Types

- REC

What is a *type*?

- **Kyle Simpson:**

  "[A] type is an intrinsic, built-in set of characteristics that uniquely identifies the behavior of a particular value and distinguishes it from other values, both to the engine and to the developer."

```
var myString = "4"
var myNum = 4


/* Both appear the same when
 logged to the console */

console.log(myString)              // 4
console.log(myNum)                 // 4


/* But they behave differently... */
console.log(myString + myString)   // 44
console.log(myNum + myNum)        // 8
```

Types in JavaScript

String	`"hello"`	Primitive
Number	`2`	
Boolean	`false`	
Undefined	`undefined`	
Null	`null`	

Object	Object	`{property: "value"}`
	Array	`[1,2,3]`
	Date	`new Date()`
	RegExp	`/.*/g,`
	Function	`function(){}`

Numbers

Simply typed numeric digits.

How many cookies do we have?

Includes integers, positive/negative, and decimal (floating point numbers)

```
// Integers are numbers
var x = 5;

// Fractional values are numbers
var y = 5.5;

// Negative values are numbers
var z = -3;
```

Basic Arithmetic Operators

- Used to perform operations between numerical pieces of data.
- Common Arithmetic Operators
  - Addition: +
  - Subtraction: -
  - Multiplication: *
  - Division: /
  - Remainder(Modulus): %
  - Can anyone think of a use case for this?

```
// Addition
var sum = 5 + 5;

// Subtraction
var dif = 5 - 5;

// Multiplication
var prod = 5 * 5;

// Division
var quot = 5 / 5;

// Modulus
var remainder = 5 % 5;
```

Immutability & Shorthand

- Operations on numbers DO NOT modify the numbers they operate on.
- This is true of all primitives in JavaScript. Modification requires *reassignment*.
- This syntax works the same for subtraction, multiplication, and division.

```
var num = 12;
num / 2;
// does nothing. num still 12

num += 2; // num is now 14
myNum ++; // num is now 15

console.log(myNum) // 15
```

Booleans

- For use when there are ONLY two possible states: true or false.
- Do I have any cookies left?
- true and false are <u>RESERVED words</u>.
- Attempting to use a reserved word as a variable name will cause the interpreter to throw an error.

```
var false = "hello";
// SyntaxError: Unexpected token false.
```

Null

- Used often as a placeholder to represent data that could be present but is currently ‘null’ in value.
    - Not to be confused with 0 or undefined.

```
var middleName = null;
```

Undefined

- A variable that has been DECLARED but a value has not been defined.
    - This is different then null because the value has NOT been set.

```
var middleName;
```

Strings

- A way of keeping a word/phrase/character as a data type.
  - Who's eating all of my cookies?
- Any information that is wrapped in quotes
- (single ' or ")

```
var name = 'Corey';

var monster = "Say 'Hello' to me.";

var vegMonster = 'This is not Corey';
```

Concatenation

- The process of putting two things together.
  - NOT addition.
- But like addition, uses the + operator

```
var firstName = 'Amy';
var lastName = 'Smith';


var fullName = firstName + ' ' + lastName;

console.log(fullName)  // Amy Smith
```

Accessing a String

- Strings are made up of characters
  - Each character has an ‘index’ representing its location within the string.
  - These indices begin at 0.
- So to access the first letter we would write str[0]

```
var myDog = 'Fluffy';

console.log(myDog[0])  // F
console.log(myDog[1])  // l

console.log(myDog.length)  // 6
```

Built-in String Methods

- `[] or .charAt()` // Reference single string character
- `.slice()` **or** `.substring()`
  // returns a copy of a portion of a string
- `.indexOf()` // returns the index of the first occurrence of value
- `.toUpperCase()` // returns an uppercase version of the string
- `.toLowerCase()` // returns a lowercase version of the string
- `.split()` // Splits string into array using arg as delimiter
- `.length` // (actually a property) the length of the string
- And more: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/prototype

The `typeof` operator

- The `typeof` operator checks the type of the value it precedes, and returns a string value that (mostly) indicates its type.
- Note that `typeof [1,2,3]` returns `"object"` because arrays are of type object.

```
typeof undefined     === "undefined"; // true
typeof true          === "boolean";   // true
typeof 42            === "number";    // true
typeof "42"          === "string";    // true
typeof [1, 2, 3]     === "object";    // true

/* Some notes:

1.typeof returns a string
2.the first letter of the string is LOWERCASE
3.using typeof on an array returns "object"
  AS IT SHOULD

*/

// ODDLY:
typeof null          === "object";    // true
```

Changes in Type

- Coercion occurs when a value changes (is "coerced") from one type to another.

- Recall the built in types in JavaScript (ES5):

```
1. null
2. undefined
3. boolean
4. number
5. string
6. object
```

```
var myNum = 4;
var myString = "8";

var combined = myNum + myString;
console.log(combined)           // 48
console.log(typeof combined)    // string
console.log(typeof myNum)       // number


/*

How did adding myNumber produce a string?

When "added", the underlying value was
first converted to a string and then
concatenated to myString.

This did NOT change the type of myNum

*/
```

When does Coercion happen?

