

JavaScript

Lab 2: Arrays, Functions

J. Pieck, L. Van Houdt, S. Van Peborgh, T. Van Nuland

1 ARRAYS 3

1.1 INTRODUCING ARRAYS 3

1.2 IMPROVING GETAVERAGERATING 4

1.3 REFACTORING GETHIGHESTRATING 5

1.4 REFACTORING PRINTFAVORITEGAMES 5

1.5 ARRAY DESTRUCTURING 6

1.6 ARRAY SPREADING 6

2 HIGHER-ORDER FUNCTIONS 9

2.1 ITERATING OVER ARRAYS: THE *FOREACH* METHOD 9

2.2 FILTERING ARRAYS: THE *FILTER* METHOD 10

2.3 TRANSFORMING ARRAYS: THE *MAP* METHOD 12

3 SUBMITTING YOUR RESULT 13

1 Arrays

In the previous exercises, you likely noticed that we had to repeat a lot of code to handle each game individually. This violates a fundamental programming principle known as DRY (Don't Repeat Yourself). To fix this, we will now refactor our code using arrays. An array is a data structure that allows us to store a collection of items—in our case, all our game objects—in a single variable.

By grouping our games into one array, we can update our functions to accept the entire collection as a single parameter. This makes our code much cleaner, more efficient, and reusable for different sets of data. Finally, we will also learn how to merge multiple arrays, further reducing repetition.

1.1 Introducing arrays

So far, we've used a separate variable for each game. This approach is inefficient because it forces us to manually call `addStatus` for every single game inside the `printAllGames` function. We can automate this process by storing all our game objects in a single collection called an array. This allows us to loop through the entire list and process each game programmatically, eliminating the need for repetitive code.

```
const animals = [
  animal1,
  animal2
];

for (const animal of animals) {
  console.log(animal.name);
}
```



Refactor your code to use an array. Start by creating a single array named `games` that contains all of your existing game objects.

Next, update the `printAllGames` function. Remove the repetitive, manual calls to `addStatus`. Replace them with a `for...of` loop that iterates through every element in your new games array, calling `addStatus` for each game.

1.2 Improving getAverageRating

With our games now stored in an array, we can improve the `getAverageRating` function to be more dynamic. By using a `for-loop`, the function can automatically process any number of games, meaning you no longer have to update it manually whenever the collection changes. This is the same technique used in the farm animals demo, where a simple `for-loop` calculated the average age by iterating through the entire list of animals.

```
let sumOfAges = 0;
for (const animal of animals) {
  sumOfAges += animal.age;
}
return sumOfAges / animals.length;
```



Refactor the `getAverageRating` function.
Modify it to dynamically calculate the average by iterating through the `games` array with a loop. The result displayed on the page should look the same as before.

Some statistics ...
Average rating: 4.4
Fifa23 is game with the highest rating: 7



If your calculated average is a long number with many decimals, you can format it using JavaScript's built-in `toFixed()` method. For example, `result.toFixed(1)` will round the number to one decimal place.

1.3 Refactoring getHighestRating

Next, you'll need to refactor the `getHighestRating` function to work with the `games` array. Instead of comparing individual variables, you will now iterate through the array using a loop to find the game with the highest rating.

An important part of this update is handling the edge case where the array might be empty. Add an `if-statement` at the beginning to check if the `games` array is empty. If it is, display an appropriate message (e.g. "No games in the library.") instead of attempting to find a highest rating.



Update the `getHighestRating` function. Modify it to find the highest-rated game by iterating through the games array with a loop. Be sure to include a check for an empty array and display an appropriate message.

1.4 Refactoring printFavoriteGames

The final function that needs to be refactored is printFavoriteGames. It's the last piece of code that still relies on individual, hard-coded variables instead of the games array.



Update the `printFavoriteGames` function.
Modify the function to use a `for-loop` that iterates through the entire `games` array. Inside the loop, use an `if-statement` to check if a game's `isFavorite` property is set to `true`, and only print the games that meet this condition.

1.5 Array destructuring

Now that all our functions are updated to work with the games array, our next goal is to display only the first two games from the collection. To extract specific items from an array, we can use a convenient syntax called destructuring. This technique provides a shortcut to pull elements from an array and assign them to their own variables. For example, this is how you would extract the first two items from the farm animals array:

```
const animals = [ duck, dog, cat];
const [first, second] = animals;
```



Display the names of the first two games from your `games` array.

First, add a meaningful title to the page using `addStatus`. Then, use array destructuring to extract the first two `game` objects from the games array into new constants.

Finally, call `addStatus` for each of these two games to display their names on the page.



