Modulo 2. Variables, Tipos de Datos, Conversión de Tipos de Datos y Comentarios

Las variables. Imagína a las variables como contenedores en los que puedes almacenar cierta información (dicha información se denominará valores de variables). Cada contenedor debe tener su propio nombre, mediante el cual podremos indicarlo claramente.

Por lo general, tenemos bastante libertad cuando se trata de inventar estos nombres, pero recuerda que deben referirse a lo que almacenaremos en la variable (por ejemplo, altura, color, contador de pasos, etc.). Por supuesto, JavaScript no verificará la correlación entre el nombre y el contenido de la variable; es simplemente una de las muchas buenas prácticas que facilitan que nosotros y otros entendamos el código más adelante.

En la mayoría de los lenguajes de programación, se debe declarar una variable antes de usarla, y JavaScript no es una excepción. Declarar una variable es simplemente "reservar" el nombre de la variable. De esta manera, le informamos al programa que en la parte posterior de la ejecución, usaremos este nombre para referirnos a nuestro contenedor, para recuperar un valor de él o guardar un valor en él.

En JavaScript, los nombres de las variables pueden constar de cualquier secuencia de letras (minúsculas y mayúsculas), dígitos, guiones bajos y signos de dólar, pero no deben comenzar con un dígito. Hay una lista de palabras reservadas que no se pueden usar como nombres de variables (consulta la tabla a continuación).

Lo importante también es que el intérprete de JavaScript distingue entre minúsculas y mayúsculas, también en nombres de variables, por lo que nombres como `test`, `Test`, o `TEST` serán tratados como variables diferentes.

Son partes integrales del lenguaje y se les asigna un significado que no se puede cambiar.

abstract	arguments	await	boolean
break	byte	case	catch
char	class	const	continue
debugger	default	delete	do
double	else	enum	eval
export	extends	false	final
finally	float	for	function
goto	implements	if	import
in	instanceof	int	interface
let	long	native	new
null	package	private	protected
public	return	short	static
super	switch	synchronized	this
throw	throws	transient	true
try	typeof	var	void
volatile	while	with	yield

Para las declaraciones, usamos las palabras clave `var` o `let` para **variables** y `const` para **constantes**

La palabra clave `var` proviene de la sintaxis JavaScript original, y la palabra clave `let` se introdujo mucho más tarde. Por lo tanto var se encontrara en programas más antiguos. Actualmente, se recomienda enfáticamente usar la palabra `let`

Una de las diferencias básicas en el uso de `var` y `let` es que `let` nos impide declarar otra variable con el mismo nombre (se genera un error). El uso de `var` permite volver a declarar una variable, lo que puede generar errores en la ejecución del programa.

Al declarar con `let`, el intérprete verifica si dicha variable ya ha sido declarada, sin importar si se usó let o var en la declaración anterior.

La variable debe ser **inicializada**, en otras palabras, debe recibir su primer valor. La **inicialización** se realiza asignando un determinado valor a una variable (indicado por su nombre). Para asignarlo usamos el operador =.

```
var test = "hola";
console.log(test);

//Imprimira "hola" por que es el valor que se le asigno a la variable 'test'
```

Puedes asignar a una variable: un valor específico; el contenido de otra variable; o, por ejemplo, el resultado devuelto por una función.

```
var test = "hola";
var hola;

hola = test;
```

La inicialización se puede realizar junto con la declaración o por separado como un comando independiente. Es importante ingresar el primer valor en la variable antes de intentar leerla, modificarla o mostrarla.

Al principio de nuestro código, agregamos `"use strict";`. Esta sentencia ha cambiado radicalmente el comportamiento del intérprete. ¿Por qué? Lo usamos cuando queremos obligar al intérprete a comportarse de acuerdo con los estándares modernos de JavaScript. Entonces, siempre que no estés ejecutando un código realmente antiguo, siempre debes usarlo. Y esta vez, usar una variable sin su declaración anterior se trata como un error.

```
"use strict";

height = 180; // -> Uncaught ReferenceError: height is not defined
console.log(height);
```

Eso podria aplicar si queremos depurar nuestro codigo y tiene codigo antiguo.

La palabra clave `const` se usa para declarar contenedores similares a variables. Dichos contenedores se denominan **constantes**. Las constantes también se utilizan para almacenar ciertos valores, pero una vez que se han ingresado valores durante la inicialización, ya no se pueden modificar. Esto significa que este tipo de contenedor se declara e inicializa simultáneamente. Por ejemplo, la siguiente declaración de la constante greeting es correcta:

```
const greeting = "¡Hola!";
```

El propósito principal de una constante es erradicar la posibilidad de cambiar accidentalmente un valor almacenado en ella. Esto es importante cuando tenemos algunos valores que realmente nunca deberían cambiar. Los ejemplos típicos de constantes son rutas a recursos, tokens y otros datos que nunca cambian durante la vida útil del script.

El uso de una const, además de evitar que un valor se cambie por error, permite que el motor de JavaScript optimice el código, lo que puede afectar su rendimiento.

Ocupar `const` y `let` dentro de un bloque de codigo, tendra el comportamiento de si es global o no, dependiente.

```
const test; //Global

{
  let hola; local
}
```

En el caso de declaraciones de variables usando la palabra clave `var`, la situación es ligeramente diferente. La variable declarada utilizándola fuera de los bloques será, como en el caso de let, global, es decir, será visible en todas partes. Si la declaras dentro de un bloque, entonces... bueno, por lo general volverá a ser global.

¿Las variables declaradas usando `var` siempre, independientemente del lugar de declaración, serán globales? Definitivamente no. El problema es que `var` ignora los bloques de programa ordinarios, tratándolos como si no existieran. Entonces, ¿en qué situación podemos declarar una variable local usando var? Sólo dentro de una función.

Si declaramos una variable usando la palabra clave var dentro de una función, su alcance se limitará solo al interior de esa función (es un alcance local). Esto significa que el nombre de la variable se reconocerá correctamente solo dentro de esta función.

JavaScript permite **sombreado de variables.** ¿Que significa eso? Significa que podemos declarar una variable global y una variable local con el mismo nombre.

