JS BASICS

- Also referred to as ECMAScript, ES5/ES6
- When including JS code to HTML, the following works

```
<script>
  console.log("I'm a script");
</script>

<script src="my-javascript.js"></script>

// This is problematic, code inside script tag is ignored
<script src="my-javascript.js">
  console.log('Hello, World!');
</script>
```

DATA TYPES

- Five primitive data types: numbers, boolean, string, null, undefined

- `NaN` is a primitive value
- Primitive string values can be enormous, compound objects can also be tiny

- Compound data type: object (arrays, functions) can be modified without losing their identity
- Numbers: JS uses floating point system to represent all numbers
- As a result, adding two decimal numbers result in discrepancies
- A few number values:
    - Infinity: number greater than any other number
    - -Infinity: number less than any other number
    - NaN: not a number, e.g. `Math.sqrt(-1)` or `2/0`
- Boolean represents the true-values of logic
- String: has no size limit, no distinction between single and double quoted strings. Some special chars include:

- \n new line
- \t tab
- \r carriage return
- \v vertical tab
- \b backspace

- To escape standard chars with single quoted strings, `'"It\'s hard to fail." - Roosevelt'`
- To escape standard chars with double quoted string, `"\"It's hard to fail.\" Roosevelt"`
- Accessing a character in a string can be done using the `String.prototype.charAt()` method on the string
    - e.g. `'hello'.charAt(1); // e`
    - bracket notation also works: `'hello'[1];`, note this is an operator and not a method in Ruby

Primitive values

- Primitive values are immutable: can't chanage them once they are created
- Strings may appear to change, but they are only reassigned

```
a = 'hello';
a.toUpperCase();  // the "hello" string is not mutated, but a new "HELLO" string is returned
a;                // still "hello"
```

- JavaScript doesn't actually change the values; instead, it assigns wholly new values to variables that used to contain different values. This means that you should remember to assign an expression to change the value in a variable; no function, method, or other operation will modify it for you. If you don't assign the new value to the desired variable, JavaScript won't do it for you.

Variables

- To declare a variable: `var number;`
- To assign value to a variable: `number = 3;`
- Combine declaration with initialization: `var myVar = 'Hi!;`
- If a var is declared but not assigned a value, is initialized to value `undefined`

```
var foo;
foo;      // undefined
```

Dynamic typing

- JS types dynamically, meaning a var can hold a reference to any data type, and can be reassigned a different type without error

```
// initialize to a string
var myVariable = 'Hello, World';

// reassign to a number
myVariable = 23;
```

```
// now the variable holds a number value
myVariable;      // 23
```

Arithmetic operators

- 10 % 3 returns 1 no matter whether % is treated as a remainder or modulo
- In JS, `10 % -3` returns 1, in other languages, `10 % -3` might return -2

Comparison operators

- JS performs implicit conversion, thus avoid using == or `!=` in favour of the stricter versions
- Strings can be compared lexicographically
- Logical `&&`:
    - `false && []; - false`
- Logical `||`:
    - `false || true; // true`
    - `false || []; // [](second operand is non-boolean, returned as is)`
- Logical Not `!`
    - `!1; //false`
    - `![]; //false`
- `"78" > "6"` returns `true` because JS compares char strings char by char, `"7" > "6"` in this case

Expressions

- An expression is any valid code that resolves to a value
- `var myNumber = 3;` contains both a statement (var declaration) and an expressions(assignment of 3 to myNumber)

```
'hello';   // a single string is an expression
'Hello' + ', World' // expressions that evaluates to a character string
`10 + 13;`   // arithmetic operations are expressions
`sum = 10;`  // assignments are expressions
`10 > 9;` // expressions that evaluate to true or false

var a;
var b;
var c;

a = 3;
b = 10 + 3;         // 10 + 3 is an expression that resolves to 13 and used as part of the assignment for sum
c = (a + (3 + b));  // a more complicated expression
```

- These two return values are different, why?

```
var foo = false;  // returns undefined

var foo;
foo = false;     // returns false
```

- Last line of the first expression is a statement, whereas the second line of the second set is an expression.
- Return value of a JS statement is `undefined`, return value of an expression with an operator is operator dependent

Variable references

- When a variable is declared but unassigned, JS initializes it to `undefined`

```
var myVariable = 'Hellow, World';
myVariable = 23;
```

- Changing the value a variable references does not alter the value. When reassign a variable, the original value does not change. A new value is assigned to the variable.

Increment/Decrement Operations in Expressions

```
var a = 1;
a++;        // same as "a = a + 1"; a is now 2
++a;        // same as "a = a + 1"; a is now 3
a--;        // same as "a = a - 1"; a is now 2
--a;        // same as "a = a - 1"; a is now 1
```

- Note how the prefix and postfix forms behave differently as part of more complex expression

```
var a;
var b;
var c;
```

```
a = 1;
b = a++;  // equivalent to "b = a; a++;". so now b is 1 and a is 2
c = ++a;  // equivalent to "++a; c = a;". so now c is 3 and a is 3
```

Short circuit evaluation in expressions

- For an expression like `a || b`, if a is true, the result is always true
- As with `a && b`, JS short circuits the evaluation if a is false, and returns false without evaluating b.

```
var a = true;
var b = false;

a || (b = true);  //true, b is still false after this, because (b = true) is never evaluated
b && (a = 1);     //false, a is still true after this, because (a = 1) is never evaluated
```

- With `||`, the expression executes until it reaches `true`, otherwise it's `false`
- With `&&`, the expression short-circuits as soon as a `true` is reached

```
var a = 0;

0 && (a += 1); // 0, expression evaluates to false && true, so false, which is 0 in this case
a && (a -= 1); // 0, expression evaluates to 0 / false && -1, so false, which is 0
a || (a += 1); // 1, expression evaluates to 0/false || 1, true, which is 1
```

Statements

- Statements in JS don't always evaluate to a value
- Statements control the execution of the program
- Var assignments are expressions, var declarations are statements

```
var a;                // a statement to declare variables
var b;
var c;
var b = (a = 1);      // this works, because assignments are expressions too
var c = (var a = 1);  // this gives an error, since a statement can't be used as part of an expression
```

- Statements help to do something, instead of giving a value to use
- `if ... else`, `switch`, and `while ... for` are also statements

JS Input and Output

- To get user input

```
var name = prompt('What is your name?');
var guess = prompt();           // blank prompt window
```

- To prompt message to user

```
alert('Hello, world');            // alert dialog box with a message
alert();                          // an empty alert dialog box
```

- Logging debugging messages

```
var name = prompt('What is your name?');
console.log('Hello, ' + name);
```

Explicit type coersions

- Primitive JS values can be converted to values of different types
- Since primitive datat types are immutable, values are not converted but a new value of the proper type is returned

```
// Turn strings that contain numeric value into number
Number('1');             // 1
Number('abc123');        // NaN

// Turn strings into numbers, each handling numeric values in an integer or floating-point format. The `radix` argument is specified fo
parseInt('123', 10);     // 123
parseInt('123.12', 10);  // 123, will only return whole numbers
parseInt('123.12');      // 123
parseFloat('123.12');    // 123.12, can handle floating point values

// Converting numbers into strings
String(123);             // "123"
String(1.23);            // "1.23"
(123).toString();        // "123"
(123.12).toString();     // "123.12"
123 + '';                // "123"
'' + 123.12;             // "123.12"
```

```
// Converting booleans into strings
String(true);            // "true"
String(false);           // "false"
true.toString();         // "true"
false.toString();        // "false"

// Comparing primitives to boolean
var a = 'true';
var b = 'false';
a === 'true';            // true
b === 'true';            // false

// Boolean conversions based on truthy/falsey rules
Boolean(null);           // false
Boolean(NaN);            // false
Boolean(0);              // false
Boolean('');             // false
Boolean(false);          // false
Boolean(undefined);      // false
Boolean('abc');          // true
Boolean(123);            // true
Boolean('true');         // true
Boolean('false');        // true

!!(null);                // false
!!(NaN);                 // false
!!(0);                   // false
!!('');                  // false
!!(false);               // false
!!(undefined);           // false

!!('abc');               // true
!!(123);                 // true
!!('true');              // true
!!('false');             // true
```

