# Equality Comparison Operator

Using == as a comparator allows Javascript to do type conversion

but

using === is a stricter form of comparison that does NOT allow conversion

# Expressions and Statements

• Statements does something. Statements affect the world (are external).

• Expressions evaluates to a value. Expressions affect a statement (internal), also called side effects.

# Break

This is a different way of breaking out of a for loop.

for (let current = 20; ; current = current + 1) {

if (current % 7 == 0) { console.log(current); break;

}

// → 21

# Cleaner Nested If Statements

if (x == "value1") action1();

else if (x == "value2") action2();

else if (x == "value3") action3();

else defaultAction();

# Switch

switch (prompt("What is the weather like?")) {

case "rainy":

console.log("Remember to bring an umbrella.");

break;

case "sunny":

console.log("Dress lightly.");

case "cloudy":

console.log("Go outside.");

break;

default:

console.log("Unknown weather type!");

break;

}

# Binding and Scopes

The part of the binding that is visible is called the scope.

A binding that is outside any function is called *global.*

A binding that is inside a function parameter or body is called *local.*

Every time the function is called, new instances of these bindings are created.

# Ways Functions() can be Written

*Function Expressions are written as (notice the word let in there. This means that the function binding can still be updated later since it is not const):*

Let launchMissiles = function(x) {

Console.log(‘launched’ + x)

}

*Function declaration is written as (important: you can use the function BEFORE it is declared. Function declarations do not follow the normal top-to-bottom flow. Function declarations are moved to the top of their scope. ):*

*launchMissiles(x)*

Function launchMissiles(x) {

Console.log(‘launched’ + x)

}

*Arrow functions (they are like function expressions; they are NOT hoisted to the top of the scope).*

# Closure

When you wrap local variables in a function, those local variables are contained only within that instance of the function. That is why you can call wrapValue(n) multiple times here and the local variable doesn’t carry from one call to the next:

function wrapValue(n) {

let local = n;

return () => local;

}

let wrap1 = wrapValue(1);

let wrap2 = wrapValue(2);

console.log(wrap1());

// → 1

console.log(wrap2());

// → 2

**Technical definition**: “A function that references bindings from local scopes around”

# Recursion

For simple implementations, it is slower to use recursion than a for-loop.

# Side Effect

Any application state change that is observable outside the called function other than its return value.

# Data Structures

Subtle difference between the two ways of accessing property below is in the way the x is interpreted:

• Value.x

• Value[x]

When using a dot, the word after the dot is the literal name of the property.

When using square brackets, the expression between the brackets is evaluated to get the property name. Like how in arrays[2], “2” is not the literal name of the property in the array, “2” has to be evaluated to be the third index of an array.

Note that value.length returns the number of characters for a string value. But for arrays, it is not array[“length”] but simply *array*.

# Mutability

Immutable values: numbers, strings, Booleans

Objects can have the same value but they still point to a different memory address. Just like in Java,

let object1 = {value: 10};

let object2 = object1;

let object3 = {value: 10};

console.log(object1 == object2);

// → true

console.log(object1 == object3);

// → false

object1.value = 15;

console.log(object2.value);

// → 15

console.log(object3.value);

// → 10

# Array Loop

We loop through an array like this:

for (let i = 0; i < JOURNAL.length; i++) {

let entry = JOURNAL[i];

// Do something with entry

}

There is a shorter, more modern way to do this in JavaScript:

for (let entry of JOURNAL) {

console.log(`${entry.events.length} events.`);

}

# Some Useful Methods of String

• slice

• indexOf

• trim

• split

# Rest parameters

Write (…numbers) instead of the entire array:

function max(...numbers) {

let result = -Infinity;

for (let number of numbers) {

if (number > result) result = number;

}

return result;

}

console.log(max(4, 1, 9, -2));

// → 9

# Destructuring

Notice that by wrapping the binding in braces for name, we easily grab the object’s name:

let {name} = {name: "Faraji", age: 23};

console.log(name);

// → Faraji

# Higher Order Functions

Functions that operate on other functions, either by taking them as arguments or by returning them.

It allows us to abstract over actions, not just values.

# Encapsulation

Divide program into its own isolated piece. The individual pieces don’t need to be aware of HOW the other pieces work or their implementation. All it needs to know is the input/output.

The program interacts with each other through interfaces, which are limited sets of functions or bindings. Private properties are only available by an individual piece whereas properties that are available to all pieces are called public.

# This

This is a binding. It automatically points at the object that it was called on.

function speak(line) {

console.log(`The ${this.type} rabbit says '${line}'`);

}

let whiteRabbit = {type: "white", speak};

whiteRabbit.speak("Oh my ears and whiskers, " +

"how late it's getting!");

// → The white rabbit says 'Oh my ears and whiskers, how

// late it's getting!'

# Prototypes

A prototype is another object that is used as a fallback source of properties.

When an object gets a request for a property that it does not have, its prototype will be searched for the property, then the prototype’s prototype, and so on:

let empty = {};

console.log(empty.toString);

// → function toString()…{}

console.log(empty.toString());

// → [object Object]

What is the prototype of an empty object? Object.prototype.

Likewise, functions and arrays have their own prototypes: Function.prototype and Array.prototype.

The prototype of a constructor is Function.prototype.

# Classes

Prototypes are for defining properties for which all instances of a class share the same value, such as methods.

A class defines what methods and properties it has.

Objects are said to be an instance of the class.

A constructor makes sure that an object has the properties of the class it is from.

Putting the word new in front of a function call makes it a constructor. It will automatically create an object with the right prototype.

# Class Notation

Classes are constructor functions with a prototype property.

class Rabbit {

constructor(type) {

this.type = type;

}

speak(line) {

console.log(`The ${this.type} rabbit says '${line}'`);

}

}

let killerRabbit = new Rabbit("killer");

let blackRabbit = new Rabbit("black");

# Polymorphism

…