JavaScript Advanced

Types and Coercion

Overview

1. Types
2. Values
3. Natives
4. Coercion
5. Grammar

1. Built-in Types

- JavaScript defines seven built-in types:
  1. null
  2. undefined
  3. boolean
  4. number
  5. string
  6. object
  7. symbol -- added in ES6!
- **Note:** All of these types except **object** are called "**primitives**".

1. Built-in Types

- The ***typeof*** operator inspects the type of the given value, and always returns one of seven string values

```
typeof undefined     === "undefined"; // true
typeof true          === "boolean";   // true
typeof 42            === "number";    // true
typeof "42"          === "string";    // true
typeof { life: 42 }  === "object";    // true

// added in ES6!
typeof Symbol()      === "symbol";    // true
```

1. Built-in Types

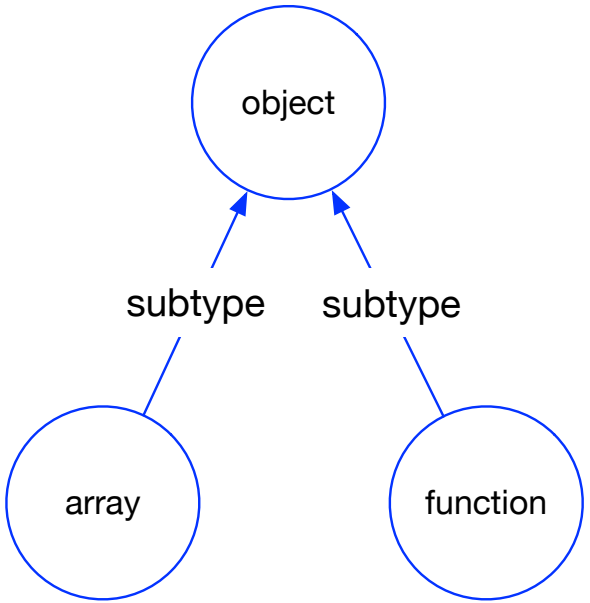
- Seventh string value that **typeof** return:

```
typeof function a(){ /* .. */ } === "function"; // true
```

1. Built-in Types

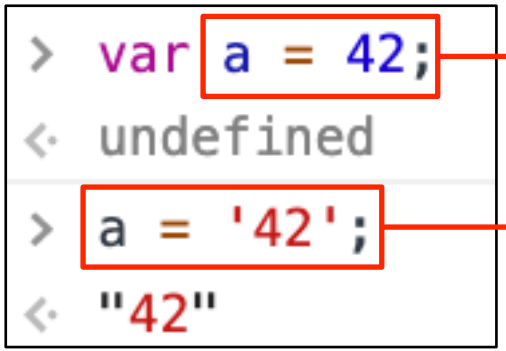
- What about arrays? They're native to JS, so are they a special type?

```
typeof [1,2,3] === "object"; // true
```



1. Values as Types

- In JavaScript, variables don't have types -- **values have types**. Variables can hold any value.
- Another way to think about JS types is that JS doesn't have "type enforcement,"
- A variable can, in one assignment statement, hold a string, and in the next hold a number, and so on



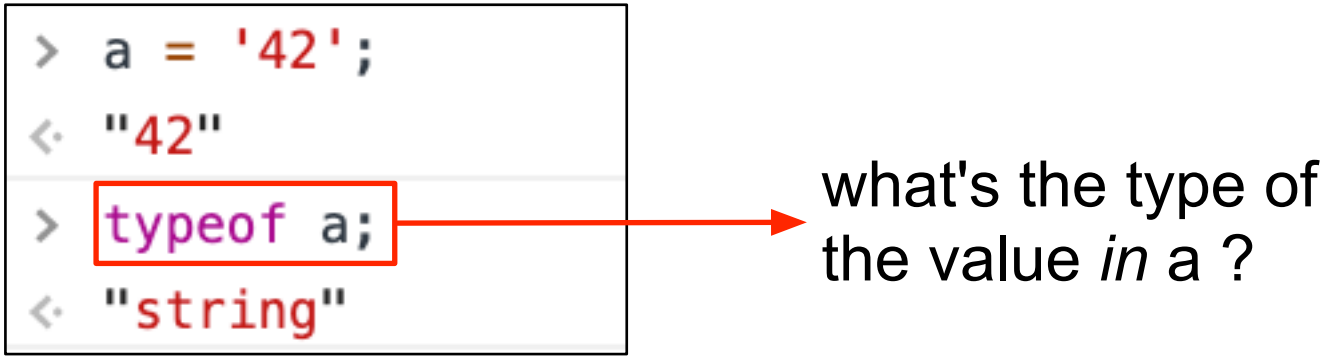
Store a number

Store a string

We already know that number and string are different type (behavior)

