JavaScript Foundations

All knowledge (here) was passed down from Kyle Simpson

Types & Coercion

There is debate over whether `coercion` is a useful feature or a flaw in the design of the language... `coercion` is often said to be magical, evil, confusing, and just downright a bad idea...

But instead of complaining we should understand it fully, flaws and all...

Goal is to fully explore the pros and cons of coercion, so you know what you're getting into if you decide to use it

Converting Values

Converting a value from one type to another is often called 'type casting', when done **explicitly**(forthright doing it) When done **implicitly**(done behind the scenes), it is called `coercion`

So we have **implicit coercion** vs **explicit coercion**

The difference should be obvious:

explicit coercion is when it is obvious from looking at the code that a type conversion is intentionally occurring

whereas,

implicit coercion is when the type conversion will occur as a less obvious side effect of some other intentional operation

```
let a = 42;

let b = a + ""; // implicit coercion

let c = String(a); // explicit coercion
```

For `b`, the coercion that occurs happens implicitly, because the `+` operator combined with one of the operands being a `string` value ("") will insist on the operation being a `string` concatenation (adding 2 strings together), which *as a hidden side effect* will force the 42 value in a to be coerced to its string equivalent: "42"

By contrast, the `String(..)` function makes it pretty obvious that it's explicitly tking the value in `a` and coercing it to a string representation

Both approaches accomplish the same effect: "42" comes from 42 But it's the *how* that is at heart of the heated deabtes over JavaScript's coercion

Before we can explore explicit versus implicit coercion, we need to learn the basic rules that govern how values become either a string,number, or boolean

We will specifically pay attention to ToString, ToNumber, and ToBoolean

ToString

When any non-string value is coerced to a string representation, the conversion is handled by the `ToString` abstract operation

Built-in primitive values have natural stringification: null becomes "null", undefined becomes "undefined", and true becomes "true" numbers are generally expressed in the natural way you'd expect

Arrays have an overridden default `toString()` that stringifies as the (string) concatenation of all its values (each stringified themselves), with "," in between each value:

```
let a = [1,2,3];
a.toString(); // "1,2,3"
```

Again, toString() can either be called explicitly, or it will automatically be called if a non-string is used in a string context

ToNumber

If any non-number value is used in a way that requires it to be a number, such as a mathematical operation, the ES5 spec defines the `ToNumber` abstract operation

For example, `true` becomes `1` and `false` becomes `0` `undefined` becomes `NaN`,but (curiously) `null` becomes `0`

`ToNumber` for a `string` value essentially works for the most part like the rules/syntax for numeric literals

If it fails, the result is `NaN`

```
var a = {
 valueOf: function(){
 return "42";
 }
};

var b = {
 toString: function(){
 return "42";
 }
};

var c = [4,2];
```

```
c.toString = function(){
 return this.join( "" ); // "42"
};

Number( a ); // 42
Number( b ); // 42
Number( c ); // 42
Number( "" ); // 0
Number( [] ); // 0
Number( [ "abc" ] ); // NaN
```

ToBoolean

JS has actual keywords `true` and `false`,and they behave exactly as you'd expect `boolean` values

It's a common misconception that the values `1` and `0` are identical to `true`/`false` While that may be true in othre languages, in JS the `numbers` are `numbers` and the `booleans` are `booleans` You can coerce `1` to `true` (and vice versa) or `0` to `false` (and vice versa). **But they are not the same**

But that's not the end of the story, we need to discuss how values other than the two `booleans` behave whenever you coerce *to* thier `boolean` equivalent

All of JavaScript's values can be divided into two categories:

1. Values that will become `false` if coerced to `boolean`
2. Everything else (which will obviously become `true`)

The following will output `false` if force a `boolean` coercion on it

- `undefined`
- `null`
- `false`
- `+0, -0, and NaN`
- `""`

Explicit Coercion

Explicit coercion refers to type conversions that are obvious and explicit. There’s a wide range of type conversion usage that clearly falls under the explicit coercion category for most developers

We’ll start with the simplest and perhaps most common coercion operation: coercing values between string and number representation.

```
var a = 42;

var b = String( a );

var c = "3.14";

var d = Number( c );
```

```
b; // "42"
d; // 3.14
```