Implicit primitive type coersions

```
1 + true       // true is coerced to the number 1, so the result is 2
'4' + 3        // 3 is coerced to the string '3', so the result is '43'
false == 0     // false is coerced to the number 0, so the result is true

// Unary plus operator converts value to a number
+('123')        // 123
+(true)         // 1
+(false)        // 0
+('')           // 0
+(' ')          // 0
+('\n')         // 0
+(null)         // 0
+(undefined)    // NaN
+('a')          // NaN
+('1a')         // NaN

// + means either additions for numbers or concatenation for strings
// If + is used with mixed operands that includes a string, the other operand is converted to a string too
'123' + 123     // "123123" -- if a string is present, coerce for string concatenation
123 + '123'     // "123123"
null + 'a'      // "nulla" -- null is coerced to string
'' + true       // "true"

// When both operands are a combo of numbers, booleans, null, undefined, they are converted to numbers and added
1 + true        // 2
1 + false       // 1
true + false    // 1
null + false    // 0
null + null     // 0
1 + undefined   // NaN

// When one operand is an object (array or function), both are converted to strings and added
[1] + 2                    // "12"
[1] + '2'                  // "12"
[1, 2] + 3                 // "1,23"
[] + 5                     // "5"
```

```
[] + true                  // "true"
42 + {}                    // "42[object Object]"
(function foo() {}) + 42   // "function foo() {}42"

// Other arithmetic operators are only defined for numbers, so operands are converted to numbers
1 - true                // 0
'123' * 3               // 369 -- the string is coerced to a number
'8' - '1'               // 7
-'42'                   // -42
null - 42               // -42
false / true            // 0
true / false            // Infinity
'5' % 2                 // 1

// Equality operators are either strict equality operators or non-strict equality operators
1 === 1               // true
1 === '1'             // false
0 === false           // false
'' === undefined      // false
'' === 0              // false
true === 1            // false
'true' === true       // false

'42' == 42            // true
42 == '42'            // true
42 == 'a'             // false -- becomes 42 == NaN
0 == ''               // true -- becomes 0 == 0
0 == '\n'             // true -- becomes 0 == 0

// Boolean gets converted to a number with equality operators
42 == true            // false -- becomes 42 == 1
0 == false            // true -- becomes 0 == 0
'0' == false          // true -- becomes '0' == 0, then 0 == 0 (two conversions)
'' == false           // true -- becomes '' == 0, then 0 == 0
true == '1'           // true
true == 'true'        // false -- becomes 1 == 'true', then 1 == NaN

* What about the `==` operator is true?
  - When comparing a `number` and `string`, JS coerces `string` to `number`

// One operand is `null`, other is `undefined`, non-strict operator returns `true`. Both are `null`, or both are `undefined`, return va
null == undefined      // true
undefined == null      // true
null == null           // true
undefined == undefined // true
undefined == false     // false
null == false          // false
undefined == ''        // false
undefined === null     // false -- strict comparison

NaN == 0              // false
NaN == NaN            // false
NaN === NaN           // false -- even with the strict operator
NaN != NaN            // true -- NaN is the only JavaScript value not equal to itself

11 > '9'              // true -- '9' is coerced to 9
'11' > 9              // true -- '11' is coerced to 11
123 > 'a'             // false -- 'a' is coerced to NaN; any comparison with NaN is false
123 <= 'a'            // also false
true > null           // true -- becomes 1 > 0
true > false          // true -- also becomes 1 > 0
null <= false         // true -- becomes 0 <= 0
undefined >= 1        // false -- becomes NaN >= 1
```

Methods used for type conversion

- Coersing string to numeric: `parse(num, 10)`
- To join numbers into a string: `'' + npa`, or `String(npa)`
- Another method to convert values to strings: `bool.toString()`

Truthy, falsy and conditionals

- Possible `falsy` values, including boolean `false`, otherwise all truthy

```
if (false)        // falsy
if (null)         // falsy
if (undefined)    // falsy
if (0)            // falsy
if (NaN)          // falsy
if ('')           // falsy

if (true)         // truthy
if (1)            // truthy
if ('abc')        // truthy
if ([])           // truthy
if ({})           // truthy

// With the logical operator the return values are such:
1 || 2;           // 1
1 && 2;           // 2

// Using the logical operator as a `condition` in an if statement
if (1 || 2)       // truthy
if (1 && 2)       // truthy
```

- Switch statement: the `switch` will continue to the next cases following it until it reaches `default` or `break` statement, otherwise, use `break`

```
var reaction = 'negative';
switch (reaction) {
  case 'positive':
    console.log('The sentiment of the market is positive');
  case 'negative':
    console.log('The sentiment of the market is negative');
  case 'undecided':
    console.log('The sentiment of the market is undecided');
  default:
    console.log('The market has not reacted enough');
}
// console output
The sentiment of the market is negative
The sentiment of the market is undecided
The market has not reacted enough
```

Comparing values with NaN

```
console.log(Number('abc'));  // NaN
console.log(Math.sqrt(-1));  // NaN
console.log(undefined + 1);  // NaN
console.log(typeof(NaN)); // number

// NaN is a JS number
console.log(typeof(NaN));    // number

// Comparing NaN to any value evaluates to `false`
console.log(10 === NaN);     // false
console.log(10 < NaN);       // false
console.log(10 > NaN);       // false
console.log(NaN === NaN);    // false
//
```

- How to check if a variable holds NaN? `console.log(isNaN(foo));`
  - Returns `true` for any value that's not numeric
  - `console.log(isNaN('hello')); // true`
  - We can test for `NaN` directly or check for numeric

```
function isValueNaN(value) {
  return value!== value;
}

function isValueNaN(value) {
  return typeof value === 'number' && isNaN(value);
}
```

Looping

```
while(condition) {
  // statements
}
```

- Executes the statements in loop body if has truthy value

- `break` exits a loop immediately
- `continue` skips current iteration of a loop, returns to top of loop
- `do ... while` similar to `while`, except it will always execute once
- `for` sets the three key elements of a loop: initial state setting, evaluting condition, makign change before re-evaluation
-