Smooth operator

Let's look at a few of the more esoteric JavaScript operators, from the not not to the dot dot dot.
If anyone remembers the Sade tune of the same title, that's a bonus - keep it in your head or your earphones while reading.

Pictured in the preceding illustration, there are 10 types of operator which may seem a bit baffling on first encounter. They are:

1. `!`

The **NOT OPERATOR**.

2. `!!`

The **NOT NOT OPERATOR**.

3. `...`

The **REST OPERATOR**, and **SPREAD SYNTAX** - depending on where it is used.

4. `?.`

The **OPTIONAL CHAINING OPERATOR** - new in 2020.

5. `??`

The **NULLISH COALESCING OPERATOR**.

6. `?.??`

The **OPTIONAL CHAINING OPERATOR USED WITH THE NULLISH COALESCING** - new in 2020.

7. The `?` and `:` operators

Used together in a **TERNARY EXPRESSION**.

8. `++i`

The **PREFIX INCREMENT** - Also decrement, as `--i`. Where i is any number.

9 `i++`

The **POSTFIX INCREMENT** - Also decrement, as `i--`.

10 `+i`

The **UNARY PLUS** - Also unary negation - `-i`.

Look, if you are coming in to frontend from the creator side, bear with me.

These operators are grouped together here arbitrarily by me, and have no special relationships other than that. Sample output is shown in the Node console.

I very often kick off with some of these at the very start of my courses, just to change it up a little. If you can get into the JS mindset they make perfect sense.

1-2. the ! (not) and !! (not not) operators

```
console.log(Boolean(1));    // true
console.log(Boolean(0));    // false
console.log(!1);            // false
console.log(!0);            // true
console.log(!!1);           // true
console.log(!!0);           // false
```

3A. the rest operator, spread syntax

```
const Musician = function(name, age, ...instruments){   // rest parameters
    this.name = name;
    this.age = age;
    this.instruments = [...instruments];                // spread syntax
}
let alan = new Musician('Alan', 54, 'trombone', 'vox', 'keyboards');    // 3 args instruments
let ian = new Musician('Ian', 74, 'flute', 'vox');                      // 2 args instruments
let ariana = new Musician('Ariana', 28, 'vox');                         // 1 args instruments

Musician.prototype.plays = function(){
    let played = '';
    for(let i = 0; i < this.instruments.length; i++){
        if(i === this.instruments.length - 1){
            played += `${ this.instruments[i] }. `;
        } else {
            played += `${ this.instruments[i] }, `;
        }
    }
    return `${ this.name } plays ${ played }`;
}
console.log(alan.plays());
console.log(ian.plays());
console.log(ariana.plays());
```

1-3. output

```
TERMINAL    PROBLEMS    OUTPUT    DE
trainer@STAYAHEADSMBP2 frontend-cook
trainer@STAYAHEADSMBP2 JS-section %
true
true
false
false
true
true
false
Alan plays trombone, vox, keyboards.
Ian plays flute, vox.
Ariana plays vox.
```

1. `!`

The **NOT OPERATOR** is a unary logical negation. That is to say, when used in a boolean expression, it returns the opposite truth value of the operand to which it is applied. So

```
!truth === false
!false === true
!(3 > 4) === true
!('') === true
!('s') === false
```

Ok so the last two need a bit of explanation. Empty strings are "falsy" - they have a truth value of false, while non-empty strings are "truthy" - truth value of true. When grouped, with the **not** operator (actually you don't need the brackets here, they are just for

3B. handling nullish values an old school way - no default parameters with rest operator!

```
// what about controlling output for null/undefined input?
// ES6 default parameters, BUT...
// SyntaxError: Rest parameter may not have a default initializer
const MaybeAMusician = function(name = 'not supplied', age = 0, ...instruments){
    this.name = name;
    this.age = age;
    this.instruments = [...instruments];
}
let elon = new MaybeAMusician('Elon');
let grimes = new MaybeAMusician('Grimes', 33, 'vox', 'guitar');

MaybeAMusician.prototype.plays = function(){
    let played = '';
    if(!this.instruments[0]){   // empty array check
        played = '[instruments unknown]';
    } else {
        for(let i = 0; i < this.instruments.length; i++){
            if(i === this.instruments.length - 1){
                played += `${ this.instruments[i] }. `;
            } else {
                played += `${ this.instruments[i] }, `;
            }
        }
    }
    return `${ this.name } plays ${ played }`;
}
```

readability following on from the 3 > 4 example) they give their truth opposites. This segways into:

2. `!!` The **NOT NOT OPERATOR**.