`String(..)` coerces from any other value to a primitive string value, using the rules of the `ToString` operation discussed earlier. `Number(..)` coerces from any other value to a primitive number value, using the rules of the `ToNumber` operation discussed earlier. I call this explicit coercion because in general, it's pretty obvious to most developers that the end result of these operations is the applicable type conversion

Now, let's examine coercing from any non-boolean value to a boolean. Just like with `String(..)` and `Number(..)` above, `Boolean(..)` (without the new, of course!) is an explicit way of forcing the `ToBoolean` coercion:

```
let a = "0";
let b = [];
let c = {};
let d = "";
let e = 0;
let f = null;
let g;

Boolean( a ); // true
Boolean( b ); // true
Boolean( c ); // true
Boolean( d ); // false
Boolean( e ); // false
Boolean( f ); // false
Boolean( g ); // false
```

Implicit Coercion

Implicit coercion refers to type conversions that are hidden, with nonobvious side effects that implicitly occur from other actions.

In other words, implicit coercions are any type conversions that aren't obvious (to you).

While it's clear what the goal of explicit coercion is (making code explicit and more understandable), it might be too obvious that implicit coercion has the opposite goal: making code harder to understand.

```
let a = "42";
let b = "0";

let c = 42;
let d = 0;

a + b; // "420"
c + d; // 42
```

Operators `||` and `&&`

It's quite likely that you have seen the `||` (logical or) and `&&` (logical and) operators

Logical operators aren't completely correct, they could be called "selector operators" or "operand selector operators" Why? Because they don't actually result in a `logic` value in JavaScript, as they do in some other languages

So what do they result in? They reuslt in the value of one of their two operands In other words, they select one of the two operand's value

Quoting the ES5 spec

The value produced by a `&&` or `||` operator is not necessarily of type boolean They value produced will always be the value of one of the two operand expressions

Let's illustrate:

```
let a = 42;
let b = "abc";
let c = null;

a || b; ///42
a && b; //"abc"

c || b; //"abc"
c && b; //null
```

How, does this make sense?? In languages like C and PHP, those expressions result in `true` or `false`, but in JS the result comes from the values themselves

Both `||` and `&&` operators perform a `boolean` test on the *first operand (a or c)* If the operand is not already *boolean*, a normal `ToBoolean` coercion occurs, so that the test can be performed

For the `||` operator, if the test is true, the `||` expression results in the value of the *first operand* (a or c) If the test is `false`, the `||` expression results in the value of the *second operand* (b)

Inversely, for the `&&`, if the test is `true`, the `&&` expression results in the value of the *second operand* (b) If the test is `false`, the `&&` expression results in the value of the *first operand* (a or c)

The result of a `||` or `&&` expression is always the underlying value of one of the operands, *not* the result of the test In `c && b`, c is `null`, and thus false. But the `&&` expression itself results in the `null`, not in the coerced `false` used in the test

Do you see how these operators act as "operand selectors" now? Another way of thinking about these operators:

```
a || b;
// roughly equivalent to:
a ? a : b;
```

```
a && b;
// roughly equivalent to:
a ? b : a;
```

An extremely common and helpful usage of this behavior, which there's a good chance you may have used before and not full understood,is:

```
function foo(a,b) {
  a = a || "hello";
  b = b || "world";

  console.log(a + "" + b);
}

foo(); // "hello world"
foo("yeah", "yeah!"); // "yeah yeah!"
```

Loose Equals Versus Strict Equals

Loose equals is the `==` operator Strict equals is the `===` operator

Both operators are used for comparing two values for "equality", but the "loose" versus "strict" indicates a very important difference in behavior between the two, **specifically** in how they decide "equality"

A very common misconception about the two operators is: `==` checks values for equality and `===` checks both values **and** types for equality

But this is incorrect...

`===` allows coercion in the equality comparison `===` disallows coercion in the equality comparison

Equality Performance

In the first explanation, it seems obvious that `===` is doing more work than `==` because it has to *also* check the type

In the second explanation, `==` is the one *doing more work* because it has to follow through the steps of coercion if the types are different

Don't fall into the trap of thinking this has anything to do with performance, though, as if `==` is going to be slower than `==` in any relevant way... While it's measurable that coercion does take a *little bit* of processing time, it's mere microseconds...