En el alcance local, en el que declaramos una variable local usando su nombre, tendremos acceso al valor local (la variable global está oculta detrás de la local, por

lo que no tenemos acceso a ella en este alcance local). Usar este nombre fuera del alcance local significa que nos referiremos a la variable global. **Sin embargo, esta no es la mejor práctica de programación y debemos evitar tales situaciones.** No es difícil adivinar que con un poco de falta de atención, el uso de este mecanismo puede conducir a situaciones no deseadas y probablemente a errores en el funcionamiento del programa.

El intérprete de JavaScript escanea el programa antes de ejecutarlo, buscando errores en su sintaxis, entre otras cosas. Hace una cosa más en esta ocasión. Busca todas las declaraciones de variables y las mueve al principio del rango en el que fueron declaradas (al principio del programa si son globales, al principio del bloque si es una declaración `let` local, o al principio de la función si es una declaración local `var`). Todo esto sucede, por supuesto, en la memoria del intérprete, y los cambios no son visibles en el código.

Hoisting, es un mecanismo bastante complejo y francamente bastante incoherente. Comprenderlo bien requiere la capacidad de usar libremente muchos elementos de JavaScript, que aún no hemos mencionado.

El **Hoisting** nos ayuda a declarar las variables hasta el inicio del codigo. Basicamente puede funcionar como la declaracion de funciones sin su implementación.

```
var height = 180;
console.log(height);  // -> 180
console.log(weight);  // -> undefined
var weight = 70;
console.log(weight);  // -> 70

//El interprete hace un escaneo de todas las variables y tiene conocimiento
//de que existen dichas variables (aunque aún no sean definidas)
//el intérprete ha movido la declaración al comienzo del rango
//(en este caso, el programa).
```

El hositing solo se refiere a la declaración, no a la inicialización. Entonces el valor 70, que le asignamos a la variable weight, permanece en la línea donde está la declaración original. El hoisting en JavaScript es un comportamiento especial del intérprete de JavaScript durante la fase de compilación. Consiste en elevar (o "levantar") las declaraciones de variables y funciones al inicio del ámbito en el que se encuentran, sin importar dónde se hayan escrito realmente en el código.

El hoisting permite que las declaraciones de variables y funciones se muevan virtualmente al principio de su ámbito, aunque en el código estén ubicadas después

de su uso. Esto significa que puedes utilizar una variable o una función antes de declararla explícitamente.

Nota: Sin embargo, es importante tener en cuenta que solo las declaraciones son elevadas, no las inicializaciones. Las asignaciones de valores a variables no se ven afectadas por el hoisting.

El hoisting funciona de manera diferente para las variables declaradas con `let` y `const` en comparación con las variables declaradas con `var`. Cuando se utilizan `let` y `const` para declarar variables, no se produce el hoisting en el sentido de que las variables no se elevan al principio del ámbito. En cambio, las declaraciones de `let` y `const` permanecen en su ubicación original en el código y solo están disponibles para su uso después de la línea en la que se declaran.

Mientras que las variables declaradas con `var` se ven afectadas por el hoisting y pueden ser accedidas antes de su declaración (con un valor `undefined`), las variables declaradas con `let` y `const` no se ven afectadas por el hoisting y arrojarán un error si se intenta acceder a ellas antes de su declaración.

Tipos de Datos y sus Conversiones

Distinguir los datos por sus tipos es uno de los rasgos característicos de cualquier lenguaje de programación. Cada tipo de dato está conectado con ciertas operaciones que podemos realizar sobre él. Por lo general, también existen métodos para convertir datos entre tipos seleccionados (por ejemplo, un número se puede convertir para que se guarde como una cadena).

En JavaScript, los tipos de datos se dividen en **primitivos** (o simples) y **complejos** (o compuestos). Entre los tipos primitivos podemos encontrar **números** y **cadenas de caracteres**, mientras que los tipos complejos incluyen, por ejemplo, **arreglos** y objetos.

La diferencia entre estos tipos de datos se encuentra precisamente en sus nombres. Los tipos primitivos, bueno, simplemente no son complejos. Si escribes datos de un tipo primitivo en una variable, se almacenará allí un valor particular. Este valor será atómico, es decir, no será posible extraer componentes de él. Los datos de tipos complejos, como un arreglo, constarán de muchos elementos de tipos primitivos (no complejos).

Los literales son una forma de anotar valores específicos (datos) en el código del programa. Los literales se encuentran en prácticamente todos los lenguajes de

programación, incluido JavaScript. Usamos literales en el capítulo anterior al inicializar variables.

```
let year = 1990;
console.log(year); // -> 1990
console.log(1991); // -> 1991
console.log("Alice"); // -> Alice
```

En este ejemplo, declaramos la variable year e inmediatamente la iniciamos con el valor 1990. Los dígitos 1990, escritos en el código en el lugar de inicialización de la variable, son un literal que representa un **número** . El valor 1990 se muestra en la consola utilizando la variable year. Luego mostramos en la consola el valor 1991 y "Alice", en ambos casos usando literales (que representan un **número** y una **cadena** respectivamente). En JavaScript, casi cada tipo de datos tiene su propio literal.

typeof: El operador `typeof` es unario (solo toma un argumento) y nos informa del tipo de datos indicados como un argumento dado. El argumento puede ser un literal o una variable; en este último caso, se nos informará sobre el tipo de datos almacenados en él. El operador typeof devuelve una cadena con uno de los valores fijos asignados a cada uno de los tipos. Todos los posibles valores de retorno del operador typeof son:

"undefined"
"object"
"boolean"
"number"
"bigint"
"string"
"symbol"
"function"

Su uso es el siguiente:

```
let year = 1990;
console.log(typeof year); // -> number
console.log(typeof 1991); // -> number

//Imprimira en consola "number"
```

Datos primitivos

En JavaScript, hay seis tipos de datos primitivos (o simples): **Boolean, Number, BigInt, String, Symbol** y **undefined**. Además, el valor primitivo **null** también se trata como un tipo separado. Un dato primitivo, como ya hemos dicho, es un tipo de dato cuyos valores son atómicos. Esto significa que el valor es un elemento indivisible.

