JavaScript Tutorial

https://playcode.io/javascript/map

https://www.w3schools.com/js/

1. for loop

The for loop is one of the most widely used loops in and is ideal when you know in advance how many times you want to repeat a block of code. It is particularly useful when working with **arrays** or any data structures where you know the **index** or length.

When to use:

- When you know the number of iterations in advance.
- When iterating through **arrays** or **arrays-like objects** (e.g., NodeLists).
- When you need access to the **index** of the array.

Syntax:

```
for (let i = 0; i < array.length; i++) {
  // Code to execute for each iteration
}
```

Example:

Using a for loop to iterate over an **array** of numbers:

```
const numbers = [1, 2, 3, 4, 5];
for (let i = 0; i < numbers.length; i++) {
  console.log(numbers[i]); // Logs each number in the array
}
```

Use Cases:

- When you need the **index** of the array (e.g., accessing elements by index).
- When you need to loop through a specific range of elements (e.g., looping over indices 1 through 4).

---

2. for...of loop

The for...of loop is a more modern and concise way to iterate over **iterables** like arrays, strings, maps, sets, etc., without needing the index. It's cleaner and easier to read, especially when you don't need the index of the elements.

When to use:

- When iterating over **arrays**, **strings**, **sets**, or **maps**.
- When you do **not need access to the index** of the array or other iterable.
- When you want cleaner, more readable code.

Syntax:

```
for (const item of iterable) {
  // Code to execute for each item
}
```

Example:

Using for...of to iterate over an **array**:

```
const numbers = [1, 2, 3, 4, 5];
for (const number of numbers) {
  console.log(number); // Logs each number in the array
}
```

Use Cases:

- When you just need to process the **values** in an iterable (e.g., array elements, string characters, or map values).
- More readable and concise for simple iterations over arrays.

---

3. for...in loop

The for...in loop is typically used to iterate over **enumerable properties** (keys) of an object or the **indices** of an array. It is less common for arrays because it may not guarantee the order of iteration and is intended more for objects.

When to use:

- When iterating over **object properties** (i.e., key-value pairs).

- When working with arrays **in a non-index-specific way** (though for...of is typically better for arrays).

Syntax:

```
for (const key in object) {
  // Code to execute for each key in the object
}
```

Example:

Using for...in to iterate over an **object**:

```
const person = {
  name: "John",
  age: 30,
  job: "Developer"
};

for (const key in person) {
  console.log(key, person[key]); // Logs the key and value
}
```

Use Cases:

- Iterating over the **properties of an object** (not the values or elements of an array).
- **Not recommended for arrays** in most cases, as it does not guarantee the order of indices.

---

4. while loop

The while loop repeats a block of code **while a condition is true**. It is useful when the number of iterations is **unknown** or you want to loop until a specific condition is met.

When to use:

- When you **don't know the number of iterations** in advance.
- When you want to **loop until a certain condition** becomes false.
- Typically used for situations where you are checking a **condition before each iteration**.

Syntax:

```
while (condition) {
  // Code to execute while the condition is true
}
```

Example:

Using while loop to print numbers while they are less than 5:

```
let i = 0;
while (i < 5) {
  console.log(i);
  i++; // Increment to avoid infinite loop
}
```

Use Cases:

- When the **number of iterations** is not known and depends on some dynamic condition.
- Useful for implementing **loops that process data until a condition is met**, like waiting for a user input or checking for a specific value.

---

5. do...while loop

The do...while loop is similar to the while loop, except that it guarantees that the loop will run at least **once**, even if the condition is false initially.

When to use:

- When you want to run the loop **at least once**, and then continue as long as a condition is true.
- Useful when you need to **perform an action at least once** before validating the condition.

Syntax:

```
do {
  // Code to execute at least once
} while (condition);
```

Example:

Using do...while to print numbers at least once:

```
let i = 0;
do {
  console.log(i);
  i++;
} while (i < 5);
```

Use Cases:

- When you want to ensure that the block of code runs **at least once** regardless of the condition (e.g., showing a message to the user at least once).

---

6. Array.prototype.forEach()

The forEach() method executes a given function **once for each array element**. It is useful when you need to apply a function to every element of an array, but it is **not breakable** (i.e., you cannot exit the loop early).

When to use:

- When you need to **perform an action** on each element of an array.
- When you don't need the **index** or need to stop the iteration.

Syntax:

```
array.forEach((item, index, array) => {
  // Code to execute for each item
});
```

Example:

Using forEach() to log each element of an array:

```
const numbers = [1, 2, 3, 4, 5];
numbers.forEach((number) => {
  console.log(number);
});
```

Use Cases:

- When you need to **perform an operation** on every element of an array.
- Not suitable for operations that require **early exits** (use break or return), as forEach() does not support that.

Summary Table

Loop Type	When to Use	Example Use Cases
for loop	When you know the number of iterations (e.g., iterating through arrays or ranges).	Iterating through arrays, performing actions with indices.
for...of	When iterating over arrays, strings, sets, or other iterable objects, and you don't need the index.	Processing array elements or string characters.
for...in	When iterating over object properties (keys).	Iterating over object keys (not recommended for arrays).
while	When the number of iterations is unknown and depends on a condition that could change during execution.	Loops that continue until a condition is met (e.g., user input).
do...while	When you want the loop to execute at least once, regardless of the condition.	Executing code at least once before checking the condition.
forEach()	When you want to apply a function to every element in an array.	Applying functions to each array element without needing an index.

