Verse of the day

Proverbs 21:5

The plans of the **diligent** lead to profit

as surely as **haste** leads to poverty.

FACULTY OF INFORMATION TECHNOLOGY

Universitas Advent Indonesia

Javascript Array

FACULTY OF INFORMATION TECHNOLOGY

Universitas Advent Indonesia

Outlines

1: **Introduction**

- What is an array in JavaScript?
- How is it used?
- Basic syntax and structure

2: **Creating and Initializing Arrays**

- Using the **Array** constructor
- Using array literals
- Initializing an array with a set of values

FACULTY OF INFORMATION TECHNOLOGY

Universitas Advent Indonesia

Outlines

3: **Accessing and Modifying Array Elements**

- Accessing elements by index
- Modifying elements by value
- Adding and removing elements from an array

4: **Array Methods**

- Common array methods: **push, pop, shift, unshift, splice, slice, concat, sort, reverse, indexOf, lastIndexOf, join**
- Iteration methods: **forEach, map, filter, reduce**

FACULTY OF INFORMATION TECHNOLOGY

Universitas Advent Indonesia

1. Introduction

What is an Array?

- Definition: A data structure that stores a collection of elements
- Example: An array of numbers, an array of strings, an array of objects, etc.

How is an Array Used?

- Storing a list of items
- Keeping track of a collection of related data
- Iterating over the elements in the array
- Storing data for a specific index

1. Introduction

Basic Syntax and Structure

- How to create an array: **var arr = new Array();** or **var arr = [ ];**
- Accessing elements by index: **arr[0]**
- The length property: **arr.length**

Example of an array of numbers, strings, and objects

```
var numbers = [1, 2, 3, 4, 5];
var strings = ["hello", "world", "javascript"];
var objects = [{name: "John", age: 30}, {name: "Mary", age: 25}];
```

This creates an array called "numbers" that contains the elements 1, 2, 3, 4, and 5. Another array called "strings" that contains the elements "hello", "world", "javascript". And another array called "objects" that contains two objects, each one with a name and age properties.

2. Creating and Initializing Arrays

Different ways to create an array in JavaScript:

1. **Using the Array Constructor**: The **new Array()** syntax creates an array object. The constructor can take a number of arguments, which will be used to initialize the elements of the array.

```
var arr1 = new Array(); // creates an empty array
var arr2 = new Array(1, 2, 3); // creates an array with the
values 1, 2, 3
var arr3 = new Array(3); // creates an array with 3 empty
elements
```

2. Creating and Initializing Arrays

2. **Using Array Literals:** The **[ ]** syntax creates an array object, with the elements specified inside the brackets. This is the most common and recommended way to create an array.

```
var arr4 = []; // creates an empty array
var arr5 = [1, 2, 3]; // creates an array with the values 1, 2, 3
```

3. **Using the Array.of() method: Array.of()** method creates an array with a set of values passed as arguments.

```
var arr6 = Array.of(1,2,3);  // creates an array [1,2,3]
```

2. Creating and Initializing Arrays

4. **Using the spread operator:** Spread operator can be used to convert an array like object to an array

```
let arrayLike = {0: 'a', 1: 'b', length: 2};
let arr7 = [...arrayLike]; // creates an array ['a', 'b']
```

It's worth noting that the different methods have slightly different behaviors and use cases, For example, using **new Array(3)** creates an array with 3 empty elements, which can be useful if you want to pre-allocate a certain number of elements in an array. But it is generally recommended to use **[ ]** or **Array.of()** to create an empty array instead.

3. Accessing and Modifying Array Elements

1. **Accessing elements by index**:

- In JavaScript, arrays are zero-indexed, meaning that the first element has an index of 0.
- To access an element at a specific index, you can use the square brackets notation and the index you want to access. For example,

```
var numbers = [1, 2, 3, 4, 5];
console.log(numbers[0]); // Output: 1
console.log(numbers[2]); // Output: 3
```

3. Accessing and Modifying Array Elements

2. Modifying elements by value:

- To modify an element at a specific index, you can use the assignment operator (=) and the square brackets notation. For example:

```
var numbers = [1, 2, 3, 4, 5];
numbers[2] = 100;
console.log(numbers); // Output: [1, 2, 100, 4, 5]
```

3. Accessing and Modifying Array Elements

3. Adding and removing elements from an array:

- To add an element to the end of an array, you can use the push() method. For example:

```
var numbers = [1, 2, 3, 4, 5];
numbers.push(6);
console.log(numbers); // Output: [1, 2, 3, 4, 5, 6]
```