Boolean: El Boolean (booleano) es un tipo de dato lógico. Solo puede tomar uno de dos valores: `true` o `false`. Se usa principalmente como una expresión condicional necesaria para decidir qué parte del código debe ejecutarse o cuánto tiempo debe repetirse algo

Number: Este es el tipo numérico principal en JavaScript que representa tanto números reales (por ejemplo, fracciones) como enteros. El formato en el que se almacenan los datos de este tipo en la memoria hace que los valores de este tipo sean a veces aproximados (especialmente, pero no únicamente, valores muy grandes o algunas fracciones). Los números en JavaScript generalmente se presentan en forma decimal, a la que estamos acostumbrados en la vida cotidiana. Sin embargo, si un literal que describe un número está precedido por un prefijo apropiado, podemos presentarlo en hexadecimal `(0xâ¦)`, octal `(0o...)` o binario `(0b...)`. También podemos escribir números en forma exponencial, por ejemplo, en lugar de `9000`, podemos escribir `9e3`, y en lugar de `0.00123`, podemos escribe `123e-5`. Probablemente ya estés familiarizado con los términos que usamos hace un momento, como representación decimal, hexadecimal o exponencial.

```
let a = 10; // decimal - default
let b = 0x10; // hexadecimal
let c = 0o10; // octal
let d = 0b10; // binary

console.log(a); // -> 10
console.log(b); // -> 16
console.log(c); // -> 8
console.log(d); // -> 2

let x = 9e3;
let y = 123e-5;
console.log(x); // -> 9000
console.log(y); // -> 0.00123
```

BigInt: El tipo de dato **BigInt** no se usa con demasiada frecuencia. Nos permite escribir números enteros de prácticamente cualquier longitud. Para casi todas las operaciones numéricas normales, el tipo **Number** es suficiente, pero de vez en cuando necesitamos un tipo que pueda manejar números enteros mucho más grandes. Podemos usar operaciones matemáticas en BigInts de la misma manera que en Numbers, pero hay una diferencia al dividir. Como BigInt es un tipo entero, el resultado de la división siempre se **redondeará hacia abajo** al número entero más cercano.

Strings: El tipo String representa una secuencia de caracteres que forman un fragmento de texto. Las operaciones comunes en los textos incluyen la concatenación, la extracción de la subcadena y la verificación de la longitud de la cadena. Las cadenas se utilizan mucho en la programación y más aún en el desarrollo web, ya que tanto HTML como gran parte del contenido de Internet es texto.

El uso más común de texto en el desarrollo web incluye:

- Enlaces y rutas a recursos.
- Tokens.
- Comprobación de formularios y entradas llenadas por el usuario.
- Generación de contenido dinámico.

Las **Cadenas**, como otros datos primitivos, son inmutables, por lo que cuando queremos cambiar incluso una letra en una cadena, en realidad, creamos una nueva cadena. Usamos comillas para indicar que un texto determinado debe tratarse como una cadena (es decir, tipo String). Los literales de cadena se pueden crear utilizando comillas simples o dobles, siempre que coincidan los caracteres de comillas inicial y final. Un mecanismo muy conveniente que se introdujo en JavaScript en 2015 es la **interpolación de cadenas**. Te permite tratar una cadena de caracteres como una plantilla, en la que puedes colocar valores en lugares seleccionados, como los que se toman de las variables. Tal literal se crea usando acentos graves en lugar de comillas. Los lugares donde se insertan los valores están marcados con corchetes precedidos por un signo de `$`.

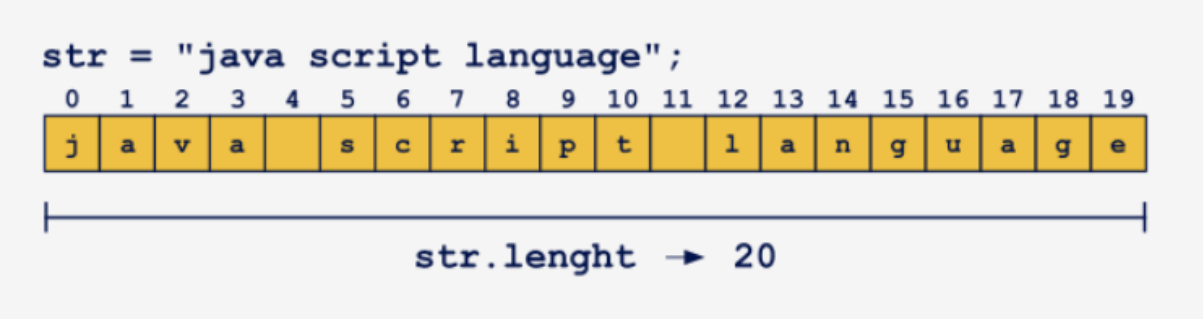
```
let country = "Malawi";
let continent = "Africa";

let sentence = ` ${country} se ubica en ${continent}.`;
console.log(sentence); // -> Malawi se ubica en Africa.
```

Concepto: ***Autoboxing.*** En JavaScript, el autoboxing es un concepto relacionado con el comportamiento automático de la conversión de tipos primitivos en objetos envoltorios (wrappers) cuando se accede a las propiedades o métodos de los tipos primitivos. Los tipos primitivos en JavaScript son `string`, `number`, y `boolean`. Cada uno de estos tipos primitivos tiene un objeto envoltorio asociado: `String`, `Number` y `Boolean`, respectivamente. Estos objetos envoltorios proporcionan propiedades y métodos adicionales que permiten trabajar con los valores primitivos como si fueran objetos.

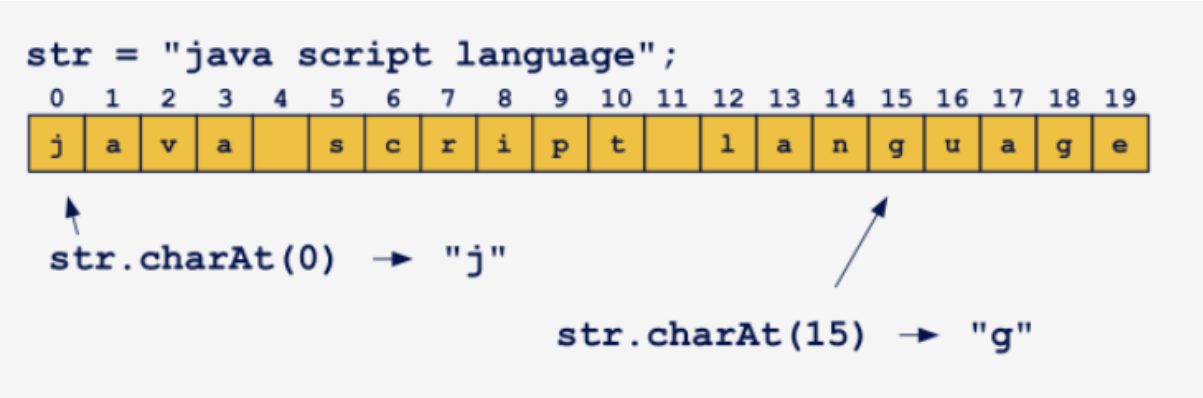
```
let numero = 42;
let textoNumero = numero.toString();

// Aquí, el valor primitivo 'numero' se convierte automáticamente en un
// objeto Number temporalmente para llamar al método 'toString()'.
// Esto permite que funcione como si 'numero' fuera un objeto Number con el
// método 'toString()'.
```