1.6 Array spreading

Your best friend sees your game-tracking system and wants to use it for their own collection. Instead of combining everything into one big list, you decide to keep your games and your friend's games in two separate arrays. This is great for organization, but sometimes you'll want to combine them—for example, to calculate total statistics or display a complete master list. To temporarily merge these two arrays into a single new one without changing the originals, you can use a powerful feature called the `spread syntax (...)`.

The spread operator (`...`) unpacks the elements from an array. When you use it inside a new array definition, it provides a clean and modern way to combine multiple arrays into one. Here's how it works:

```
const animals = [ animal1, animal 2 ];
const otherAnimals = [ animal3, animal4 ];
const allAnimal = [...animals, ...otherAnimals]
```



Exercise 1:

To make our `printAllGames` function more reusable, we're going to refactor it so it can print *any* array of games, not just our main one.

Modify the `printAllGames` function:

Change its signature so it accepts one parameter, for example, `listOfGames`.

a. Update the `for...of` loop inside the function to iterate over this `listOfGames` parameter instead of the global `games` array.
b. Remove the line that prints the title from inside the function. A function should do one thing well, and this function's job is just to print games.



2. Create your friend's game list:
   a. Create a new array named `friendGames` and populate it with a few new game objects.
3. Update the function calls:
   a. Find all the places where you originally called `printAllGames()`. You must now pass the `games` array as an argument, like this: `printAllGames(games)`.
   b. Right before each of those calls, add a separate `addStatus()` call to print the appropriate title.

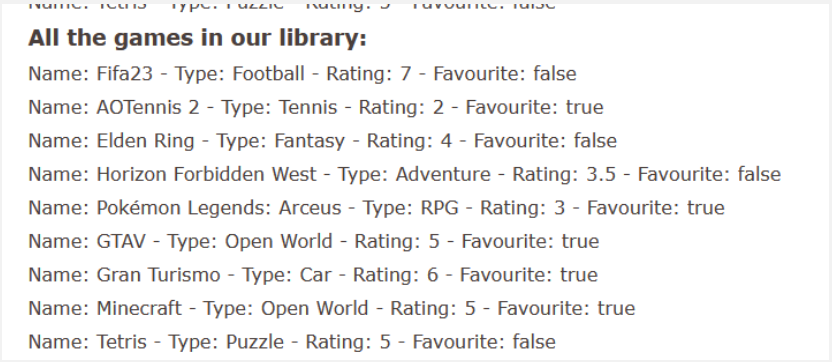
c. Finally, add a new title for your friend's collection and then call `printAllGames(friendGames)` to display their games on the page as well.



Exercise 2:

Display a combined "master list" of all games from both collections. First, add an appropriate title for this section (e.g., "All the games in our library"). Then, create a new array and use the spread operator (`...`) to merge your `games` and `friendGames` arrays into one.

Finally, pass this newly created, combined array as an argument to the `printAllGames` function to display everything at once.



2 Higher-order functions

In this section, we will continue to refactor our code by introducing higher-order functions. These are a core feature of JavaScript that allows you to write more expressive and concise code, often reducing multi-line loops to a single, elegant line. The exercises will involve both refactoring existing functions and adding new functionality.

You might feel like you're re-doing work, but this is far from the truth. Refactoring is an essential part of software development; great code is built iteratively, not all at once. This is a perfect moment to save a snapshot of your progress. By pushing your work to GitHub, you create a version history that allows you to revisit previous stages of your project at any time.

2.1 Iterating over arrays: the *forEach* method

Instead of a `for...of` loop, you can use the `.forEach()` method to iterate over an array. `forEach()` is a higher-order function that executes a function you provide—often called a callback function—once for every element in the array.

In the example below, `forEach` is used to find the oldest animal. It iterates through the `animals` array, and for each `animal`, it performs a check. The code compares the current `animal's` age to the age of the animal currently stored in the result variable. If the current animal is older, it replaces the one in `result`. This process ensures that by the end of the loop, the `result` variable holds the single oldest animal from the collection.

```
const getOldestAnimal = () => {
  let result = animals[0];
   animals.forEach((animal) => {
    if (animal.age > result.age) {
      result = animal;
    }
  });
  return result;
};
```



Refactor your code by replacing all `for...of` loops with the `forEach` method. To complete this exercise, ensure that in your `games.js` file, the `for...of` loop has been replaced by a `forEach` call in each of the following functions:

- printAllGames
- getAverageRating
- getHighestRating
- printFavoriteGames

2.2 Filtering arrays: the *filter* method

The `.filter()` method is another powerful higher-order function. Its job is to create a new array containing only the elements from the original array that pass a specific test you provide. For each element, your test function must return either `true` (to keep the element) or `false` (to discard it). Let's break down the example code. This function filters the `animals` array based on a search term (`chars`) and then prints the results:

```
const printAllAnimals = (chars) => {
  animals
    .filter((animal) => chars ? animal.name.toLowerCase()
                                .includes(chars.toLowerCase()) : true)
    .forEach((animal) => addStatus(gameToString(animal)));
}
```

Here's how it works:

- `.filter((animal) => ...)`: This starts the filtering process. For each `animal` in the array, it runs the test inside the parentheses.
- The test condition:
  `chars ? animal.name.toLowerCase().includes(chars.toLowerCase()) : true`
  - This is a compact `if-else` statement (a ternary operator). First, it checks if a search term (`chars`) was provided.
  - If `chars` exists: It performs a case-insensitive search to see if the animal's name `includes` the search term. It returns `true` if there's a match, and `false` otherwise.

