Values & Types:

Built in types available to values, not typed variables.

* `string`
* `number`
* `boolean`
* `null` and `undefined`
* `object`
* `symbol` (ES6)

Variables hold every different type, variables are simply containers for those values.

```
// typeof operator

var a;
typeof a;  // "undefined" -- Same as explicitely assigning a = undefined

a = "hello";
typeof a;  // "string"

a = true;
typeof a;   // "boolean"

a = null;
typeof a;
// "object" -- 'null returning object' a long-standing bug, so much so that fixing it
// apparently would cause more problems since some code on Web relies on it.

a= { b: "c" };
typeof a;    // "object"
```

`void` operator and functions that return no value are more examples of producing undefined values.

Objects:

Literal:

Named properties

```
var obj = {
  a: “hello world”,
  b: 33,
  c: true
};
```

```
// Bracket and dot notation
obj.a;
obj["a"];
```

Arrays:

```
var arr = ["hello", true, 33]
typeof arr;  // "object"  — object with property length and numerically positioned values.
```

Functions:

```
function foo() {
  return 33;
}

foo.bar = "hello!";

typeof foo;
typeof foo();
typeof foo.bar;
```

Built-In Type Methods:

```
var a = "hello!";
var b = 3.14159;

a.length;  // 5
a.toUpperCase();  // "HELLO!"
b.toFixed(4);  // "3.1416"
```

`String` object wrapper, called native, that pairs with primitive. This wrapper has `toUpperCase()` defined on its prototype.
Js automatically “boxes” the value to its object wrapper, when you use a primitive value as object by referencing a property or method. Prefer the primitive forms.

Comparing Values:

Two types of value comparison: equality and inequality.
The result is strictly boolean regardless of value type compared.

Coercion:

- explicit: see conversion in code from one type to another
- implicit: non-obvious side-effect of another operation, perhaps surprisingly leading to distrust

Truthy & falsy:

Falsy coercion values:
`""`, empty string
`NaN`, `-0`, `0`
`null`, `undefined`
`false`
All else truthy.

Equality:

`==`, checks for value equality with coercion allowed
`===`, checks without coercion or 'strict equality'
Although prudent to use strict equality at all times for comparisons, author argues they may not be required if you have an understanding of what values are coming through the variables of the given comparison. If certain, use `==`, if not use strict.

When comparing two non-primitive values, such as `objects`, `function` or `array`, because they are values held by reference, both `==` and `===` will check for reference match.

Example:

```
var arr = [1,2,3];
var arr2 = [1,2,3];
var split = "1,2,3";

arr == split;  // true
arr2 == split;  //true
arr == arr2;  //false
```

Inequality:

Rest of comparison operators, referred in spec as “relational comparison”.
Use on numbers and strings. (Lexicographical order)
When one of the values cannot be coerced to a number, NaN may result, which is neither greater than nor less than anything else.

Variables:

Variable names must be valid identifiers.
Must start with a-z, A-Z, $, or _ and alphanumeric thereafter.
Reserved words cannot be used as variable names, such as Js keywords if, in, for etc and true, null, false.

Function Scopes:

`var` keyword declares a variable that belongs to the current function scope, or the global scope if at the top level outside any function.

Hoisting:

When a var declaration appears within a scope, it is metaphorically “moved” to the top of its enclosing scope. Naturally the process is better explained by how code is compiled, but it is a good thought model.

```
var a = 2;
foo();

function foo() {
  a = 3;
  console.log(a); // 3
  var a;  // declaration is "hoisted"
}
console.log(a);  // 2
```

Nester scopes:

Variable is available inside any inner/lower scopes. `ReferenceError` gets thrown if variable value is not available in scope.

```
function foo() {
  var a =1;
  function bar() {
    var b = 2;
    function baz() {
      var c = 3;
      console.log(a,b,c);  // 1 2 3
    }
    baz();
    console.log(a,b);  // 1 2
  }
  bar();
  console.log(a);  // 1
}
foo();
```

Always formally declare variables with `var` keyword. Otherwise unwanted global variables result (or error in “strict mode”.)

ES6 `lets` allows block scoping - variable declaration belong to individual blocks `{ .. }`

Conditionals:

`if..else` (already covered) and `switch` statements

```
switch (a) {
  case 2:
    // do something!
    break;
  case 10:
    // do the 10 thing
    break;
```

```
  default:
    // default thing
}
```

Conditional operator or ternary operator:

```
var a = 42;
var b = (a > 41) ? "Yes!" : "Fail";
```

Strict Mode:

ES5, restrictions to keep code closer to a set of guidelines. Adhering to strict mode makes code generally more optimizable by the engine.