Los metodos más utilizados para los datos primitivos de tipo string son:

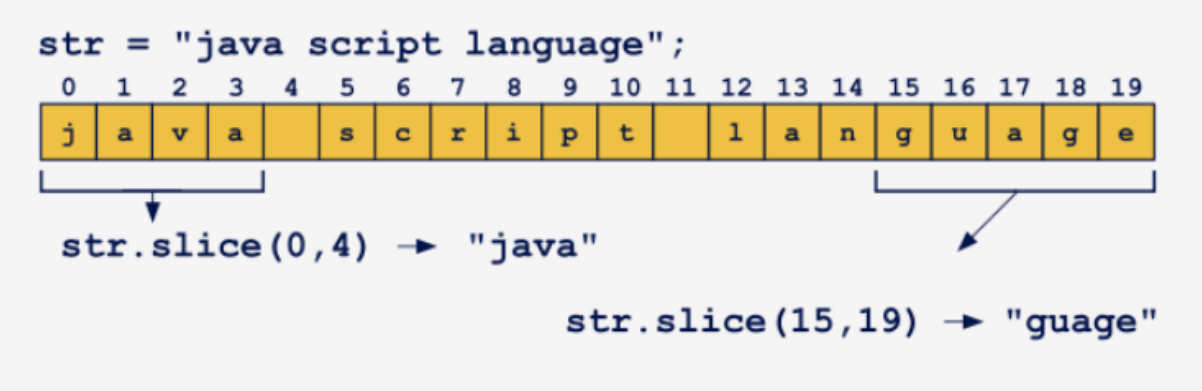
- `length`: propiedad que devuelve el número de caracteres en una cadena.



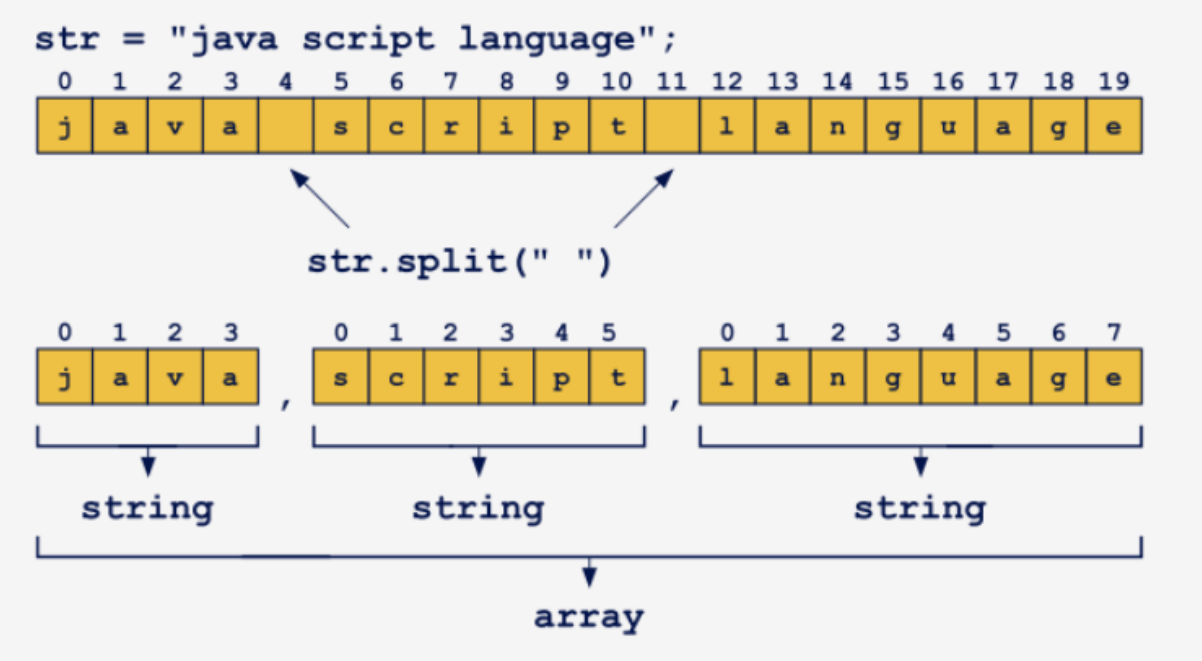
- `charAt(Index)`: método que devuelve el carácter en la posición "Index" en las cadenas (los índices comienzan desde 0).



• `slice(beginIndex, [opcional] endIndex)`: método que devuelve una nueva cadena que se crea a partir de los caracteres entre `beginIndex` (incluido) y `endIndex` (excluido); si se omite `endIndex`, entonces la nueva cadena es desde `beginIndex` hasta el final de la cadena.



• `split(separator, [opcional] limit)`: método que divide la cadena en subcadenas cada vez que se encuentra un separador en esa cadena y devuelve una arreglo de esas subcadenas (diremos algunas palabras sobre arreglos en un momento), mientras que un límite opcional `limit` limita el número de subcadenas añadidas a la lista.




```
let str = "java script language";

console.log(str.length); // -> 20
console.log('test'.length); // -> 4
```

```
console.log(str.charAt(0)); // -> 'j'
console.log('abc'.charAt(1)); // -> 'b'

console.log(str.slice(0, 4)); // -> 'java'
console.log('test'.slice(1, 3)); // -> 'es'

console.log(str.split(' ')); // -> ['java', 'script', 'language']
console.log('192.168.1.1'.split('.'));  // -> ['192', '168', '1', '1']
```

Undefined: El tipo undefined (indefinido) tiene un solo valor: `undefined`. Es el valor predeterminado que tienen todas las variables después de una declaración si no se les asigna ningún valor. También puedes asignar el valor `undefined` a cualquier variable, pero en general, esto debe evitarse, porque si necesitamos marcar una variable para que no tenga ningún valor significativo, debemos usar `null`.