1. Values as Types

- If you use **typeof** against a **variable**, it's not asking "what's the type of the variable?" as it may seem, since JS variables have no types.
- Instead, it's asking "what's the type of the value *in* the variable?"



1. undefined vs undeclared

- Variables that have no value *currently*, actually have the **undefined** value.
- Calling typeof against such variables will return "**undefined**"

```
var a;

typeof a; // "undefined"

var b = 42;
var c;

// later
b = c;

typeof b; // "undefined"
typeof c; // "undefined"
```

1. undefined vs undeclared

`undefined` ≠ `undeclared`

declared but *at the moment* has no value in it

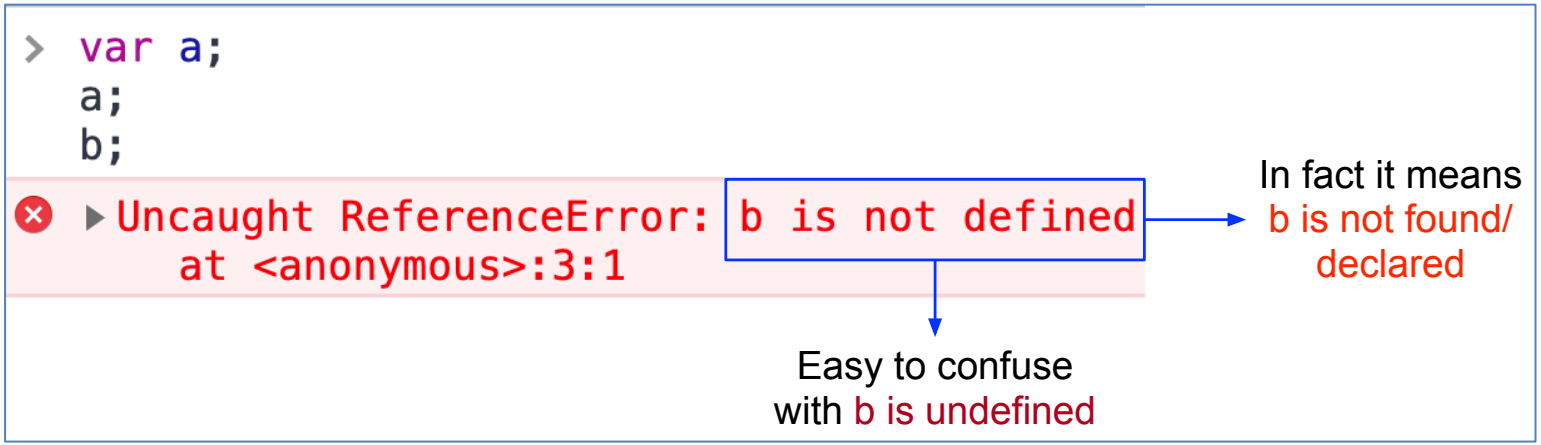
has not been formally declared

1. undefined vs undeclared

- Consider:

```
> var a;
  a;
  b;
✕ ▶Uncaught ReferenceError: b is not defined
      at <anonymous>:3:1
```

1. undefined vs undeclared



1. Types – Summary

- JavaScript has 7 built-in *types*: *null, undefined, boolean, number, string, object, symbol*.
- They can be identified by the **typeof** operator.
- Variables don't have types, but the values in them do. These types define intrinsic **behavior** of the values.
- **undefined** is a value that a declared variable can hold. ”**undeclared**" means a variable has never been declared.