- **Coercion occurs in two primary places:**
  - 1. Operations
    - `myNum + myStr`
  - 2. Test Expressions
    - `if (bool) { … }`
- **Coercion *always* results in a primitive value.**
  - *Operations* can result in any primitive.
  - *Test expressions* will coerce to a boolean

```
// Coercion with operations
var strOne = "1"
var strTwo = "2"
var sum = strOne*2 + strTwo;

// Multiplication operation coerces to num
// "+" operation coerces back to string
console.log(sum)   // 22
```

```
var myStr = "hello world";

// Expression in 'if' statement is
// a 'test expression', coerced to boolean
if (myStr) {
    console.log("coerced to true");
} else {
    console.log("coerced to false");
}  // logs: coerced to true
```

Implicit vs. Explicit

- "explicit coercion" is when it is obvious from looking at the code that a type conversion is intentionally occurring
- "implicit coercion" is when the type conversion will occur as a less obvious side effect of some other intentional operation.

```
var a = 42;

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

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

Knowing the Result of Coercion

- Coercion rules are set by the ECMA Script Specification.
- Don’t focus on trying to memorize every possible permutation of coercion. Instead, understand the process exists, and use the `` `typeof` `` operator to check a value if you’re unsure.
- This lecture will focus on coercion that results in a boolean value. This is the kind of coercion that occurs in test expressions:
  - If blocks, while blocks, for blocks, ternary expressions

Truthy/Falsey

underlying boolean value

Coerced to Boolean

- Every JavaScript value and expression can be coerced to a boolean.
- Values that coerce to `true` are referred to as "truthy". Those that coerce to `false` are "falsey".
- If the interpreter expects a boolean it will coerce your value to one.

```
var myStr = "false";
var myNum = 12;
var myArr = [1, 2, 3];
var myNull = null;
var myUndefined = undefined;

// In each of the following instances, the
// interpreter expects a boolean

if (myArr) { … }

while (myStr) { … }

for (var i=0; myNum; myNum--) { … }

myNull || myUndefined && myStr

// if the value is not already a boolean
// it will be coerced to one
```

! (logical NOT)

- `!` is the 'logical NOT' operator (also called the 'bang' operator).
- It converts whatever value follows to boolean, and then swaps `true` to `false` and vice versa.
- Accordingly, using `!!` before a value will coerce the value to it's boolean.

```
// The 'bang' operator toggles the boolean following
var trueBool = true;
var falseBool = false;

console.log(!trueBool)    // false
console.log(!falseBool)    // true

// If the value that follows is NOT a boolean
// The 'bang' operator first coerces it to boolean

console.log(!0)    // true
console.log(!"hello world")    // false
```

```
// Using the ! operator "bang bang" (one right after
// the other) will reveal the underlying boolean
// value for any term
console.log(!!"hello world")   // true

// therefore we can say that the string "hello world"
// is a truth value
```

truthy or falsey?

- There is a simple way to know whether a value is truthy or falsey.

- The following values are falsey:

1. `false`
2. `0`
3. `''` and `""`
4. `null`
5. `undefined`
6. `NaN`

Everything else is truthy!

```
/*
    Falsey values in JS:
*/

console.log(!!false)    // false
console.log(!!0)    // false
console.log(!!"")    // false
console.log(!!null)    // false
console.log(!!undefined)    // false
console.log(!!NaN)    // false
```

```
/*
   All other values are truthy!!
*/

console.log(!!true)    // true
console.log(!!-1)    // true
console.log(!!"false")    // true
console.log(!![null])    // true
```

How can we use this?

- Now we can make our 'test expressions' more concise.

- For example there's no reason to test whether a value `=== 0` or whether a string is empty.

```
/*
    Old way to log even values
*/

for (var i=0; i<10; i++) {
    if (i % 2 === 0) {
         console.log("value is even!");
    }
}
```

```
/*
    Taking advantage of coercion and truthy falsey
*/

for (var i=0; i<10; i++) {
    if (!(i % 2)) {
         console.log("value is even!");
    }
}

// If even, i % 2 is 0 (falsey). So precede it
// with the bang operator to get truthy
```

Quick Practice

```
/*
What would the following expressions log out?
*/

!!5
!!(4 % 2)
!!(undefined)
!!("a".length - 1)
!!([false])
!!([])
```

=== vs. ==

Strict Equality (===)

- **Only returns true if:**
  - Values compared are the same AND
  - Values compared are same type

```
1 === 1;      // => true
1 === 2;      // => false
1 === '1';    // => false

'' === false; // => false
true === 1;   // => false
```

Loose Equality (==)

- **Returns true if:**
  - Values compared are the same AFTER being coerced into the same type

```
1 == 1;      // => true
1 == 2;      // => false
1 == '1';    // => true

'' == false; // => true
true == 1;   // => true
```

Loose Equality (==)

- **What coercion rules does == use?**
  - There are 36 (see MDN table <u>here</u> for every possible combination of types)
- **What would a programmer do?**
  - Memorize all 36 rules and apply them perfectly in every line of code the write? ✗
  - Almost never use the loose equality operator unless they have a very, VERY good reason for doing so? ✓