Symbol: Los símbolos se utilizan para crear propiedades de objetos que sean únicas y no colisionen con otras propiedades.

Null: El valor `null` es bastante específico. El valor en sí es primitivo, mientras que el tipo al que pertenece no es un tipo primitivo, como Number o undefined. Esta es una categoría separada, asociada con tipos complejos, como objetos. El valor `null` se usa para indicar que la variable no contiene nada y, en la mayoría de los casos, es una variable que pretende contener valores de tipos complejos. En pocas palabras, podemos suponer que el valor `undefined` se asigna automáticamente a las variables no inicializadas, pero si queremos indicar explícitamente que la variable no contiene nada, le asignamos un valor `null`. Una advertencia importante para `null` es que cuando se verifica con el operador `typeof`, devolverá `"object"`

Conversiones de tipos de datos (cast)

A veces se requiere pasar de un tipo de dato a otro (como por ejemplo, pasar un number a string). Para ello, usamos funciones especificas.

```
const num = 42;

const strFromNum1 = String(num); //"42"
const strFromNum2 = String(8); //"8"
const strFromBool = String(true); //"true"
const numFromStr = Number("312"); //312
const boolFromNumber = Boolean(0); //false
```

Las conversiones son las más fáciles de entender, ya que intentan cambiar directamente el valor a una cadena, y esto se puede hacer para todos los tipos de datos primitivos. Así que no hay sorpresas allí.

Conversionces implicitas

Las conversiones también pueden ocurrir automáticamente y ocurren todo el tiempo.

```
const str1 = 42 + "1";
console.log(str1);        // -> 421
console.log(typeof str1); // -> string

const str2 = 42 - "1";
console.log(str2);        // -> 41
console.log(typeof str2); // -> number
```

La respuesta corta es que cuando intentamos realizar una suma cuando uno de los argumentos es una cadena, JavaScript también convertirá el resto de los argumentos en una cadena. Esto es lo que sucede con `str1` en el ejemplo. Sin embargo, la resta con una cadena no tiene mucho sentido, por lo que, en ese caso, JavaScript convierte todo a Number.

Datos complejos

Objeto: Los objetos tienen muchas aplicaciones en JavaScript. Una de las más básicas es utilizarlo como una estructura conocida en informática como registro. Un **registro** es una colección de campos con nombre. Cada campo tiene su propio nombre (o clave) y un valor asignado. En el caso de los objetos de JavaScript, estos campos suelen denominarse propiedades. Los registros, o en nuestro caso, los objetos, te permiten almacenar múltiples valores de diferentes tipos en un solo lugar. En JavaScript, hay algunas formas de crear objetos, pero la más fácil y rápida es usar el literal de llaves {}.

```
let testObj = {
    nr: 600,
    str: "texto"
};
console.log(typeof testObj); // -> object
```

Toma en cuenta que hemos creado objetos usando el mismo literal, pero al mismo tiempo hemos creado propiedades que son pares clave-valor. Las propiedades están separadas por comas. Posteriormente, se puede hacer referencia a una propiedad (campo) específica de un objeto con notación de puntos. Esta notación requiere que el nombre del objeto (un literal o el nombre de una variable que contiene el objeto) sea seguido por un punto, seguido por el nombre del campo (clave) nuevamente.

```
console.log(testObj.nr); // -> 600
console.log(testObj.str); // -> texto
```

Las propiedades de un objeto, como hemos indicado anteriormente, se ponen a disposición con un punto y un nombre clave. Podemos tanto leer como modificar el valor asociado a una clave en particular. Además, también podemos modificar todo el objeto añadiendo una nueva propiedad que antes no existía. También hacemos esto usando la notación de puntos: si durante un intento de modificar la propiedad, el intérprete no encuentra la clave que especificamos, la creará.

```
console.log(user1.name); // -> Calvin
console.log(user2.name); // -> Mateus

console.log(user1.age); // -> 66
user1.age = 67;
console.log(user1.age); // -> 67

console.log(user2.phone); // -> undefined
user2.phone = "904-399-7557"; // Este dato es el que se creara
console.log(user2.phone); // -> 904-399-7557
```

Para borrar una propiedad, basta con utilizar la palabra reservada delete

```
console.log(user2.phone); // -> 904-399-7557
delete user2.phone;
console.log(user2.phone); // -> undefined
```

Los objetos serán estructuras simples, formadas por pares de clave-valor.

Arreglos: Un **arreglo**, como un objeto, es un tipo de datos complejo que se puede usar para almacenar una colección de datos. Similar a un objeto, los datos almacenados (los valores) pueden ser de cualquier tipo. La diferencia entre estas

estructuras es que en un arreglo solo almacenamos valores, sin los nombres asociados (es decir, las claves). ¿cómo sabemos a qué elemento del arreglo nos referimos si no podemos señalarlo por su nombre? Lo sabemos porque los elementos del arreglo están ordenados (pero no necesariamente clasificados) y ocupan posiciones consecutivas y numeradas dentro de el. El número del campo donde se encuentra un valor particular en el arreglo se denomina índice o posición. El índice comienza desde 0.

La forma más sencilla de crear arreglos en JavaScript es usar corchetes (es un literal de arreglo). De esta forma, podemos crear tanto un arreglo vacío, en el que se insertarán los elementos más tarde, como un arreglo que contenga algunos elementos iniciales (que estarán separados por comas). Al referirnos a un elemento del arreglo en particular, usamos la notación de corchetes: después del nombre de la variable del arreglo, escribimos un corchete, en el que ponemos el índice del elemento que nos interesa.

```
let days = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"];
console.log(days[0]); // -> Sun
console.log(days[2]); // -> Tue
console.log(days[5]); // -> Fri

days[0] = "Sunday";
console.log(days[0]); // -> Sunday

let emptyArray = [];
console.log(emptyArray[0]); // -> undefined
```

¿Cómo podemos agregar un nuevo elemento a un arreglo existente, por ejemplo, uno vacío?