2. Values Overview

- **arrays**, **strings**, and **numbers** are the most basic building-blocks of any program
- JavaScript has some **unique characteristics** with these types
- Understand those **characteristics** help you to build better program

2. Arrays

- As compared to other type-enforced languages, JavaScript **arrays** are just **containers** for any type of value, from string to number to object to even another array:

```
var a = [ 1, "2", [3] ];

a.length;                // 3
a[0] === 1;              // true
a[2][0] === 3;  // true
```

2. Arrays

- **arrays** are numerically indexed (as you'd expect), but they also are **objects** that can have string keys/properties added to them (but which don't count toward the length of the array):

```
var a = [ ];

a[0] = 1;
a["foobar"] = 2;

a.length;               // 1
a["foobar"];    // 2
a.foobar;               // 2
```

2. Arrays

- Be aware of is that if a **string** value intended as a key can be converted to a standard base-10 number, then it is assumed that you wanted to use it as a **number index** rather than as a string key!

```
var a = [1, 2, 3, 4];
a['1'] = 5; // similar to bracket notation
console.log(a);
```

```
▶ (4) [1, 5, 3, 4]
```

2. Strings

- It's a very **common belief** that strings are essentially just arrays of characters.
- While the implementation under the covers may or may not use arrays, it's important to realize that JavaScript strings are really **not the same as arrays of characters**.

2. Strings

- For example, let's consider these two values:

```
var a = "foo";
var b = ["f","o","o"];
```

2. Strings

- **Strings** do have a shallow resemblance to arrays -- array-likes, as above -- for instance, both of them having a **length** property, an **indexOf(..)** method and a **concat(..)** method:

```
a.length;                                                   // 3
b.length;                                                   // 3

a.indexOf( "o" );                                       // 1
b.indexOf( "o" );                                       // 1

var c = a.concat( "bar" );                          // "foobar"
var d = b.concat( ["b","a","r"] );      // ["f","o","o","b","a","r"]

a === c;                                                    // false
b === d;                                                    // false

a;                                                                  // "foo"
b;                                                                  // ["f","o","o"]
```

2. Strings

- So, they're both basically just "**arrays of characters**", right?
- **Not exactly**:

```
a[1] = "0";
b[1] = "0";

a; // "foo"
b; // ["f","0","o"]
```

2. Strings

- JavaScript strings are **immutable**, while arrays are quite **mutable**.
- Moreover, the **a[1]** character position access form was **not** always widely valid JavaScript.
- Older versions of IE did not allow that syntax (but now they do).
- Instead, the *correct* approach has been **a.charAt(1)**.

2. Strings

- None of the string methods that alter its contents can modify in-place, but rather must create and return new strings.
- By contrast, many of the methods that change array contents actually ***do*** modify in-place.

```
c = a.toUpperCase();
a === c;        // false
a;                      // "foo"
c;                      // "FOO"

b.push( "!" );
b;                      // ["f","O","o","!"]
```

2. Strings

- Let's take another example: reversing a string (incidentally, a common JavaScript interview trivia question!).
- arrays have a reverse() in-place mutator method, but strings do not:

```
a.reverse;                  // undefined

b.reverse();     // ["!","o","O","f"]
b;                                  // ["!","o","O","f"]
```

2. Strings

- **Solution:** convert the string into an array, perform the desired operation, then convert it back to a string.

```
var c = a
        // split `a` into an array of characters
        .split( "" )
        // reverse the array of characters
        .reverse()
        // join the array of characters back to a string
        .join( "" );

c; // "oof"
```

2. Strings

- When you need to modify **strings**:
  1. Store them as arrays rather than as strings.
  2. Change element in the arrays
  3. Then call join("") on the array *of characters* whenever you actually need the string representation.

```
var s = 'Hello Fresher';
var chars = s.split('');

chars[0] = 'B';
chars[1] = 'y';
chars[2] = 'b';
chars[3] = 'y';
chars[4] = 'e';

chars.join('');
"Bybye Fresher"
```

2. Numbers

- JavaScript has just one numeric type: **number**.
- This type includes both **integer** values and **fractional decimal** numbers.
- JavaScript specifically uses the "double precision" format (aka "64-bit binary") of the standard
- But in fact only 53-bit is used