If you're comparing two values of the same types, `==` and `===` use the identical algorithm, and so other than minor differences in engine implementation, they should do the same work...

If you're comparing two values of different types, the performance isn't the important factor What you **should** be asking yourself is, when comparing these two values, do I want coercion or not?

If you want coercion, use `==` loose equality, **but** if you don't want coercion, use `===` strict equality

You **must** remember that

- `NaN` is **never** equal to itself
- `+0` and `-0` **are equal** to each other

Comparing: strings to numbers:

```
let a = 42;
let b = "42";

a === b; // false
a ==b; // true
```

As we'd expect, a == b fails, because no cercion is allowed, and indeed the 42 and "42" values are different

Comparing: anything to boolean

One of the biggest gotchas with the *implicit* coercion of `==` loose equality pops up when you try to compare a value directly to `true` or `false`

Consider:

```
let a = "42";
let b = true;

a == b; // false
```

But we know that "42" is a truthy value. So, how come it's not `==` loose equal to `true`?

The reason is both simple and deceptively tricky, the specs say:

1. If `Type(x)` is Boolean, return the result of the comparison `ToNumber(x) == y`
2. If `Type(y)` is Boolean, return the result of the comparison `x == ToNumber(y)`

Let's break that down:

```
let x = true;
let y = "42";

x == y; //false
```

The `Type(x)` is indeeed `Boolean`, so it performs `ToNumber(x)`, which coerces `true` to `1`... Now, `1` == "42" is evaluated

The types are still different, so we reconsult the algorithm, which just as above will coerce "42" to 42, and 1 == 42 is clearly `false`

Reverse it, and we still get the same outcome:

```
let x = "42";
let y = false;

x == y; //false
```

The `Type(y)` is `Boolean` this time, so `ToNumber(y)` yields 0 "42" == 0, recursively becomes 42 == 0, whcih is of course `false`

In other words, the value "42" is neither `== true` nor `== false`

Review of Types & Coercion

JavaScript type conversions is called `coercion`, which can be characterized as either *explicit* or *implicit*

Coercion gets a bad rap, but it's actually quite useful in many cases An important task for the responsible JS developer is to take the time to learn all the ins and outs of coercion to decide which parts will help improve their code, and which parts they really should avoid

Explicit coercions is code where it is obvious that the intent is to convert a valuefrom one type to another The benefit is improvement in readability and maintainability of code by reducing confusion

Implicit coercion is coercion that is "hidden" as a side effect of some other operation, where it's not as obvious that the type conversion will occur. While it may seem that *implicit* coercion is the opposite of *explicit* and is thus bad, **actually** *implicit* coercion is also about improving the readability of code

Especially for the *implicit* type, coercion must be used responsibly and consciously. Know why you're writing the code you're writing, and how it works Strive to write code that others will easily be able to leran from and understand as well

Grammar

Statements & Expressions

Most common for developers to assume that the term "statement" and "expression" are roughly equivalent... But here we need to distinguish between two, because there are some very important differences in our JS programs

In JavaScript "grammar", statements are sentences, expressions are phrases, and operators are conjunctions/punctuation

Every expression in JS can be evaluated down to a single, specific value result

```
let a = 3 * 6;
let b = a;
```

```
b;
```

In this snippet, `3 * 6` is an expression But `a` on the second line is also an expression, as is `b` on the third line. The `a` and `b` expressions both evaluate to the values stored in those variables at that moment, which also happens to be 18

Moreover, each of the three lines is a statement containing expressions

```
let a = 3 * 6
b = a;
b;
```

These are assignment expressions. The third line contains just the expression `b`, but it's also a statement all by itself

Statement Completion Values

It's a fairly little known fact that statements all have completion values **(even if that value is just undefined)**

How would you even go about seeing the completion value of a statement?

The most obvious answer is to type the statement into your browser's developer console, because when you execute it, the console by default reports the completion value of the recent statement it executed

Let's consider `let b = a`. What's the completion value of that statement?