La forma más sencilla sería asignar un nuevo valor a una posición específica utilizando la notación de corchetes. Para el intérprete, no importa si ya hay algo en este índice o no. Simplemente coloca un nuevo valor allí. Lo interesante es que no tenemos que llenar el arreglo con elementos uno por uno; puedes dejar espacios vacíos en el arreglo.

```
let animals = [];
console.log(animals[0]); // -> undefined

animals[0] = "dog";
animals[2] = "cat";

console.log(animals[0]); // -> dog
console.log(animals[1]); // -> undefined
console.log(animals[2]); // -> cat
```

Podemos crear fácilmente un arreglo que contenga elementos de diferentes tipos.

```
let values = ["Test", 7, 12.3, false];
```

Es interesante es el hecho de que también podemos almacenar arreglos como elementos de un arreglo, y podemos acceder a los elementos de este arreglo anidado usando múltiples corchetes.

```
let names = [["Olivia", "Emma", "Mia", "Sofia"], ["William", "James", "Daniel"]];
console.log(names[0]); // -> ["Olivia", "Emma", "Mia", "Sofia"]
console.log(names[0][1]); // -> Emma
console.log(names[1][1]); // -> James

let femaleNames = names[0];
console.log(femaleNames[0]); // -> Olivia
console.log(femaleNames[2]); // -> Mia
```

El ejemplo muestra una declaración de un arreglo que contiene otros dos arreglos como sus componentes. Toma en cuenta que los arreglos internos no tienen que tener la misma longitud (en muchos otros lenguajes de programación, esto es obligatorio).

Podemos crear arreglos de objetos

```
let users =[
    {
        name: "Calvin",
        surname: "Hart",
        age: 66,
        email: "CalvinMHart@teleworm.us"
    },
    {
        name: "Mateus",
        surname: "Pinto",
        age: 21,
        email: "MateusPinto@dayrep.com"
    }
];

console.log(users[0].name); // -> Calvin
console.log(users[1].age); // -> 21
```

Para agregar un nuevo dato, podemos hacer lo siguiente

```
users[2] = {
    name: "Irene",
    surname: "Purnell",
    age: 32,
    email: "IreneHPurnell@rhyta.com"

}

console.log(users[0].name); // -> Calvin
console.log(users[1].name); // -> Mateus
console.log(users[2].name); // -> Irene
```

En JavaScript, todo excepto los datos primitivos son objetos. Los **arreglos** también se tratan como un tipo especial de objeto. El operador `typeof` no distingue entre tipos de objetos (o más precisamente, clases), por lo que nos informa que la variable days contiene un objeto. Si queremos asegurarnos de que la variable contiene un arreglo, podemos hacerlo usando el operador `instanceof`, entre otros. Está estrechamente relacionado con la programación orientada a objetos, de la que no hablaremos en este curso, pero por el momento solo necesitamos saber cómo usarlo en esta situación específica.

Hay muchos métodos muy útiles que nos ayudan a trabajar con arreglos, como combinar arreglos, cortar elementos, ordenar o filtrar. Algunos metodos que nos facilitan los arreglos son:

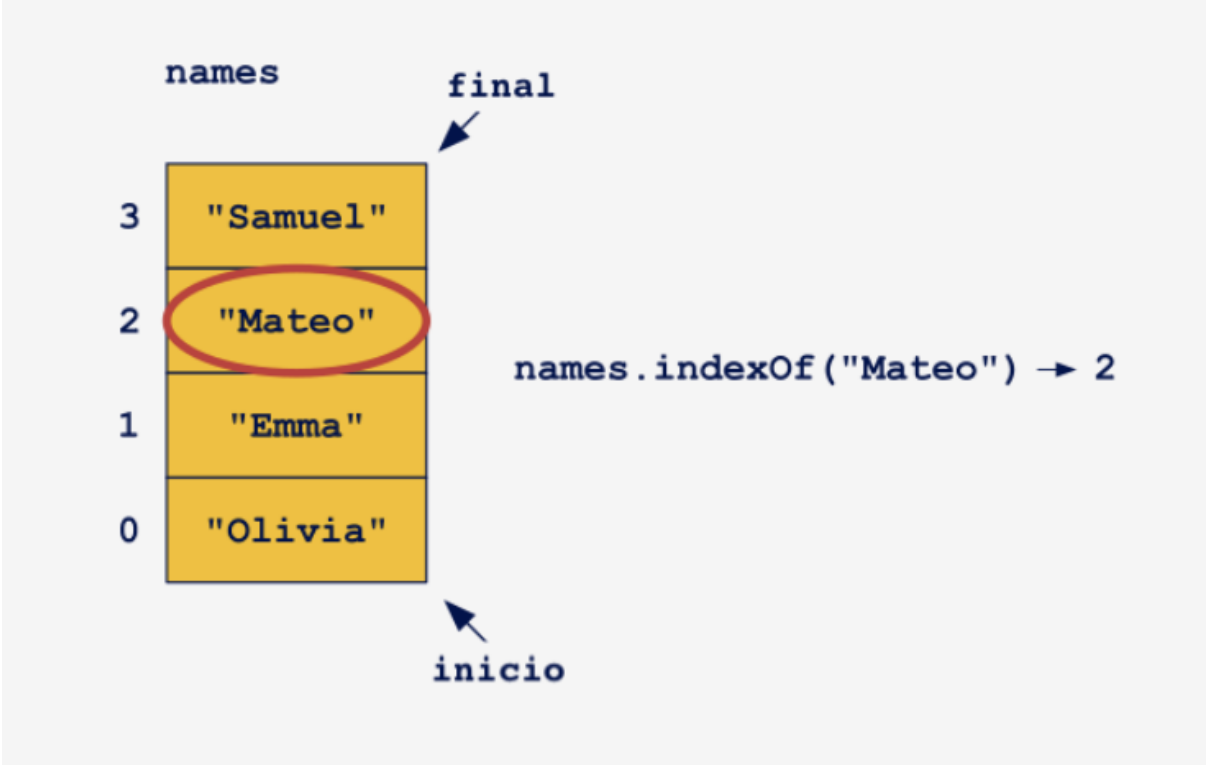
- lenght: La propiedad length se utiliza para obtener información sobre la longitud (la cantidad de elementos) del arreglo (incluidas las posiciones vacías entre los elementos existentes).

```
let names  = ["Olivia", "Emma", "Mateo", "Samuel"];
console.log(names.length); // -> 4

names[5] = "Amelia";
console.log(names.length); // -> 6

console.log(names); // -> ["Olivia", "Emma", "Mateo", "Samuel", undefined, "Amelia"]
console.log(names[3]); // -> Samuel
console.log(names[4]); // -> undefined
console.log(names[5]); // -> Amelia
```