2. Numberic Syntax

- Because number values can be boxed with the **Number** object wrapper, number values can access methods that are built into the Number.prototype.
- For example, the **toFixed**(..) method allows you to specify how many fractional decimal places you'd like the value to be represented with:

```
var a = 42.59;

a.toFixed( 0 ); // "43"
a.toFixed( 1 ); // "42.6"
a.toFixed( 2 ); // "42.59"
a.toFixed( 3 ); // "42.590"
a.toFixed( 4 ); // "42.5900"
```

2. Small Decimal Values

- The most **(in)famous** side effect of using binary floating-point numbers (which, remember, is true of **all** languages that use IEEE 754 -- not *just* JavaScript as many assume/pretend) is:

```
0.1 + 0.2 === 0.3; // false
```

2. Small Decimal Values

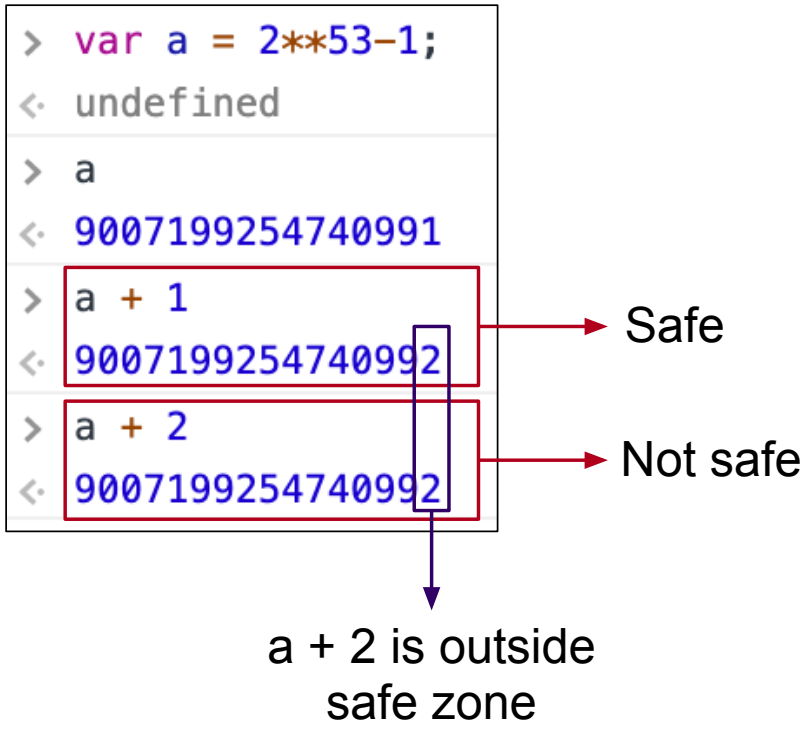
- **Mathematically**, we know that statement should be **true**.
- Why is it **false**?
  - Simply put, the representations for 0.1 and 0.2 in binary floating-point are not exact,
  - So when they are added, the result is not exactly 0.3.
  - It's **really** close: 0.30000000000000004
  - It’s not exactly 0.3

2. Small Decimal Values

- What if we *did* need to compare two numbers ?
  - Use a tiny "rounding error" value as the *tolerance* for comparison.
  - This tiny value is often called "machine epsilon,”
  - Which is commonly 2^-52 (2.220446049250313e-16)

2. Safe Integer Ranges

- Range of "safe" is significantly less than **Number.MAX_VALUE**.
- The maximum integer that can "safely" be represented is 2^53 - 1, which is **9007199254740991**.



2. Special Values – Non-value

- Both **undefined** and **null** are often taken to be interchangeable as either "empty" values or "non" values.
- Other developers prefer to distinguish between them with nuance. For example:
  - **null** is an empty value
  - **undefined** is a missing value

  Or:
  - **undefined** hasn't had a value yet
  - **null** had a value and doesn't anymore

2. Special Numbers

- **Not A Number**
- Any **mathematic** operation you perform without both operands being numbers (or values that can be interpreted as regular numbers in base 10 or base 16) will result in the operation failing to produce a valid number, in which case you will get the **NaN** value.