You can opt in for strict mode on individual function or an entire file, depending on where the pragma (“use strict”) is located.

Strict mode for example disallows the auto-global variable declaration from omitting `var`.

Functions as Values:

Function declaration is basically a variable in the outer enclosing scope that’s given a reference to the function being declared. The `function` itself has value that can be assigned to a variable or passed to or returned from other functions. In other words, a function value is an expression.

```
// Anonymous function
var foo = function() {
  // ...
}
// Named function expression
var x = function bar(){
  // ...
}
```

Immediately Invoked Function Expressions (IIFEs):

Execute a function expression when it is declared.

```
// function executed after declaration
function foo(){
  // ..
}
foo();

// function executed immediately
(function IIFE(){
  // ..
})();
```

As any function, IIFE will create a variable scope that won't affect the surrounding code, and is able to return values.

```
var x = (function IIFE(){
  return 42;
})();

x;  // 42
```

Closure:

Deeply misunderstood and important concept in Js.
In author's summary, closure is a way of "remembering" and continuing to accessibility a function's scope (its variables) even once the function has finished running.

```
function makeAdder(x) {
  // parameter 'x' is an inner variable
  // inner funciton 'add()' uses 'x' so it has a "closure" over it
  function add(y) {
    return y + x;
  }
  return add;
}
// We create a reference to inner add(..) that remembers x is a certain value.
var plusOne = makeAdder(1);
var plusTen = makeAdder(10);

plusOne(3);  // 4
plusOne(13); // 14
plusTen(11); // 21
```

Modules:

Most common usage of closure in Js, the module pattern.
Define private implementation details hidden from the outside world, and expose public API. More on this later naturally.

this Identifier:

keyword `this` also an often misunderstood part of Js
To find what this points to, one must examine how the function in question was called.

```
function foo() {
  console.log(this.bar);
}

var bar = "global";
var obj1 = {
  bar: "obj1",
  foo: foo
```

```
};
var obj2 = {
  bar: "obj2"
};
// Four rules for how this gets set.
foo();  // "global"
obj1.foo();  // "obj1"
foo.call(obj2); // "obj2"
new foo();  // undefined
```

1. `foo()` sets `this` to the global object in non-strict mode, and `undefined` in strict mode.
2. `obj1.foo()` sets `this` to the `obj1` object.
3. `foo.call(obj2)` sets `this` to the `obj2` object.
4. `new foo()` sets `this` to a brand new empty object.

Prototypes:

The prototype chain serves Js objects as an internal prototype reference linkage to “lookup” properties that are not found.
Built-in utility `Object.create(..)` is used to create and illustrate a reference link.

```
var foo = {
  a: 42
};
// create 'bar' and link to 'foo'
var bar = Object.create(foo);
bar.b = "hello world";
bar.b;  // "hello world"
bar.a;  // 42 -- delegated
```

The author argues although prototypes is used to emulate prototypical inheritance and class, a more natural way of applying them is a pattern called “behavior delegation”.

Old & New:

Bringing newer JS features to older browsers.

Polyfilling:

Coined by Remy Sharp, it is taking a newer feature and producing code that is equivalent in behavior that is able to run in older Js environments.
For example:
`NaN === NaN` is `false` and ES6 provides for non buggy way to check for `NaN`
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/isNaN
For older browsers the polyfill is:

```
Number.isNaN = Number.isNaN || function(value) {
    return value !== value;
}
```

ES5-Shim https://github.com/es-shims/es6-shim

Transpiling:

Transform + compile - converts new code syntax into a form readable with older syntax.
Step is added into build process, similar to lining/minifying. You can write in new syntax and browsers can use their older version.
For example: using default parameter values in ES6, as in `function foo(a = 2) { .. }`
This code will transpire in non-ES6 browsers to allow `undefined` to be a default:

```
// Newer syntax
function foo(a = 2) {
  console.log(a);
}
// Polyfill for the above
function foo() {
  // check to see if first argument is void 0 (undefined),
  var a = arguments[0] !== (void 0) ? arguments[0] : 2;
  console.log(a); // pass the default value as before
}

foo(); //  2
foo(13);  // 13
```

There are transpilers Babel and Traceur, Babel appears to be most popular.

Non-JavaScript:

Most common environment is the browser. The DOM API will be most familiar to Js devs
For example:
`var el = document.getElementById("foo");`
The `document` global variable exists as a special Js `object`, "host object"
Built-in methods provided by the DOM from your browser, like `getElementById()`
`alert()` and `console.log()` all hook into the browser.

Review:

We've only just begun. Learn the basics and then take a refreshing deep dive into the different areas of Js.