Best Practices

- Use **for** or **for...of** when working with arrays and you need to either iterate over each element or use the index.
- Use **for...in** when working with **objects** to iterate over keys.
- Use **while** or **do...while** when the number of iterations depends on a dynamic condition.
- **forEach()** is great for simple, non-iterable-based array processing but avoid it if you need **early termination** or performance optimization in large arrays.

The switch statement in is used when you need to compare one value against multiple possible conditions (cases). It's an alternative to using multiple if-else conditions, especially when the comparison involves the same variable or expression being checked against different possible values.

When to Use a switch Statement

1. **When you have multiple possible values to compare**: If you need to check one variable or expression against several different values, a switch is cleaner and more readable than a series of if-else statements.
2. **When the variable is being compared to specific constant values**: switch works best when you're comparing a single variable (or expression) to specific, **discrete** values. It is not ideal when comparing ranges, complex conditions, or when you need to check various variables.
3. **When you want cleaner, more readable code**: If you have a long series of if-else conditions that are all checking the same variable, a switch can make your code easier to read and maintain.
4. **When you want to perform different actions based on different values of the same variable**: A switch allows you to handle multiple potential values for a single expression with much less code and better organization than using multiple if-else statements.

Basic Syntax of a switch Statement

```
switch (expression) {
 case value1:
  // Code to execute if expression === value1
  break;

 case value2:
  // Code to execute if expression === value2
  break;

 case value3:
  // Code to execute if expression === value3
  break;
 default:
   // Code to execute if no case matches
}
```

- expression: The value you want to check.
- case value: The value you're comparing expression against. If they match, the associated block of code runs.
- break: Ends the switch block. Without break, the code will "fall through" and execute the next case (which is usually not the desired behavior).
- default: A fallback option that runs if none of the case conditions match.

When Not to Use switch

1. **When you need to compare conditions that involve ranges or complex expressions**: If you need to evaluate something like a range of values (e.g., numbers between 1 and 10), switch isn't suitable. You would likely use if-else conditions in such cases.

Example of a case where if-else is better:

```
if (value >= 1 && value <= 10) {
  // Handle values between 1 and 10
} else if (value > 10 && value <= 20) {
  // Handle values between 11 and 20
} else {
  // Handle other cases
}
```

2. **When comparing multiple variables**: If you're checking different variables (e.g., if (x === 1 && y === 2)), then switch is not suitable. You would need to use if-else conditions.

Examples of Using switch

1. Using switch with numbers or simple values:

If you have a variable and you want to perform different actions based on its value, a switch makes the code cleaner.

```
const day = 3;


switch (day) {
 case 1:

  console.log("Monday");

  break;

 case 2:

  console.log("Tuesday");

  break;

 case 3:

  console.log("Wednesday");

  break;

 case 4:

  console.log("Thursday");

  break;

 case 5:

  console.log("Friday");

  break;

 case 6:

  console.log("Saturday");

  break;

 case 7:

  console.log("Sunday");

  break;

 default:

  console.log("Invalid day");

}
```

In this case, the switch checks the value of day and matches it with the appropriate case (e.g., case 3 outputs "Wednesday"). If no match is found, the default case is executed.

2. Using switch for handling strings:

```
const fruit = 'apple';


switch (fruit) {
 case 'banana':
  console.log('Banana is yellow.');
  break;
 case 'apple':
  console.log('Apple is red or green.');
  break;
 case 'grape':
  console.log('Grape is purple or green.');
  break;
 default:
  console.log('Unknown fruit');
}
```

Here, switch is used to match the string value (fruit) against different possible fruit names. The output would be 'Apple is red or green.' because fruit === 'apple'.

3. Fall-through behaviour in switch:

If you don't want to use break statements, you can take advantage of **fall-through behaviour**, where the code from one case continues to the next case. However, this behaviour should be used carefully to avoid unintended results.

```
const fruit = 'apple';


switch (fruit) {
 case 'banana':
 case 'apple':
  console.log('This is a fruit!');
  break;
 case 'carrot':
  console.log('This is a vegetable.');
  break;
 default:
  console.log('Unknown item');
}
```

In this example, both 'banana' and 'apple' will log 'This is a fruit!' because both cases fall through to the same block of code. carrot would log 'This is a vegetable.'.

4. Using switch with expressions (like true):

You can use switch with an expression instead of a specific value. One common pattern is using switch(true) for handling complex condition checking.

```
const value = 10;


switch (true) {
  case (value > 0 && value <= 10):
    console.log('Value is between 1 and 10');
    break;
  case (value > 10 && value <= 20):
    console.log('Value is between 11 and 20');
    break;
  default:
    console.log('Value is out of range');
}
```

This pattern allows switch to behave like an if-else ladder, which can sometimes be more readable than multiple if-else statements.

Advantages of switch Over if-else

1. **Readability**: When checking a single variable against multiple values, switch often leads to clearer, more readable code than long chains of if-else statements.
2. **Efficiency**: While not always true, switch can be more efficient than multiple if-else statements, especially when there are many conditions. This is because many engines optimize switch statements for better performance.
3. **Maintainability**: As your conditions grow, a switch can help avoid a "pyramid of doom" that can occur with nested if-else blocks.

Summary: When to Use switch

Use switch when:

- You have a **single variable or expression** that needs to be compared to multiple possible values.
- The comparisons are **discrete** and not complex (e.g., ranges, object properties).

- You want to improve **code readability** and simplify multiple if-else conditions.

Avoid switch when:

- You need to compare **complex conditions** (e.g., ranges, expressions) or multiple variables.
- You need more flexibility than what switch provides (e.g., checking complex boolean conditions or different expressions).

By using the right loop or control flow structure (if-else vs switch), you can make your code both more efficient and more readable.