2. Special Numbers

- **NaN** literally stands for "not a number", though this label/description is very **poor and misleading**
- It would be much more accurate to think of **NaN** as being "invalid number," "failed number," or even "bad number," than to think of it as "not a number."

```
var a = 2 / "foo";                  // NaN

typeof a === "number";  // true
```

2. Special Numbers

- So, if you have a value in some variable and want to test to see if it's this special failed-number **NaN**, you might think you could directly compare to **NaN** itself, as you can with any other value, like **null** or **undefined**. Nope.

```
var a = 2 / "foo";

a == NaN;       // false
a === NaN;      // false
```

2. Special Numbers

- **NaN** is a very special value in that it's never equal to another **NaN** value (i.e., it's never equal to itself).
- So how *do* we test for it, if we can't compare to **NaN** (since that comparison would always fail)?

2. Special Numbers

- As of ES6, finally a replacement utility has been provided: **Number.isNaN(..)**. A simple polyfill for it so that you can safely check **NaN** values *now* even in pre-ES6 browsers is:

```
if (!Number.isNaN) {
        Number.isNaN = function(n) {
                return (
                        typeof n === "number" &&
                        window.isNaN( n )
                );
        };
}

var a = 2 / "foo";
var b = "foo";

Number.isNaN( a ); // true
Number.isNaN( b ); // false -- phew!
```

2. Special Numbers

- **NaNs** are probably a reality in a lot of real-world JS programs, either on purpose or by accident.
- It's a really good idea to use a reliable test, like **Number.isNaN(..)** as provided (or polyfilled), to recognize them properly.
- If you're currently using just **isNaN(..)** in a program, the sad reality is your program ***has a bug***, even if you haven't been bitten by it yet!

2. Special Numbers

- **Infinities**
- Developers from traditional compiled languages like C are probably used to seeing either a compiler error or runtime exception, like "Divide by zero," for an operation like:

```
var a = 1 / 0;
```

2. Special Numbers

- However, in JS, this operation is well-defined and results in the value Infinity (aka Number.POSITIVE_INFINITY). Unsurprisingly:

```
var a = 1 / 0;  // Infinity
var b = -1 / 0; // -Infinity
```

2. Value vs. Reference

- A reference in JS points at a (shared) **value**, so if you have 10 different references, they are all always distinct references to a single shared value; **none of them are references/pointers to each other.**

2. Value vs. Reference

- Instead, the ***type*** of the value **controls** whether that value will be assigned by value-copy or by reference-copy.

```
var a = 2;
var b = a; // `b` is always a copy of the value in `a`
b++;
a; // 2
b; // 3

var c = [1,2,3];
var d = c; // `d` is a reference to the shared `[1,2,3]` value
d.push( 4 );
c; // [1,2,3,4]
d; // [1,2,3,4]
```

2. Value vs. Reference

- **Simple values** (aka scalar primitives) are *always* assigned/passed by value-copy
- **Compound values:** objects (including arrays, and all boxed object wrappers) and functions -- *always* create a copy of the reference on assignment or passing

2. Value vs. Reference

- References are quite **powerful**
- The only control you have over **reference vs. value-copy** behavior is the **type** of the value itself

2. Quiz

- What is the output of below code ?

```
var a = 1;
var b = 2;

function swap(a, b) {
  var tmp = a;
  a = b;
  b = tmp;
}

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

2. Quiz

- What is the output of below code ?

```
var a = [];

function clear(x) {
  x = undefined;
}

clear(a);

console.log(a);
```

2. Values – Summary

- In JavaScript, **arrays** are simply numerically indexed collections of any value-type.
- **strings** are somewhat "array-like", but they have distinct behaviors and care must be taken if you want to treat them as arrays.
- **Numbers** in JavaScript include both "integers" and floating-point values.

2. Values – Summary

- Several special values are defined within the primitive types.
- The **null** type has just one value: **null**, and likewise the **undefined** type has just the undefined value.
- **undefined** is basically the default value in any variable or property if no other value is present.
- The **void** operator lets you create the **undefined** value from any other value.

2. Values – Summary

- **numbers** include several special values, like **NaN** (supposedly "Not a Number", but really more appropriately "invalid number"); **+Infinity** and **-Infinity**; and **-0**.
- Simple scalar primitives (strings, numbers, etc.) are assigned/passed by **value-copy**, but compound values (objects, etc.) are assigned/passed by **reference-copy**.
- References are not like references/pointers in other languages -- they're never pointed at other variables/references, only at the **underlying values**.

3. Natives

- Here's a list of the most commonly used natives (Object):
  - String()
  - Number()
  - Boolean()
  - Array()
  - Object()
  - Function()
  - **Date()**
  - **Error()**

3. Natives – Boxing Wrappers

- These object wrappers serve a very important purpose.
- Primitive values **don't have** properties or methods,
- To access **.length** or **.toString()** you need an object wrapper around the value