- To remove an element from the end of an array, you can use the pop() method. For example:

```
var numbers = [1, 2, 3, 4, 5];
numbers.pop();
console.log(numbers); // Output: [1, 2, 3, 4]
```

3. Accessing and Modifying Array Elements

- To add an element to the beginning of an array, you can use the unshift() method. For example:

```
var numbers = [1, 2, 3, 4, 5];
numbers.unshift(0);
console.log(numbers); // Output: [0, 1, 2, 3, 4, 5]
```

- To remove an element from the beginning of an array, you can use the shift() method. For example:

```
var numbers = [1, 2, 3, 4, 5];
numbers.shift();
console.log(numbers); // Output: [2, 3, 4, 5]
```

4. Javascript Array methods

Here are some common array methods in JavaScript:

- **push() and pop()**
- **shift() and unshift()**
- **splice() and slice()**
- **map()**
- **reduce()**
- **filter()**
- **sort**
- **concat()**

4. Javascript Array methods

Here are some common array methods in JavaScript:

push() and pop()

push() method: Adds one or more elements to the end of an array and returns the new length of the array.

```
var arr = [1, 2, 3];
arr.push(4); // returns 4, the new length of the array is [1, 2, 3, 4]
```

pop() method: Removes the last element from an array and returns that element.

```
var arr = [1, 2, 3];
arr.pop(); // returns 3, the new length of the array is [1, 2]
```

shift() and unshift()

shift() method: Removes the first element from an array and returns that element.

```
var arr = [1, 2, 3];
arr.shift(); // returns 1, the new length of the array is [2, 3]
```

unshift() method: Adds one or more elements to the beginning of an array and returns the new length of the array.

```
var arr = [1, 2, 3];
arr.unshift(0); // returns 4, the new length of the array is [0,
1, 2, 3]
```

splice() and slice()

splice() method: Adds or removes elements from an array. It takes 3 arguments: the index to start at, the number of elements to remove, and any elements to add.

```
var arr = [1, 2, 3, 4, 5];
arr.splice(1, 2); // returns [2, 3], the new length of the array is
[1, 4, 5]
arr.splice(1, 0, 2, 3); // returns []
```

slice() method: Extracts a section of an array and returns a new array. It takes 2 arguments: the starting index and the ending index (not including the element at the ending index).

```
var arr = [1, 2, 3, 4, 5];
arr.slice(1, 3); // returns [2, 3], the original array is
unchanged
```

splice()

The **splice()** method in JavaScript is used to add or remove elements from an array. It can be used to insert new elements into an array at a specific position, or to remove a certain number of elements starting at a specific position. The syntax for using the **splice()** method is as follows:

```
array.splice(start, deleteCount, item1, item2, ...)
```

- **start** is the index at which to start changing the array.
- **deleteCount** is the number of elements to remove.
- **item1**, **item2**, ... are the elements to add to the array

Javascript Array

```
let fruits = ["banana", "orange", "apple", "mango"];

// Removing the first element (banana)
fruits.splice(0, 1);
console.log(fruits); // ["orange", "apple", "mango"]

// Adding an element (kiwi) at the beginning of the array
fruits.splice(0, 0, "kiwi");
console.log(fruits); // ["kiwi", "orange", "apple", "mango"]

// Replacing the second element (orange) with a new element (lemon)
fruits.splice(1, 1, "lemon");
console.log(fruits); // ["kiwi", "lemon", "apple", "mango"]

// Removing the last two elements (apple and mango)
fruits.splice(-2, 2);
console.log(fruits); // ["kiwi", "lemon"]

// Adding multiple elements (grapes, strawberry, peach) at the end of the
array
fruits.splice(fruits.length, 0, "grapes", "strawberry", "peach");
console.log(fruits); // ["kiwi", "lemon", "grapes", "strawberry", "peach"]
```

slice()

The **slice()** method in JavaScript is used to extract a portion of an array and return it as a new array. The syntax for using the **slice()** method is as follows:

```
array.slice(start, end)
```

- **start** is the index at which to start the new array.
- **end** is the index at which to end the new array.

