as the argument of the function used when declaring the event handler.

We can use this to our advantage! With event delegation.

Using this pattern, you can declare only one event handler on a parent that has multiple children and using the `event` object find out which child triggered the event.

In exercise2.html, declare a new event handler for a click on the `div` that has an id of "button-wrapper".

Console log `currentTarget` of the event object.

```
var wrapper = document.querySelectorAll(".button-wrapper")[0];

wrapper.addEventListener(
  "click",
  function (event) {
    console.log(event.currentTarget);
  }
);
```

Using the exact same code as previously, now console log the `target` of the event object.

```
var wrapper = document.querySelectorAll(".button-wrapper")[0];

wrapper.addEventListener(
  "click",
  function (event) {
    console.log(event.target);
  }
);
```

The second example tells us which button was clicked, without having to declare one event handler per button.

The other way to achieve this, without event delegation, is to declare one event handler per button:

```
function handlerFunction (event) {
  console.log(event.target);
}

document.querySelectorAll(".button")[0].addEventListener(
  "click",
  handlerFunction
);
document.querySelectorAll(".button")[1].addEventListener(
  "click",
  handlerFunction
);
document.querySelectorAll(".button")[2].addEventListener(
  "click",
  handlerFunction
);
document.querySelectorAll(".button")[3].addEventListener(
  "click",
  handlerFunction
);
document.querySelectorAll(".button")[4].addEventListener(
  "click",
  handlerFunction
);
```

There is one more thing we need to improve, if you go back to the following code and test it:

```
var wrapper = document.querySelectorAll(".button-wrapper")[0];

wrapper.addEventListener(
  "click",
  function (event) {
    console.log(event.target);
  }
);
```

You will see that it fires when we click on the buttons, but also when we click on any area of the parent, but outside a button.

We probably want to change our event handler slightly so that the handler function checks which element is actually clicked before doing anything.

Amend the previous code to check if the `target` element is a button or not.

There are mutliple ways of doing this, you can check the type of html element of the button, or check the class on the element.

```
var wrapper = document.querySelectorAll(".button-wrapper")[0];

wrapper.addEventListener(
  "click",
  function (event) {
    if (event.target.classList.contains("button")) {
      console.log(event.target);
    }
  }
);
```

```
var wrapper = document.querySelectorAll(".button-wrapper")[0];

wrapper.addEventListener(
  "click",
  function (event) {
    if (event.target.nodeName === "P") {
      console.log(event.target);
    }
  }
);
```

why is event delegation usefull?

- lets you write less code and declare less event handlers
- if you need to remove or add elements dynamically in the DOM, attaching event handlers can quickly become a problem.

Imagine our button situation, where you need to declare an event handler for each button in the list. If you need to add or remove buttons dynamically, you will also need to update your event handlers as you create the buttons.

Using event delegation, you can add or remove buttons safely, declare one event handler on the wrapper and check which button is clicked.

EXERCISE 3

In exercise3.html, there is a list of elements with a description.

Write a small app that adds an element when clicking the add button. The description should be the text entered in the input field.

When cliking anywhere on an existing element, it removes it from the DOM.

Extra: use the revealing module pattern!

```
var
  newEl,
  newElementDescription = document.querySelectorAll(".element-description
  addButton = document.querySelectorAll(".add-button")[0],
  elWrapper = document.querySelectorAll(".element-wrapper")[0],
  inputEl = document.querySelectorAll(".element-description")[0];


elWrapper.addEventListener(
  "click",
  function (event) {
    var el = event.target;
    if (el.classList.contains("element")) {
      el.parentNode.removeChild(el);
    }
  }
);

addButton.addEventListener(
  "click",
  function(event){
    newEl = "<li class=\"element\">" + inputEl.value + "<\/li>";
    elWrapper.insertAdjacentHTML("beforeend", newEl);
  }
);
```