```
var a = "abc";

a.length; // 3
a.toUpperCase(); // "ABC"
```

3. Natives – Boxing Wrappers

- In general, there's basically no reason to use the object form directly.
- It's better to just let the boxing happen implicitly where necessary.
- In other words, never do things like *new String("abc"), new Number(42),* etc -- always prefer using the literal **primitive** values **"abc"** and **42**.

3. Natives – Boxing Wrappers

- There are some gotchas with using the object wrappers directly that you should be aware of if you ***do*** choose to ever use them. For example, consider Boolean wrapped values:

```
var a = new Boolean( false );

if (!a) {
        console.log( "Oops" ); // never runs
}
```

3. Natives – Boxing

- If you want to manually box a primitive value, you can use the **Object(..)** function (no **new** keyword):

```
var a = "abc";
var b = new String( a );
var c = Object( a );

typeof a; // "string"
typeof b; // "object"
typeof c; // "object"

b instanceof String; // true
c instanceof String; // true

Object.prototype.toString.call( b ); // "[object String]"
Object.prototype.toString.call( c ); // "[object String]"
```

3. Natives – Unboxing

- If you have an object wrapper and you want to get the underlying primitive value out, you can use the **valueOf()** method:

```
var a = new String( "abc" );
var b = new Number( 42 );
var c = new Boolean( true );

a.valueOf(); // "abc"
b.valueOf(); // 42
c.valueOf(); // true
```

3. Natives – Array(..)

- The Array constructor has a special form where if only one number argument is passed, instead of providing that value as ***contents*** of the array, it's taken as a length to "presize the array"

```
var a = new Array( 1, 2, 3 );
a; // [1, 2, 3]

var b = [1, 2, 3];
b; // [1, 2, 3]
```

3. Natives – Array(..)

- It doesn't help matters that this is yet another example where browser developer consoles vary on how they represent such an object, which breeds more confusion.

```
var a = new Array( 3 );

a.length; // 3
a;
```

3. Natives – Array(..)

- To visualize the difference, try this:

```
var a = new Array( 3 );
var b = [ undefined, undefined, undefined ];
var c = [];
c.length = 3;

a;
b;
c;
```

3. Natives – Array(..)

- To create an array and fill value to its:

```
Array(10).fill(0);
▶ (10) [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
new Array(10).fill(0);
▶ (10) [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

3. Natives – Date and Error

- The Date(..) and Error(..) native constructors are much more useful than the other natives, because there is **no literal form** for either.
- To create a **date** object value, you must use **new Date()**.
  The **Date(..)** constructor accepts optional arguments to specify the date/time to use, but if omitted, the current date/time is assumed.
- By far the most common reason you construct a date object is to get the current timestamp value (a signed integer number of milliseconds since Jan 1, 1970). You can do this by calling getTime() on a date object instance.

3. Natives – Date and Error

- **Date Usage:**

```
var d = new Date(); // current date
console.log(d);

// format: month/day/year
var yesterday = new Date('08/02/2020');
console.log(yesterday);
```

```
Mon Aug 03 2020 13:25:19 GMT+0700 (Indochina Time)
```

```
Sun Aug 02 2020 00:00:00 GMT+0700 (Indochina Time)
```

3. Natives – Date and Error

- **Date Usage:**

```
var d = new Date(); // current date

var dateUntilPaySlip = 19 - d.getDate() + 1;

console.log(`${dateUntilPaySlip} days left til pay slip`);
17 days left til pay slip
```

3. Natives – Date and Error

- The **Error(..)** constructor behaves the same with the **new** keyword present or omitted.
- The main reason you'd want to create an error object is that it **captures** the current **execution stack** context into the object
- This stack context includes the function call-stack and the line-number where the error object was created, which makes **debugging** that error much easier.