- indexOf: El método `indexOf` se utiliza para buscar en el arreglo y localizar un valor dado. Si se encuentra el valor (el elemento está en el arreglo), se devolverá su índice (posición). El método devuelve -1 si no se encuentra el

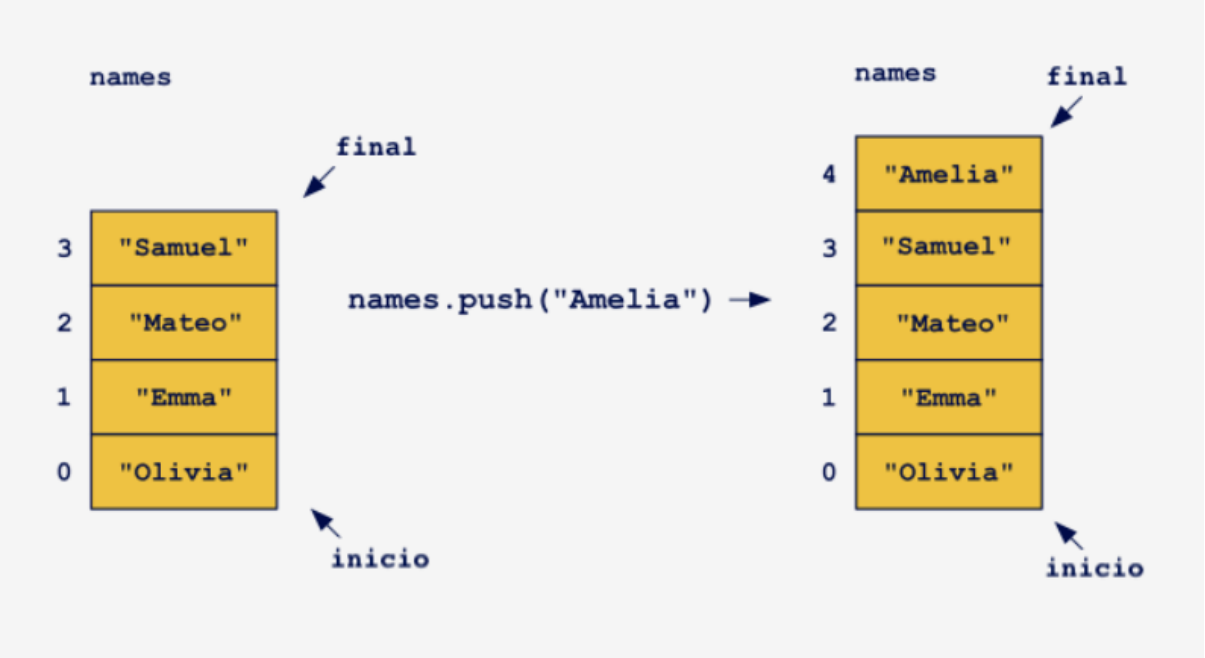
elemento. Si hay más de un elemento con el mismo valor en el arreglo, se devuelve el índice del primer elemento encontrado.

```
let names = ["Olivia", "Emma", "Mateo", "Samuel"];
console.log(names.indexOf("Mateo")); // -> 2
console.log(names.indexOf("Victor")); // -> -1
```



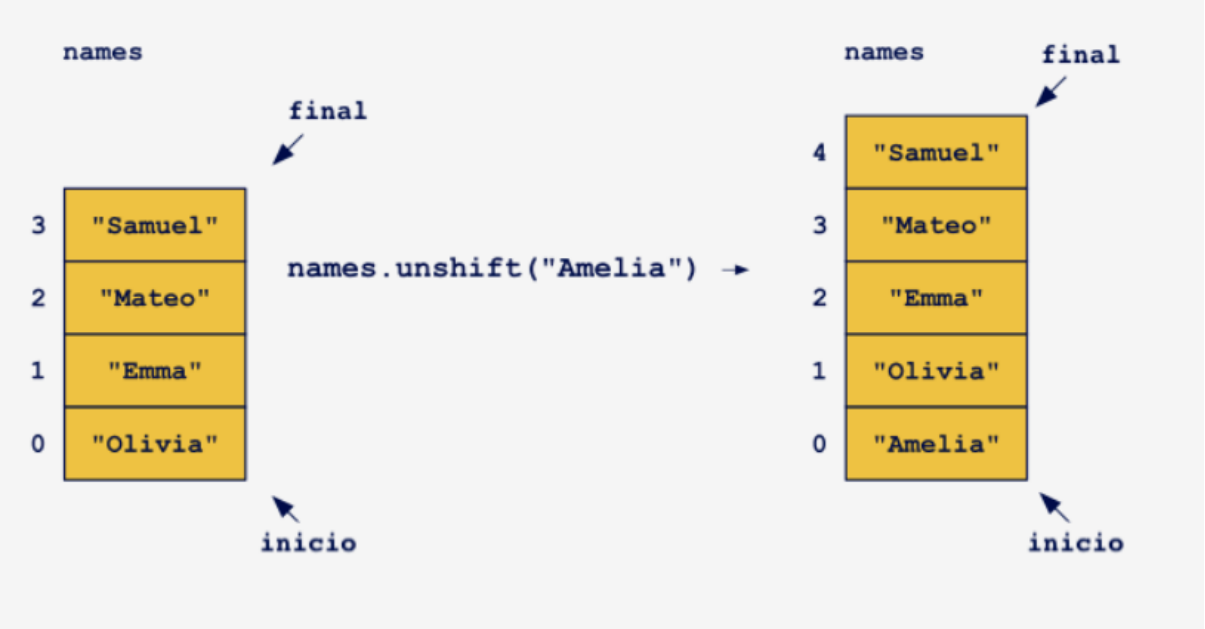
- Push: El método `push` coloca el elemento dado como su argumento al final del arreglo. La longitud del arreglo aumenta en 1 y el nuevo elemento se inserta a la derecha (tiene el índice más grande de todos los elementos).

```
let names = ["Olivia", "Emma", "Mateo", "Samuel"];
console.log(names.length); // -> 4

names.push("Amelia");
console.log(names.length); // -> 5
console.log(names); // - > ["Olivia", "Emma", "Mateo","Samuel", "Amelia"]
```