Javascript Array

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

// Extracting elements from index 2 to 5
let subArray = numbers.slice(2, 6);
console.log(subArray); // [3, 4, 5, 6]

// Extracting elements from index 3 to the end
let subArray2 = numbers.slice(3);
console.log(subArray2); // [4, 5, 6, 7, 8, 9, 10]

// Extracting elements from the beginning to index 4
let subArray3 = numbers.slice(0,4);
console.log(subArray3); // [1, 2, 3, 4]

let words = ['hello', 'world', 'this', 'is', 'javascript']

// Extracting elements from index 1 to end of array
let subArray4 = words.slice(1);
console.log(subArray4); // ['world', 'this', 'is', 'javascript']

// Extracting elements from index 0 to index 3
let subArray5 = words.slice(0,4);
console.log(subArray5); // ['hello', 'world', 'this', 'is']
```

Javascript Array

map()

.map() is a method that is used to create a new array with the results of calling a provided function on every element in the original array. It is commonly used to transform or change the elements in an array in some way.

Here's an example of how to use the **.map()** method:

```
const numbers = [1, 2, 3, 4, 5];
const doubledNumbers = numbers.map(function(number) {
  return number * 2;
});
console.log(doubledNumbers);  // [2, 4, 6, 8, 10]
```

In this example, we have an array of numbers**, [1, 2, 3, 4, 5]**, and we want to create a new array with each number doubled. The **.map()** method takes a callback function as its argument, which is applied to each element in the original array. In this case, the callback function is **function(number)** { return number * 2; }, which doubles the value of each element in the array. The result of calling the .map() method is a new array, **[2, 4, 6, 8, 10**], which contains the original array's elements, but each element has been doubled.

FACULTY OF INFORMATION TECHNOLOGY
Universitas Advent Indonesia

Javascript Array

You can also use arrow function notation to make your code cleaner:

```
const numbers = [1, 2, 3, 4, 5];
const doubledNumbers = numbers.map(number => number * 2);
console.log(doubledNumbers);  // [2, 4, 6, 8, 10]
```

The **.map()** method also passes 3 arguments to the callback function, the currentValue, index and the original array

```
const numbers = [1, 2, 3, 4, 5];
const doubledNumbers = numbers.map(function(currentValue, index,
array) {
  console.log(currentValue, index, array);
  return currentValue * 2;
});
console.log(doubledNumbers);  // [2, 4, 6, 8, 10]
```

FACULTY OF INFORMATION TECHNOLOGY
Universitas Advent Indonesia

Javascript Array

The **.map()** method is also useful when you need to transform an array of objects in some way.

```
const users = [
  { name: 'Alice', age: 25 },
  { name: 'Bob', age: 30 },
  { name: 'Charlie', age: 35 }
];
const userNames = users.map(user => user.name);
console.log(userNames);  // ['Alice', 'Bob', 'Charlie']
```

In this example, we have an array of user objects, each of which has a name and age property. We use the **.map()** method to create a new array containing only the name of each user.

map()

In summary, the **.map()** method is:

- a powerful tool for transforming arrays and creating new arrays based on the elements of the original array.
- It is a functional programming technique that can make your code more readable, expressive, and maintainable.

.filter()

The **.filter()** method is used to create a new array with all elements that pass a certain test.

It is used to filter out elements from an array based on a certain condition, leaving only the elements that satisfy that condition in the new array.

Here's an example of how to use the **.filter()** method:

```
const numbers = [1, 2, 3, 4, 5];
const evenNumbers = numbers.filter(function(number) {
  return number % 2 === 0;
});
console.log(evenNumbers);  // [2, 4]
```

Javascript Array

Here's an example of how to use the **.filter()** method:

```
const numbers = [1, 2, 3, 4, 5];
const evenNumbers = numbers.filter(function(number) {
  return number % 2 === 0;
});
console.log(evenNumbers);  // [2, 4]
```

In this example, we have an array of numbers, **[1, 2, 3, 4, 5]**, and we want to create a new array with only the even numbers. The **.filter()** method takes a callback function as its argument, which is applied to each element in the original array. The callback function in this case is **function(number)** { return number % 2 === 0; }, which checks if the number is even (i.e, if the remainder of the number divided by 2 is 0). The **.filter()** method creates a new array and only includes elements for which the callback function returned a truthy value. In this case, the new array is **[2, 4]**, which contains only the even numbers from the original array.

You can also use arrow function notation to make your code cleaner:

```
const numbers = [1, 2, 3, 4, 5];
const evenNumbers = numbers.filter(number => number % 2 === 0);
console.log(evenNumbers);  // [2, 4]
```

FACULTY OF INFORMATION TECHNOLOGY
Universitas Advent Indonesia

Javascript Array

The **.filter()** method also passes 3 arguments to the callback function, the currentValue, index and the original array