The `b = a` assignment expression results in the value that was assigned, but the `var` statement itself results in undefined Why? Because `var` statements are defined that way in the spec. If you put `var a = 42`; into your console, you'll see `undefined` reported back instead of 42

Expression Side Effects

Most expressions don't have side effects, for example:

```
let a = 2;
let b = a+3;
```

The expression `a + 3` did not itself have a side effect, like for instance changing `a` It had a result, which is `5`, and that result was assigned to `b` in the statement `b = a + 3`

The most common example of an expression with (possible) side effects is: **a function call expression**

```
function foo() {
  a = a + 1;
}

let a = 1;

foo(); // result: `undefined`, side effect: changed `a`

// There are other side-effecting expressions, though:

let a = 42;
let b = a++;

// The expression a++ has two separate behaviors
// First, it retunrs the current value of a, which is 42 (which then gets assigned to b)
// Next, it changes the value of a itself, incrementing it by one

let a = 42;
let b = a++;

a; // 43
b; // 42
```

Many developers would **mistakenly believe that b has value 43** just like a does But the confusion comes from not fully considering the *when* of the side effects of the `++` operator

The `++` increment operator and the `--` decrement operator are both unary operators, which can be used in either a **post-fix (after)** or **pre-fix(before)** position

```
let a = 42;

a++; // 42
a; //43

++a; //44
a; // 44
```

When `++` is used in the prefix position as `++a`, its side effect (incrementing a) happens *before* the value is returned from the expression, rather than *after* as with `a++`

It is sometimes mistakenly thought that you can encapsulate the *after* side effect of `a++` by wrapping it in a `( )` pair, like:

```
let a = 42;
let b = (a++);

a; // 43
b; // 42
```

Unfortunately, `( )` itself doesn't define a new wrapped expression that would be evaluated *after* the *after side effect* of the `a++` expression

In fact, even if it did, `a++` returns 42 first, and unless you have another expression that reevaluates `a` after the side effect of `++`, you're not going to get `43` from that expression, so `b` will not be assigned 43.

There's an option, though the `,` statement-series comma operator This opeartor allows you to string together multiple standalone expression statements into a single statement:

```
let a = 42, b;
b = ( a++, a );

a; // 43
b; //43
```

Contextual Rules

curly braces:

There's two main places that a pair of curl braces `{ .. }` will show up in your code...

1. `Object literals`

First, as an `object literal`

// assume there's a `bar()` function:

```
var a = {
  foo: bar();
}
```

How do we know this is an object literal?

Because the `{..}` pair is a value that's getting assigned to `a`

The `a` reference is called an "l-value" (aka left-hand value) since it's the target of an assignment The `{..}` pair is an 'r-value' (aka right-hand value) since it's used *just* as a value (in this case as the source of an assignment)

labels:

What happens if we remove the `var a =` part of the above snippet?

```
// assume there's a `bar()` function defined
{
  foo: bar()
}
```

A lot of developer assume that the `{..}` pair is just a standalone `object` literal that doesn't get assigned anywhere But it's actually entirely different

Here, `{..}` is just a regular code block. It's not very idiomatic in JS to have a standalone `{..}` block like that, but it's perfectly valid JS grammar

It can be especially helpful when combined with `let` block-scoping declarations

The `{..}` code blocl here is functionally pretty much identical to the code block being attached to some statement, like a `for/while` loop, `if` conditional But if it's a normal block of code, what's that bizarre looking `foo: bar()` syntax, and how is that legal?

Object destructuring

Another place you'll see `{..}` pairs showing up is with "destucturing assignments" specficially `object` destructuring. Consider:

```
function getData() {
  return {
    a: 42,
    b: "foo"
  }
}

var { a, b } = getData();

console.log(a,b);
```

As you can tell, `var{a,b}=` is a from of ES6 destructuring assignment, which is roughly equivalent to:

```
var res = getData();
var a = res.a;
var b = res.b;
```

Object destructuring with a `{..}` pair can also be used for named function arguments, which is sugar for this same sort of implicit object property assignment:

```
function foo({a,b,c}) {
  // no need for:
  // var a = obj.a, b = obj.b, c = obj.c
  console.log(a,b,c)
}

foo( {
  c: [1,2,3],
  a: 42,
  b: "foo"
}); // 42 "foo" [1,2,3]
```

So, the context we use `{..}` pairs in, entirely determines what they mean...