3. Natives – Summary

- JavaScript provides object wrappers around primitive values, known as **natives** (String, Number, Boolean, etc).
- These object wrappers give the values access to behaviors appropriate for each object subtype (String.trim(), Array.concat()).
- If you have a simple scalar primitive value like **"abc"** and you access its **length** property or some **String.prototype** method
- JS automatically "**boxes**" the value (wraps it in its respective object wrapper) so that the property/method accesses can be fulfilled.

4. Coercion

- Converting a value from one type to another is often called "type casting," when done **explicitly**,
- "coercion" when done **implicitly** (forced by the rules of how a value is used).
- Another terms: **"type casting"** (or "type conversion") occur in statically typed languages at compile time,
- "**type coercion**" is a runtime conversion for dynamically typed languages.

4. Coercion

- However, in JavaScript, most people refer to all these types of conversions as ***coercion:*** "implicit coercion" vs. "explicit coercion."

```
var a = 42;

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

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

4. Coercion - ToBoolean

- **Falsy Values**
- 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)

4. Coercion - ToBoolean

- We get the following as the so-called "falsy" values list:
  - undefined
  - null
  - false
  - +0, -0, and NaN
  - "" (empty string)

4. Coercion - ToBoolean

- **Truthy Values**
- What exactly are the truthy values? **a value is truthy if it's not on the falsy list.**

```
var a = "false";
var b = "0";
var c = "''";

var d = Boolean( a && b && c );

d;
```

4. Coercion - Explicit Coercion

- To convert number to string and vice versa:

```
var a = 42;
var b = String( a );

var c = "3.14";
var d = Number( c );

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

4. Coercion - Explicit Coercion

- **Explicitly: Parsing Numeric Strings**
- A similar outcome to coercing a string to a number can be achieved by parsing a number out of a string's character contents

```
var a = "42";
var b = "42px";

Number( a );     // 42
parseInt( a );   // 42

Number( b );     // NaN
parseInt( b );   // 42
```

4. Coercion - Explicit Coercion

- The pre-ES5 fix was simple, but so easy to forget: **always pass 10 (base 10) as the second argument**.
- This was totally safe:

```
var hour = parseInt( selectedHour.value, 10 );
var minute = parseInt( selectedMiniute.value, 10 );
```

4. Coercion - Implicit Coercion

- *Implicit* coercion refers to type conversions that are **hidden**, with non-obvious side-effects that implicitly occur from other actions.
- In other words, *implicit coercions* are any type conversions that aren't obvious (to you).

4. Coercion - Implicit Coercion

- **Implicitly: any => Boolean**
- It's by far the **most common** and also by far the most potentially **troublesome**.
- **Remember:** *implicit* coercion is what kicks in when you use a value in such a way that it forces the value to be converted

4. Coercion - Implicit Coercion

- But, what sort of expression operations **require/force** (*implicitly*) a boolean coercion?
  1. Test expression in an if (..) statement.
  2. Test expression (second clause) in a for (.. ; .. ; ..) loop.
  3. Test expression in while (..) and do..while(..) loops.
  4. Test expression (first clause) in ? : ternary expressions.
  5. left-hand operand to the || ("logical or") and && ("logical and") operators.

4. Coercion - Implicit Coercion

- Let's look at some examples:

```
var a = 42;
var b = "abc";
var c;
var d = null;

if (a) {
        console.log( "yep" );           // yep
}

while (c) {
        console.log( "nope, never runs" );
}

c = d ? a : b;
c;                                      // "abc"

if ((a && d) || c) {
        console.log( "yep" );           // yep
}
```

4. Coercion - Implicit Coercion

- **Loose Equals vs. Strict Equals**
- Loose equals is the == operator, and strict equals is the === operator. Both operators are used for comparing two values for "equality,"
- A very **common misconception** about these two operators is: == checks values for equality and === checks both values and types for equality
- The **correct description** is: == allows coercion in the equality comparison and === disallows coercion.