```
const numbers = [1, 2, 3, 4, 5];
const evenNumbers = numbers.filter(function(currentValue, index, array) {
  console.log(currentValue, index, array);
  return currentValue % 2 === 0;
});
console.log(evenNumbers);  // [2, 4]
```

FACULTY OF INFORMATION TECHNOLOGY
Universitas Advent Indonesia

Javascript Array

The **.filter()** method is also useful when working with arrays of objects. Here is an example that filters out users that are over 30 years old:

```
const users = [
  { name: 'Alice', age: 25 },
  { name: 'Bob', age: 30 },
  { name: 'Charlie', age: 35 },
  { name: 'David', age: 40 }
];
const usersUnder31 = users.filter(user => user.age <= 30);
console.log(usersUnder31);
/* Output :
[
  { name: 'Alice', age: 25 },
  { name: 'Bob', age: 30 },
]
*/
```

In this example, **usersUnder31** is an array containing only the users that are under 31 years old.

FACULTY OF INFORMATION TECHNOLOGY
Universitas Advent Indonesia

.filter()

.**filter()** in summary:

- the **.filter()** method is a powerful tool for filtering out elements from an array that don't satisfy a certain condition.
- It is a functional programming technique that can make your code more readable, expressive, and maintainable.
- It creates a new array based on the original array and the elements that pass the test defined in the callback function.

.reduce()

The **.reduce()** method is used to apply a function against an accumulator and each element in the array (from left to right) to reduce it to a single value. It can be used for many purposes such as calculating the sum of an array of numbers, concatenating an array of strings, or flattening an array of arrays.

Here's an example of how to use the .**reduce()** method to calculate the sum of an array of numbers:

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce(function(accumulator, currentValue) {
  return accumulator + currentValue;
}, 0);
console.log(sum); // 15
```

Javascript Array

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce(function(accumulator, currentValue) {
  return accumulator + currentValue;
}, 0);
console.log(sum); // 15
```

In this example, we have an array of numbers, **[1, 2, 3, 4, 5],** and we want to calculate the sum of all elements in the array. The **.reduce()** method takes a callback function as its first argument, which is applied to each element in the array. The callback function takes two arguments, **accumulator** and **currentValue**. The **accumulator** is the value that is accumulated as the function iterates over the elements, and the **currentValue** is the current element being processed in the array. In this example, the accumulator starts with a value of 0 and it keeps adding each currentValue in the array, one by one, finally returns the total sum.

You can also use arrow function notation to make your code cleaner:

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce((acc, cur) => acc + cur, 0);
console.log(sum); // 15
```

FACULTY OF INFORMATION TECHNOLOGY
Universitas Advent Indonesia

Javascript Array

The **.reduce()** method also passes 3 arguments to the callback function, the accumulator, currentValue and the index of the currentValue

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce(function(accumulator, currentValue, currentIndex)
{
  console.log(accumulator, currentValue, currentIndex);
  return accumulator + currentValue;
}, 0);
console.log(sum); // 15
```

The **reduce()** function is used to iterate over an array and reduce it to a single value, by applying a callback function to each element. The callback function takes three arguments:

accumulator: The accumulated value, which is initially set to the value of the initialValue parameter.

currentValue: The current element being processed in the array.

currentIndex: The index of the current element being processed in the array.

The 0, is the init, which is the starting value for the accumulator. In this case, it is set to 0, so the accumulator starts at 0 and then adds each number in the numbers array to it, one at a time. In the end, it **returns the sum** of all the numbers in the array, which is **15**.

FACULTY OF INFORMATION TECHNOLOGY
Universitas Advent Indonesia

Javascript Array

Here is an example that concatenates all the names of users in an array

```
const users = [
  { name: 'Alice' },
  { name: 'Bob' },
  { name: 'Charlie' }
];
const concatenatedName = users.reduce((acc, user) => acc + ' ' + user.name, '');
console.log(concatenatedName); // "Alice Bob Charlie"
```

In this example, we have an array of users, and we want to concatenate all the names of the users. The **.reduce()** method iterates over the array of users, in each iteration it concatenates the current user name with the accumulator and the accumulator keeps changing until the end of the array, finally, it returns the concatenated string.

.reduce()

In summary, the **.reduce()** method is :

- a powerful tool for reducing an array of values down to a single value.
- It can be used for a wide range of tasks such as finding the sum of an array, concatenating an array of strings, or flattening an array of arrays.
- It's an important functional programming method in javascript, it helps to make your code more readable.

.sort()

The **.sort()** method is used to sort the elements of an array in place and return the sorted array. By default, the sort order is based on the Unicode code point values of each element, which means that the elements are converted to strings before being sorted.

Here's an example of how to use the **.sort()** method to sort an array of numbers:

```
const numbers = [3, 1, 5, 2, 4];
numbers.sort();
console.log(numbers);  // [1, 2, 3, 4, 5]
```

In this example, the **.sort()** method sorts the elements of the numbers array in ascending order and modifies the original array.

Javascript Array

You can also use a custom sorting function as the parameter for the **.sort()** method to define your own sorting logic

```
const numbers = [3, 1, 5, 2, 4];
numbers.sort(function(a, b) {
  return a - b;
});
console.log(numbers);  // [1, 2, 3, 4, 5]
```

The sorting function takes two arguments, a and b and it should return a negative, zero or positive value depending on the order of the elements. if it returns a negative value that means a should come before b if it returns a positive value that means b should come before a if it returns 0 that means the order of a and b doesn't matter.

Javascript Array

You can also use arrow function notation to make your code cleaner:

```
const numbers = [3, 1, 5, 2, 4];
numbers.sort((a, b) => a - b);
console.log(numbers);  // [1, 2, 3, 4, 5]
```

When it comes to sorting an array of objects, you have to pass a sorting function, which compares the objects based on the certain property:

```
const users = [
  { name: 'Bob', age: 30 },
  { name: 'Charlie', age: 35 },
  { name: 'Alice', age: 25 },
];
```

Javascript Array

When it comes to sorting an array of objects, you have to pass a sorting function, which compares the objects based on the certain property:

```
const users = [
  { name: 'Bob', age: 30 },
  { name: 'Charlie', age: 35 },
  { name: 'Alice', age: 25 },
];

users.sort(function(a, b) {
  if (a.name < b.name) {
    return -1;
  }
  if (a.name > b.name) {
    return 1;
  }
  return 0;
});
console.log(users);
```

In this example, the **users** array is sorted in ascending order based on the name property.

It's important to note that the **.sort()** method modifies the original array and doesn't create a new one.

```
/* Console Output:
[
  { name: 'Alice', age: 25 },
  { name: 'Bob', age: 30 },
  { name: 'Charlie', age: 35 },
]
*/
```

FACULTY OF INFORMATION TECHNOLOGY
Universitas Advent Indonesia

.sort()

In summary, the **.sort()** method is:

- used to sort the elements of an array in place,
- by default it sorts the elements of the array in ascending order based on their Unicode code point values,
- but it can take a sorting function as an argument to define a custom sorting logic.
- It can be useful when you need to sort an array in a specific order and it can be used for arrays of both primitives and objects.

.concat()

The **.concat()** method is used to merge two or more arrays together into a new array. It does not modify the original arrays, but instead returns a new array that contains the elements from the original arrays. The new array is a combination of the elements of the original arrays in the order they were passed to the method.

Here's an example of how to use the **.concat()** method:

```
const array1 = ['a', 'b', 'c'];
const array2 = [1, 2, 3];
const array3 = array1.concat(array2);
console.log(array3); // ['a', 'b', 'c', 1, 2, 3]
```

In this example, we have two arrays, **array1** and **array2,** and we want to merge them together into a new array, **array3**. The **.concat()** method is called on **array1** and passed **array2** as an argument. The method creates a new array, **array3**, which contains all the elements from **array1** followed by all the elements from **array2**.

Javascript Array

It is also possible to concatenate multiple arrays by passing them as arguments to the **.concat()** method:

```
const array1 = ['a', 'b', 'c'];
const array2 = [1, 2, 3];
const array3 = [4, 5, 6];
const array4 = array1.concat(array2, array3);
console.log(array4); // ['a', 'b', 'c', 1, 2, 3, 4, 5, 6]
```

In this example, we have three arrays: **array1**, **array2**, and **array3**, and we want to merge them all into one array, **array4**. The .**concat()** method is called on array1 and passed on **array1**, and **array3**.

concat()

In summary, the **.concat()** method is:

- The concat() method in JavaScript is used to merge two or more arrays together, creating a new array that contains all elements from original arrays without modifying the original arrays.
- It can be used to merge multiple arrays of different types and merge a single value or multiple values with an array.
- It creates a new array that is separate from the original arrays and it's chainable.
- The concatenated array is a new array with new memory address, so any modification made on the original arrays will not affect the concatenated array.

Thank you for attending this presentation, I hope you have a better understanding of JavaScript arrays and their uses. If you have any questions, please feel free to reach out to me for further clarification.

FACULTY OF INFORMATION TECHNOLOGY

Universitas Advent Indonesia