The first not (`!`) coerces an operand into the negation of it's truth value. So its datatype, whatever it is originally, becomes Boolean. The second not (`!`) negates the negation but does not further coerce type from Boolean so you get the actual truth value of the initial value. Thus:

`!3 === false` // *non-zero numbers are truthy*

`!!3 === true` // *double-negated*

It is the operand equivalent of passing the value to a Boolean constructor function, thus:

4-6. handling nested nullish values - pre-ES11

```
console.log(elon.plays());
console.log(elon.name, elon.age, elon.instruments);
console.log(elon.instruments[0])

console.log(grimes.plays());
console.log(grimes.name, grimes.age, grimes.instruments);
console.log(grimes.instruments[0])
```

4-6. output

```
Elon plays [instruments unknown]
Elon 0 []
undefined
Grimes plays vox, guitar.
Grimes 33 [ 'vox', 'guitar' ]
vox
```

`Boolean(3)` // *true*

See code examples 1-2.

Examples 3-7 could be grouped under the term "handling nullish values". JavaScript, originally intended as a browser-only language, is built to fall down gracefully. For instance,

`1/0` // *Infinity*

If you want to be academic, this is not JS behaviour per se but follows on from its adoption of the IEEE 754 Standard for Floating-Point Arithmetic. This can cause problems with hunting down bugs, though, and very often we find something returns `null` or `undefined` because the data is missing or of the wrong type. ES6 Strict Mode, and later, TypeScript, are both good ways of mitigating some of the whackier parts, but both ES6, and later ES11 have given us some very useful little operators to tie up just this sort of loose end - a value requested is not there or has not yet been initialised into the correct type, and so returns `null` or `undefined`.

"Nullish" here means null or undefined as they are equivalent before type coercion but not afterwards:

`undefined == null` // *true*
`undefined === null` // *false*

These operators give us a graceful way out of trouble, of the `null` or `undefined` sort.

4-6. handling nested nullish values - pre-ES11

```
// add Musicians to Band
// iterate over Band with a filter() to check
// if instruments > 3 what is fourth instrument?
const band = [];
band.push(grimes);
band.push(elon);
// anonymously instantiated:
band.push(new MaybeAMusician('Bruce', 71, 'guitar', 'vox', 'keyboards', 'harmonica'));

console.table(band.filter(
    member => member.instruments[1]
));                                  // band members who play 2 or more instruments:
                                    // [ {GRIMES OBJECT}, {anonymous BRUCE OBJECT} ]
console.table(band.filter(
    member => member.instruments[3]
));                                  // band members who play 4 or more instruments:
                                    // [ {anonymous BRUCE OBJECT} ]

// care!
console.log(!![]); // empty arrays are true
console.log(!!''); // empty strings are false
```

3. . . .

The **REST OPERATOR**, and **SPREAD SYNTAX**.

This operator handles one or many arguments in a function call. It is used in function declaration, and must be the last argument defined in function parameters. See example 3A above.

The alternative would be to specify a string - "instruments" which would list all the intruments played. If we wanted to iterate around the array of instruments however, we would need to create an iterable object using messy string splitting methods, as strings are iterable but only character by character.

So our instruments array may have none, one, or many elements, and the number doesn't need to be specified imperatively in source code. But what of the case where there are no instruments?

Well we can always hack it by write our own validation code as in example 3B above, where if it's not the case there is an element at index position zero, we assign a '[instruments unknown]' message.

It is worth noting that if we attempt to assign a default parameter value to instruments as we have done

4-7. output

```
┌─────────┬──────────┬─────┬──────────────────────────────────────────────┐
│ (index) │   name   │ age │                 instruments                  │
├─────────┼──────────┼─────┼──────────────────────────────────────────────┤
│    0    │ 'Grimes' │ 33  │                [ 'vox', 'guitar' ]           │
│    1    │ 'Bruce'  │ 71  │ [ 'guitar', 'vox', 'keyboards', ... 1 more item ] │
└─────────┴──────────┴─────┴──────────────────────────────────────────────┘
┌─────────┬──────────┬─────┬──────────────────────────────────────────────┐
│ (index) │   name   │ age │                 instruments                  │
├─────────┼──────────┼─────┼──────────────────────────────────────────────┤
│    0    │ 'Bruce'  │ 71  │ [ 'guitar', 'vox', 'keyboards', ... 1 more item ] │
└─────────┴──────────┴─────┴──────────────────────────────────────────────┘
true
false
main instrument: no instruments found
main instrument: no instruments found
main instrument: no instruments found
main instrument: vox
undefined
Alan's third instrument is keyboards
Grimes doesn't have a third instrument
trainer@STAYAHEADSMBP2 JS-section %
```

for name, it throws an error, and cannot be default-initialised in this way. Far better alternatives are with us now since ES11/ECMAScript 2020:

4. `?.`

The **OPTIONAL CHAINING OPERATOR** - new in 2020. What did we do before this? Well one option was to use a functional array method to return a filtered version of the data, see example 4-6 above.

Another, covered in the MDN docs, was to use a short-circuit logical operator to only evaluate the right hand side of an expression when the left side returned true.

