**You don’t Know JS - UP & GOING**

**Statement:** a group of words, numbers, and operators that performs a specific task

**Variables:**  holds values

E**xpression:**  is any reference to a variable or value, or a set of variable(s) and value(s) combined with operators.

* literal value expression : 2
* variable expression:b
* arithmetic expression:b\*2
* assignment expression a= b\*2
* call expression alert(a)

**Interpreter**/**compiler**: is used to translate the code you write into commands a computer can understand.

**Interpreting the code**: Translation of commands. It is typically done from top to bottom, line by line, every time the program is run.

**Compiling the code**: translation is done ahead of time

The **JavaScript engine** actually **compiles** the program on the fly and then immediately **runs** the compiled code.

**Operators:**  are how we perform actions on variables and values. = ,\*

**Common operators**:

* Assignment: =
* Math: -+ (addition), - (subtraction), \* (multiplication), and / (division)
* Compound assignment: +=, -=, \*=, and /=
* Increment/decrement: ++, --
* Object property access: console.log()
* Equality: == (loose-equals), === (strict-equals), != (loose not-equals), !== (strict not-equals), as in a == b.
* Comparison: < (less than), > (greater than), <= (less than or loose-equals), >= (greater than or loose-equals), as in a <= b. See “Values & Types” on page 10 and Chapter 2. Logical && (and), || (or), as in a || b that selects either a or b.

**Primitive values**

* When you need to do math, you want a number.
* When you need to print a value on the screen, you need a string (one or more characters, words, or sentences).
* When you need to make a decision in your program, you need a Boolean (true or false).

**Literals** : Values that are included directly in the source code.

**Coercion**: If you have a number but need to print it on the screen, you need to convert the value to a string, and in JavaScript this conversion is called “coercion.” The first console.log(..) command has to implicitly coerce that number value to a string to print it out.

**Static typing (type enforcement):** is typically cited as a benefit for program correctness by preventing unintended value conversions.

**Weak typing(dynamic typing):**  allows a variable to hold any type of value at any time

**Constants** : when you declare a variable with a value and intend for that value to not change throughout the program. Another common usage of variables is for centralizing value setting.

**Block:**  group a series of statements together

**Conditionals** – if , switch(shorthand for a series of if..else statements), loops etc

**Loop:** A loop includes the test condition as well as a block (typically as { .. }). Each time the loop block executes, that’s called an **iteration**

**Function:** is generally a named section of code that can be “called” by name, and the code inside it will be run each time.

**Scope:**  is basically a collection of variables as well as the rules for how those variables are accessed by name. Only code inside that function can access that function’s scoped variables.

**Lexical scope:**  code in one scope can access variables of either that scope or any scope outside of it.

**Into JavaScript**

**Built-in types**

* string
* number
* Boolean
* null and undefined
* object
* symbol (new to ES6)

**typeof null** is an interesting case because it errantly returns "object" when you’d expect it to return "null".

**Object:** The object type refers to a compound value where you can set properties (named locations) that each hold their own values of any type. dot notation - obj.a or bracket notation - obj["a"]

**Array:** is an object that holds values (of any type) not particularly in named properties/keys, but rather in numerically indexed positions. var arr = [ "hello world", 42, true ]; arr[0]

**Function:** The other **object subtype** you’ll use all over your JS programs is a function: function foo(){return 42; }

**Built-In Type Methods** The built-in types and subtypes we’ve just discussed have behaviors exposed as properties and methods that are quite powerful and useful. Ex: a.length; a.toUpperCase(); b.toFixed(4); // "3.1416

A string value can be wrapped by a String object, a number can be wrapped by a Number object, and a boolean can be wrapped by a Boolean object. JS automatically “boxes” the value to its object wrapper counterpart (hidden under the covers)

**Comparing Values** There are two main types of value comparison that you will need to make in your JS programs: equality and inequality. The result of any comparison is a strictly boolean value (true or false), regardless of what value types are compared.

**Coercion** comes in two forms in JavaScript**: explicit and implicit**. Explicit coercion is simply that you can see from the code that a conversion from one type to another will occur, whereas implicit coercion is when the type conversion can happen as more of a nonobvious side effect of some other operation.

a non-boolean value is coerced to a Boolean • "" (empty string) • 0, -0, NaN (invalid number) • null, undefined • false Any value that’s not on this “falsy” list is “truthy.” Here are some examples of those: • "hello" • 42 • true • [ ], [ 1, "2", 3 ] (arrays) • { }, { a: 42 } (objects) • function foo() { .. } (functions)

Equality: There are four equality operators: ==, ===, !=, and !==. The ! forms are of course the symmetric “not equal” versions of their counterparts; non-equality should not be confused with inequality.

== looseequality comparison and not allowed with the === strict-equality

Math: floor - Round a number downward to its nearest integer

Math.floor(1.6); 🡪 1; var e = Math.floor(-5.1); 🡪 -6

Math.random();

The random() method returns a random number from 0 (inclusive) up to but not including 1 (exclusive).

The first parameter is a function to be executed.

The second parameter indicates the number of milliseconds before execution.

window.clearTimeout(timeoutVariable)

window.setTimeout(function, milliseconds);

myVar = setTimeout(*function*,*milliseconds*);  
clearTimeout(myVar);

If the function has not already been executed, you can stop the execution by calling the clearTimeout() method: