

## Going deeper in to JavaScript

We will be covering:

- [Operators](#)
- [Operator precedence](#)
- [Associativity](#)
- [Implicit vs. Explicit type coercion](#)
- [Static vs. dynamic typing](#)
- [Weak vs. strong typing](#)

### **Operators**

You've used these before! They are just special functions written in a different syntax (e.g.  $2 + 2$  is the addition operator). This is called infix notation, where the operator is between the operands (aka the parameters). This is the most common notation for operators.

Another notation you may see is prefix – where the operator comes before the operand. E.g. the logical not operator `!` (e.g. `!true`)

Postfix notation – where it comes after! E.g. the `++` at the end of a for loop (`for (let i = 0; i < 2; i++)`)

You can also get unary operator (work with only 1 operand), binary operators (work with 2 operands) and ternary operators (takes 3 operands and is like an alternative to an “if” statement e.g. `isMember ? '$2.00' : '$10.00'`).

### **Operator precedence**

We are going to discover some situations where the output to the console is different to what you'd expect or a bit weird!

We've turned “eager evaluation” off so we're not seeing the answer in advance, we're working in the console in chrome.

Operator precedence is the order that the operators are parsed in (like BODMAS in maths).

List can be found here: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator\\_Precedence#table](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence#table)

Multiplication has higher precedence than addition so  $5 + 4 * 2$  should equal 13 as  $4 * 2 = 8$  then you add 5.

If you wanted it to do  $5 + 4$  first, then group it in smooth brackets  $(5 + 4) * 2$  equals 18

Parentheses have the highest precedence.

Use the above table to help debug when you're getting an unexpected result.

## Associativity

When precedence doesn't help us, associativity comes in to play. Associativity is whether the code is being left-to-right or right-to-left. This is detailed in the above precedence table and differs depending on the operator!

e.g.

```
let a = 1;  
let b = 2;  
let c = 3;  
a = b = c;
```

a, b and c are all now equal to 3!

In the above case, it is evaluating from right to left, as that's how associativity works for the assignment operator. So it evaluates what c is, then assigns it to b, then because everything to the right of "a =" is 3, it also assigns a to 3.

## Implicit vs Explicit coercion

```
1 < 2 returns true  
1 < 2 < 3 also returns true  
3 > 2 returns true  
3 > 2 > 1 returns false!!!
```

Returns false because  $3 > 2$  is true, then true is converted to 1 in inequality operators, therefore  $true > 1$  becomes  $1 > 1$ , which is false. So would need to add parentheses to make things clear:  $(3 > 2) > 1$ .

This also means the 2<sup>nd</sup> line of code above is true but not for the reason we think – it's because  $(1 < 2) = true$  which resolves to 1, then  $1 < 3$  is also true.

This is implicitly coercing the types of our operators! And it's doing it behind our backs – sneaky JS!

We've seen this before when we do:

```
5 + 5 equals 10 (number)  
"5" + "5" equals "55" (string)  
5 + "5" equals "55" (string) – so it has implicitly converted the number to a string
```

This happens because JavaScript is a weakly typed language, this doesn't happen in strongly typed language (such as python – if you do  $3 > 2 > 1$  in python, it will return true!). This flexibility can be really powerful but can also mean software can be prone to weird bugs.

Helena's favourite example:

```
let fruit = ['b' + 'a' + + 'x' + 'a'].toString()  
console.log(fruit.toLowerCase())
```

Gives you banana!!!

This is because of the + + 'x' – this gives you NaN (not a number) so it is b+a+NaN+a!

Explicit coercion would be doing something like Number("1") which would give you 1 that is of type integer. This is why in server/database in week 5, we had to coerce the id of a URL (/id) to a number in order to use it to query the database with Number(req.params.id)

## Static/Dynamic Typing

Static – when you set a variable to a type, you CANNOT change it (e.g. Java)

Dynamic – you can change the type (e.g. JS, Python)

## Strong/Weak Typing

In python Str = 5 + "hello" would throw an error – this is **strong typing**

In PHP, \$str = 5 + "hello" would equal 5 because "hello" is implicitly casted to 0 – this is **weak typing**

See this StackOverflow (particularly : <https://stackoverflow.com/questions/1517582/what-is-the-difference-between-statically-typed-and-dynamically-typed-languages>)

And this article that is specific to JavaScript: <https://www.freecodecamp.org/news/js-type-coercion-explained-27ba3d9a2839/>

Then do quiz: <https://jsisweird.com/>

Then find JS meme (or come up with your own meme) that covers these oddities and post it in the #random channel to try and make the coaches laugh

# TypeScript


To begin with, watch these videos:

- TS in 100 seconds: <https://www.youtube.com/watch?v=zQnBQ4tB3ZA>
- TS basics: <https://www.youtube.com/watch?v=ahCwqrYpluM>
- TS pros and cons: <https://www.youtube.com/watch?v=D6or2gdrHRE>

My takeaways in answer to the questions we were asked:

- What is TypeScript?
  - A strongly typed, static typed programming language, based on JavaScript
- What problem does it solve?
  - The weirdness of what happens with weakly typed languages when you try to work with different data types
  - Gives you greater feedback in development on errors in your code
  - “Would you rather have “silly” errors in development, or insanity causing problems in production”
  - Gives access to tuples that are very powerful and used in other languages such as Python
- What are some of the pros of TypeScript?
  - Can reduce bugs in production as can pick them up sooner than JS (due to static typing)
  - Can integrate better with your IDE
    - Powerful auto-imports
    - Means you can get autocomplete for all things (as you’ve told TS what things to expect)
  - Minimal learning curve as is basically JavaScript with declared types!
  - Backwards compatible with old version of JS (e.g. ES3)
  - Supports Future JS & Beyond
  - Customisable using tsconfig.json (e.g. what version of JS your code gets compiled to using “target”, “watch”:true which recompiles on changes, “lib” (to use certain libraries such as DOM))
  - Can set up custom types (e.g. Style = ‘bold’ | ‘italic’) and even customise object structure (e.g. interface Person {first: string; last: string;}) and have optional types
  - Very useful in large scale projects (as types always defined so you don’t need to go digging in the code to know how use aspects of it)
  - Can still make it give you back a JS file so can be used in browsers
- What are some of the cons of TypeScript?
  - Could make development take longer as you always have to declare type (implicitly or explicitly)
    - E.g. not always useful in smaller projects
  - Could make you a lazy developer! (computer always catching bugs for you)
  - Can take some time to learn the intricacies
    - E.g. Error messages can be cryptic
  - Typescript won’t work in the browser or node (so need compiler to run)
    - This can add complexity as you need to manage this process yourself

- Not all JS libraries come with types in (so you don't always get the benefit)
- Needs setting up to work how you want, using tsconfig.json
- Doesn't autocomplete for certain third-party libraries (e.g. lodash) like JavaScript does
- Static typing doesn't give you any flexibility with your data
  - Could make you use many more variables if your data is changing type (e.g. string to number)


```
demo > payment > JS index.js >  grantAccessToService
1  export function grantAccessToService(payment) {
2      const shouldAllowAccessToService = payment.isValid && payment.isAuthorized;
3
4      const GOLD_TIER = 10000;
5      const SILVER_TIER = 5000;
6      const BRONZE_TIER = 2000;
7
8      if (shouldAllowAccessToService) {
9          if (payment.amount > GOLD_TIER) {
10             return "Access to gold tier granted!";
11          } else if (payment.amount > SILVER_TIER) {
12             return "Access to silver tier granted!";
13          } else if (payment.amount > BRONZE_TIER) {
14             return "Access to bronze tier granted!";
15          }
16      }
17      return "Access not granted";
18  }
19
20  const customerPayment = {
21      customerId: 501500090595,
22      isValid: true,
23      isAuthorised: true,
24      amount: 4500,
25      paymentMethod: "CREDIT_CARD",
26      address: ["123 Abc Street", "England", "UK"],
27  };
28
29  const result = grantAccessToService(customerPayment);
30
31  console.log(result);
```

The above code, when run returns “Access not granted”.

This is because `isAuthorised` on line 23 is spelt with a “s” whereas `isAuthorized` on line 2 is with a “z”.

Typescript would pick this up automatically!

Typescript equivalent where we are defining the structure of the `CustomerPayment` object:

```
demo > payment > JS index.js >  grantAccessToService
1— type CustomerPayment = {
2—   customerId: number;
3—   isValid: boolean;
4—   isAuthorised: boolean;
5—   amount: number;
6—   paymentMethod: string;
7—   address: string[];
8— };
9—
10— export function grantAccessToService(payment: CustomerPayment) {
11   const shouldAllowAccessToService = payment.isValid && payment.
12
13   const GOLD_TIER = 10000;
14   const SILVER_TIER = 5000;
15   const BRONZE_TIER = 2000;
16
17   if (shouldAllowAccessToService) {
18     if (payment.amount > GOLD_TIER) {
```

Typescript showing the error with payment.isAuthorized does not exist:

```
demo > payment > JS index.js > grantAccessToService
1- Payment = {
2-   number;
3-   boolean;
4-   id: boolean;
5-   iber;
6-   od: string;
7-   ring[];
8-
9-
10- on grantAccessToService(payment: CustomerPayment) {
11-   .allowAccessToService = payment.isValid && payment.isAuthorized;
12- }
```

any  
Property 'isAu  
Did you mean '  
index.ts(4, 3)  
View Problem

Backend example:

```
JS app.js U X
demo > express > JS app.js > getBookById
1  import express from "express";
2
3  const app = express();
4
5  function getBookById(idToSearchFor) {
6    const books = [
7      { id: 1, title: "The Tired Janitor" },
8      { id: 2, title: "Earth to River" },
9    ];
10
11    return books.find((book) => book.id === idToSearchFor);
12  }
13
```



```

14  app.get("/books/:id", function (req, res) {
15      const bookId = req.params.id;
16      const book = getBookById(bookId);
17
18      res.json({
19          success: true,
20          book,
21      });
22  });
23
24  export default app;

```

In the above code, we have forgotten to change the `req.params.id` to a number on line 15. JS doesn't pick this up, but Typescript screenshot below shows this error being picked up on line 16, before we even run our code:

```

5  function getBookById(idToSearchFor: number) {
6      const books = [
7          { id: 1, title: "The Tired Janitor" },
8          { id: 2, title: "Earth to River" },
9      ];
10
11      return books.find((book) => book.id === idToSearchFor);
12  }
13
14  app.get("/books/:id", function (req, res) {
15      const bookId = req.params.id;
16      const book = getBookById(bookId);
17

```

TypeScript documentation recommends using implicit type inference where possible, only use explicit type annotation where needed



```
1
2 // Explicit annotation
3 let isExcited: boolean = true;
4
5 // I let isExcited1: boolean
6 let isExcited1 = true;
```

Explicit might be needed when you want to allow options for type using | operator:

```
6 let isExcited1: boolean | null = true;
```