Operator Precedence

JavaScript's version of `&&` and `||` are interesting in that they select and return one of their operands, rather than just resulting `true` and `false` That's easy to reason about if there are only two operands and one operator:

```
let a = 42;
let b = "foo";

a && b; // "foo"
a || b; // 42
```

But what about when there's two operators involved, and three operands?

```
let a = 42;
let b = "foo";
let c = [1,2,3];

a && b || c // ???
a || b && c; // ???

// to better understand operator precedence let's examine this further

var a = 42, b;
b = ( a++, a );
a; // 43
b; // 43

// but what would happen if we remove the `()`?

var a = 42, b;
b = a++, a;
a; // 43
b; // 42

// why did that change the value assigned to b?
```

Because, the `,` operator has a loser precedence than the `=` operator. So `b=a++`, `a` is interpreted as `(b = a++)`, `a`. Because `a++` has 'after side effects', the assigned value to b is the value 42 before the `++` changes `a`

Tighter Binding

Does the `? :` operator have more or less precedence than the `&&` and `||` operators?

`a && b || c ? c || b ? a : c && b : a`

Is that more like this?

`a && b || (c ? c || (b ? a : c) && b : a)`

Or more like this?

`(a && b || c) ? (c || b) ? a : (c && b) : a`

The answer is the second one. But why?

Because `&&` is more precedent than `||`, and `||` is more precedent than `? :`

So, the expression `(a && b || c)` is evaluated *first* before the `? :` it participates in

Another way this is commonly explained is that `&&` and `||` "bind more tightly" than `? :`

`try...finally`

You're familiar with `try...catch`, but `try` only requires either `catch` or `finally`, though both can be present if needed?

The code in the `finally` clause *always* runs (no matter what), and it always runs right after the `try` (and `catch` if present) finish, before any other code runs. In one sense, you can kind of think of the code in a `finally` clause as being a callback function that will always be called regardless of how the rest of the block behaves

So what happens if there's a `return` statement inside a `try` clause? It obviously will return a value, right? But does the calling code that receieves that value before or after the `finally`?

```
function foo() {
  try {
    return 42;
  }
  finally {
    console.log("hello");
  }
  console.log("never runs");
}

console.log(foo());
// Hello
// 42
```

The return 42 runs right away, which sets up the completion value from the `foo()` call This action completes the `try` clause and the `finally` clause immediately runs next. Only then is the `foo()` function complete, so that its completion value is returned back for the `console.log(..)` statement to use

Review of JavaScript Grammer

JavaScript grammar has plenty of nuance that we as developers should spend a little more time playing closer attention to than we typically do A little bit of effort goes a long way to solidifying your deeper knowledge of the language

Statements and expression have analogs in English language- statements are like sentences expressions are like phrases Expressions can be pure/self-contained, or they can have side effects

The JavaScript grammar layers semantic usage rules (aka context) on top of the pures syntax For example, `{ }` pairs used in various values in your program can mean statement blocks, object literals, destructuring assignments, or named function arguments

JavaScript operators all have well-defined rules for precedence (which ones bind first before others) and associativity Once you learn these rules, it's up to you to decide if precedence'associativity are *too implicit* for their own good, or if they will aid in writing shorter, clearer code...

JavaScript has several types of errors, but it's less known that it has two classification for errors: "early" (compiler throw, uncatchable) and "runtime" (try...catchable)

All syntax errors are obviously early errors that stop the program before it runs, but there are others, too

Function arguments have an interesting relationship to their formal declared narmed parameters Specifically, the `arguments` array has a number of 'gotchas' of leaky abstraction behavior if you're not careful Avoid `arguments` if you can, but if you must use it, by all means avoid using the positional slot in `arguments` at the same time as using a named parameter for that same argument

The finally clause attached to a `try` (or `try...catch`) offers some very interesting quirks in terms of execution processing order Some of these quirks can be helpful, but it's possible to create lots of confusion, especially if combined with labeled blocks As always, use `finally` to make code better and clearer, not more clever or confusing

The `switch` offers some nice shorthand for `if..else..if..` statements, but beware of many common simplifying assumptions about its behavior There are seceral quirks that can trip you up if you're not careful, but there's also some neat hidden tricks that `switch` has up its sleeve!