4. Coercion - Implicit Coercion

- **Comparing: strings to numbers**
- To illustrate == coercion, let's first build off the string and number examples earlier in this chapter:

```
var a = 42;
var b = "42";

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

4. Coercion - Implicit Coercion

- **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.

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

a == b; // false
```

4. Coercion – Summary

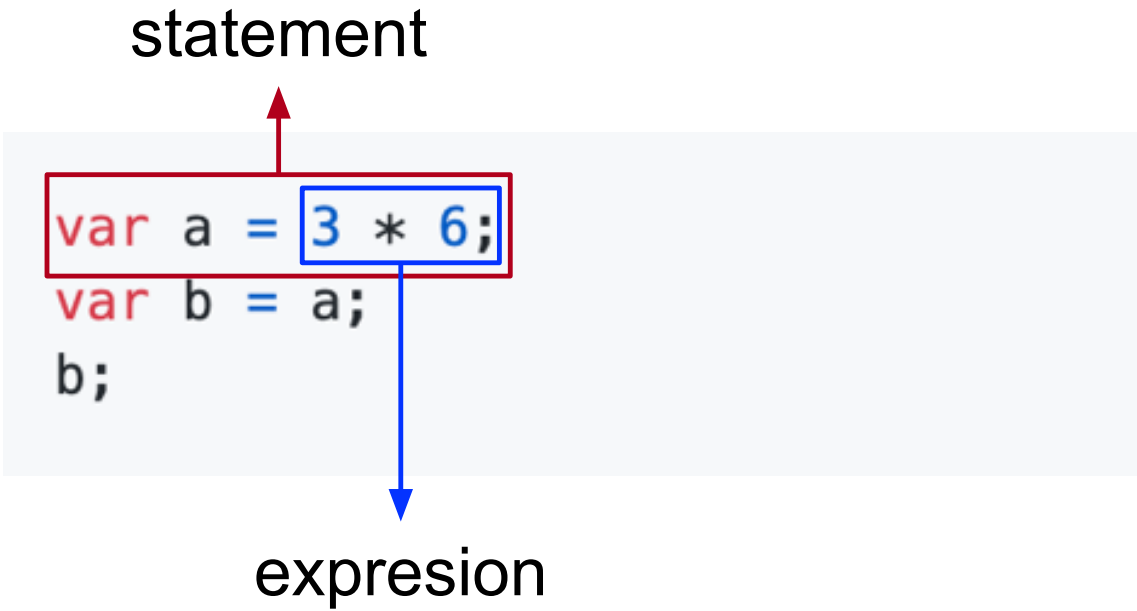
- **coercion:** JavaScript type conversions, which can be characterized as either *explicit* or *implicit*.
- *Explicit* coercion is code which is **obvious** that the intent is to convert a value from one type to another
- *Implicit* coercion is coercion that is "**hidden**" as a side-effect of some other operation, where it's not as obvious that the type conversion will occur
- Especially for *implicit*, coercion must be used **responsibly** and **consciously**

5. Grammar - Statements & Expressions

- A "**sentence**" is one complete formation of words that expresses a thought: it's comprised of one or more "**phrases**," each of which can be connected with punctuation marks or conjunction words ("and," "or," etc).
- And so it goes with JavaScript grammar. **Statements** are sentences, **expressions** are phrases, and **operators** are conjunctions/punctuation.

5. Grammar - Statements & Expressions

- Every **expression** can be evaluated to a **value** result.
- For example:



5. Grammar - Automatic Semicolons

- ASI (Automatic Semicolon Insertion) is when JavaScript assumes a ; in certain places in your JS program even if you didn't put one there.
- ASI allows JS to be tolerant of certain places where ; aren't commonly thought to be necessary.
- It's important to note that ASI will only take effect in the presence of a newline (aka line break).
- Semicolons are not inserted in the middle of a line.

5. Grammar - Automatic Semicolons

- Major case is with: **break**, **continue**, **return**:

```
function createUser(n) {
    return
    {
        name: n
    }
}
```

ASI

Always return undefined

```
function createUser(n) {
    return;
    {
        name: n
    }
}
```

5. Grammar - Automatic Semicolons

- **Always** use semicolons wherever you know they are "required," and limit your assumptions about ASI to a minimum.

Make sure no newline break after return

```
function createUser(n) {
    return {
        name: n
    }
}
```

5. Grammar – Summary

- Statements and expressions have analogs in English language -- statements are like sentences and expressions are like phrases.
- ASI (**Automatic Semicolon Insertion**) is a parser-error-correction mechanism built into the JS engine, which allows it under certain circumstances to insert an assumed ';' in places where it is required
- Always use semicolon ';'

Thank you