- If `chars` is empty or null: It automatically returns `true`, meaning no animals are filtered out when there's no search term.

- `.forEach(...)`: The new, filtered array (containing only the animals that passed the test) is immediately passed to `forEach`, which then displays each result.



Exercise 1:

Refactor the `printFavoriteGames` function to use the `.filter()` method. First, use `.filter()` to create a new array that contains only the games marked as a favorite. Then, chain a `.forEach()` method to this new, filtered array to display the name of each game.

- The `printFavoriteGames` function is updated to use the `.filter()` method.
- The `.filter()` method should use your existing `isFavorite` helper function as its test condition.
- The `.forEach()` method is chained directly after the `.filter()` call. Its only responsibility is to print the game's name; any internal `if` statement should now be removed



Exercise 2:

Create a function, `printGamesAboveRating(games, minRating)`. Inside the function, use `.filter()` on the `games` to get games with a rating higher than `minRating`, and then chain `.forEach()` to display them. Finally, add a title to your page and call this function, passing it your main `games` array and a rating of 3.

These are all games with rating above 3

Name: Fifa23 - Type: Football - Rating: 7 - Favourite: false
Name: Elden Ring - Type: Fantasy - Rating: 4 - Favourite: false
Name: Horizon Forbidden West - Type: Adventure - Rating: 3.5 - Favourite: false
Name: GTAV - Type: Open World - Rating: 5 - Favourite: true
Name: Gran Turismo - Type: Car - Rating: 6 - Favourite: true

2.3 Transforming arrays: the *map* method

The `.map()` method is another essential higher-order function for arrays. Its purpose is to transform each element in an array and return a brand-new array containing the results. This is perfect for separating the logic of data transformation from data display. For example, you can create a clean, readable chain: first, use `.map()` to convert every game object into a formatted string, then use `.forEach()` to print each of those strings to the screen

```
games
  .map(gameToString)       // Step 1: Creates a new array of strings
  .forEach(addStatus); // Step 2: Prints each string from the new array
}
```



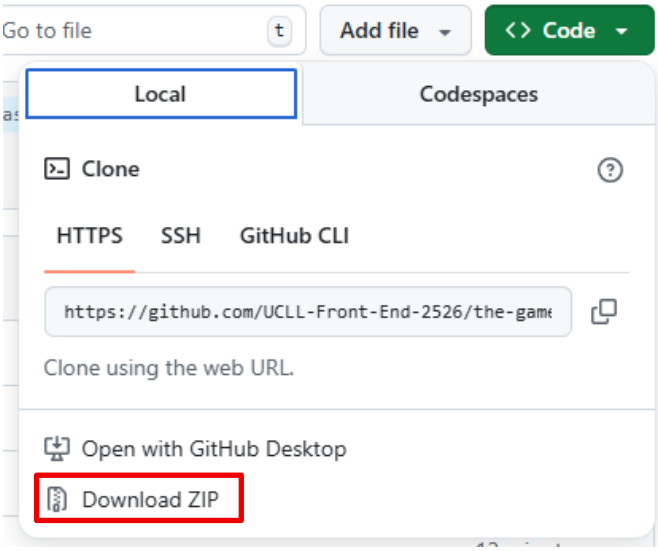
Refactor the `printAllGames` function to cleanly separate data transformation from the display logic. Modify the function to first use the `.map()` method to transform every game object into a formatted string using your `gameToString` helper function. Then, chain the `.forEach()` method directly to the result of the `.map()` call, passing the `addStatus` function to it to print each string to the screen.

3 Submitting your result

To complete the lab, you must submit a ZIP file of your project via the assignment on Toledo. Important: do not submit a link to your GitHub repository. Since you will continue to build on this same codebase in future labs, we need a "snapshot" of your code at this specific stage for grading.

The easiest way to create this snapshot is directly from GitHub:

- Navigate to your GitHub repository in your browser (e.g., https://github.com/UCLL-Front-End/the-games-library-jhnpck).
- Click the green "<> Code" button.
- Select Download ZIP from the dropdown menu.



Submit the downloaded ZIP file on Toledo to finalize your submission.