The optional chaining operator results in shorter and simpler expressions when accessing chained properties, like those deep within an object, when one or more props in the chain may be missing. It also works with nested function calls too, so it's very useful when working with APIs to bulletproof your code.

In example 4-7 below we focus on a particular case: used together with the nullish coalescing operator.

5. `??`

The **NULLISH COALESCING OPERATOR** allows us to swap in a value for where something returns null or undefined.

This solves the problem we had earlier on in our examples here where our MaybeAMusician object plays no instruments, and the `...` Rest operator cannot be initialised with a default value. It ought to be emphasised there were always ways to do this, but using nullish coalescing in conjunction with optional chaining

4-7. using ES11 optional chaining and nullish coalescing operators, and ternary expressions

```
// with ES11 optional chaining and nullish coalescing:
console.log(`main instrument: ${ elon?.foobar?.[0]?? 'no instruments found' }`);        // no instruments found
// in this case I wouldn't use a template literal – readability probably better the old way below (pre-ES6)
// just group the expression otherwise the RH side will return undefined:
console.log('main instrument: ' + (elon?.instruments?.[0]?? 'no instruments found'));   // no instruments found
console.log('main instrument: ' + (grimes?.barbaz?.[0]?? 'no instruments found'));      // no instruments found
console.log('main instrument: ' + (grimes?.instruments?.[0]?? 'no instruments found')); // vox

////////////////////////////////// pre-ES11: ///////////////////////////////////
// console.log(elon.foobar[0])      // TypeError: Cannot read property '0' of undefined
console.log(elon.instruments[0])    // undefined
//////////////////////////////////// (pre-Es11)/////////////////////////////////

console.log(!!alan.instruments[2]   ? `${ alan.name }'s third instrument is ${ alan.instruments[2] }`
|   |   |   |   |   |   |   |   |   : `${ alan.name } doesn't have a third instrument`);
console.log(!!grimes.instruments[2] ? `${ grimes.name }'s third instrument is ${ grimes.instruments[2] }`
|   |   |   |   |   |   |   |   |   : `${ grimes.name } doesn't have a third instrument`);
|
```

is a strong way of keeping it readable (once you know!) - and with just one line of code.

6. `?.??`

The **OPTIONAL CHAINING OPERATOR USED WITH THE NULLISH COALESCING** - new in 2020.

In example 4-7 above, the expression using optional chaining **AND** nullish coalescing returns *'no instruments found'* **BOTH** when there is a non-existent prop in the chain **AND** when the end result is `undefined`.

Where the props are all correctly named and there is an element in the first position of the instruments array it returns that element - in this case *Grimes' main instrument: vox*. Neat.

7. The `?` and `:` operators

Used together in a **TERNARY EXPRESSION**.

Ternary expressions have been around a while and again in example 4-7 above we have an equivalent to using the nullish coalescing together with optional chaining.

To my mind, however, the ternary expression is more cumbersome to read, because it must fit all of its output code in its internals, rather than simply be called as input to another function, even just a `console.log` statement.

So we can check for the truthiness (`!!`) of an element in the instruments array, and return the case if true (following the single question mark), or (**EXCLUSIVE OR**) the case if false, following the colon. No case in which both are true is allowed.

4-7. output

(index)	name	age	instruments
0	'Grimes'	33	['vox', 'guitar']
1	'Bruce'	71	['guitar', 'vox', 'keyboards', ... 1 more item]

(index)	name	age	instruments
0	'Bruce'	71	['guitar', 'vox', 'keyboards', ... 1 more item]

```
true
false
main instrument: no instruments found
main instrument: no instruments found
main instrument: no instruments found
main instrument: vox
undefined
Alan's third instrument is keyboards
Grimes doesn't have a third instrument
trainer@STAYAHEADSMBP2 JS-section %
```

8 - 9. `++i / i++ / --i / i--`

The **PREFIX AND POSTFIX INCREMENT/DECREMENT** Where `i` is any number. These are well-known gotchas to anyone who has done a bit of JavaScript, as together with postfix, they are two slightly different flavours of the same thing. Neither of them should be used in React owing to the way they update state directly, by-passing the internal `React.setState()`.

At first sight it appears they do the same thing, that is, increment/decrement by one. But, crucially, the timing of the update is different. Prefix makes the update before it is used, and postfix makes it after. The value of the variable is the same updated value for both, just different at the time of use:

```
let i = 0;
let j = 0;
i = i + 1; // 1    j= j+ 1;  // 1
i += 1;  // 2      j += 1;  // 2
++i // 3 - different   j++ // 2 - different
i;  // 3 - same after   j;  // 3 - same after
```

10 `+i`

The **UNARY PLUS** - Also unary negation - `-i`.

Put a minus in front of a number and it turns it into a negative number, a plus likewise. But they also coerce strings into number types:

```
let i = '42';   typeof i // 'string'
i = +i // 42    typeof i // 'number'
let j = '-42';  typeof j // 'string'
j = -j // 42    typeof j // 'number'
```