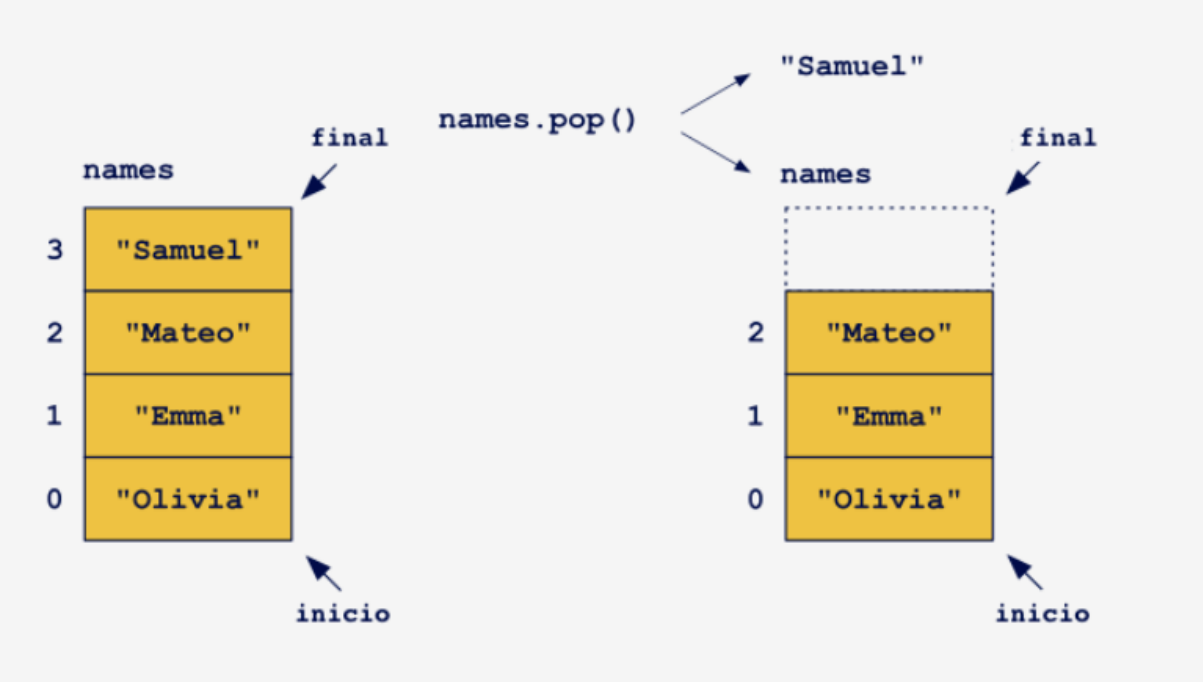
- unshif: El método `unshift` funciona de manera similar a push, con la diferencia de que se agrega un nuevo elemento al inicio del arreglo. La longitud de arreglo aumenta en 1, todos los elementos antiguos se mueven hacia la derecha y el nuevo elemento se coloca en el espacio vacío que se ha creado al inicio del arreglo. El índice del nuevo elemento es 0.

```
let names = ["Olivia", "Emma", "Mateo", "Samuel"];
console.log(names);
console.log(names.indexOf("Mateo")); // -> 2
console.log(names.indexOf("Victor")); // -> -1 -> No existe
names.unshift("Amelia");
console.log(names.indexOf("Amelia")); // -> 0
console.log(names);
```



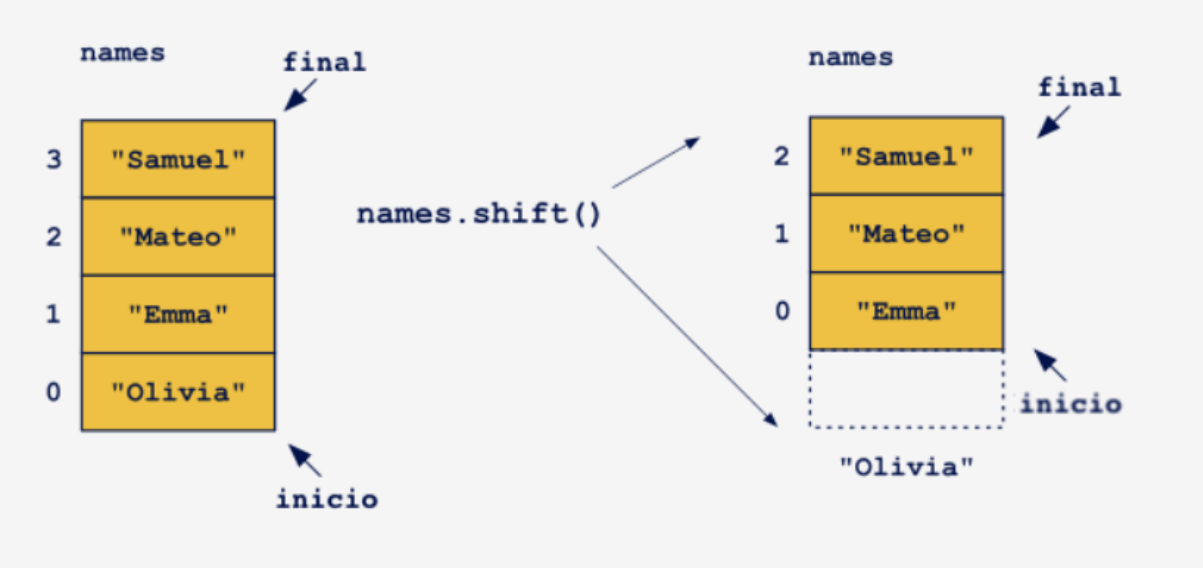
- pop: El método `pop` te permite eliminar el último elemento del arreglo. Como resultado de su ejecución, se devuelve el elemento con mayor índice, mientras que al mismo tiempo se elimina del arreglo original. La longitud del arreglo obviamente se reduce en 1.

```
let names= ["Olivia", "Emma", "Mateo", "Samuel"];
console.log(names.length); // -> 4

let name = names.pop();
console.log(names.length); // -> 3
console.log(name); // -> Samuel
console.log(names); // -> ["Olivia", "Emma", "Mateo"]
```



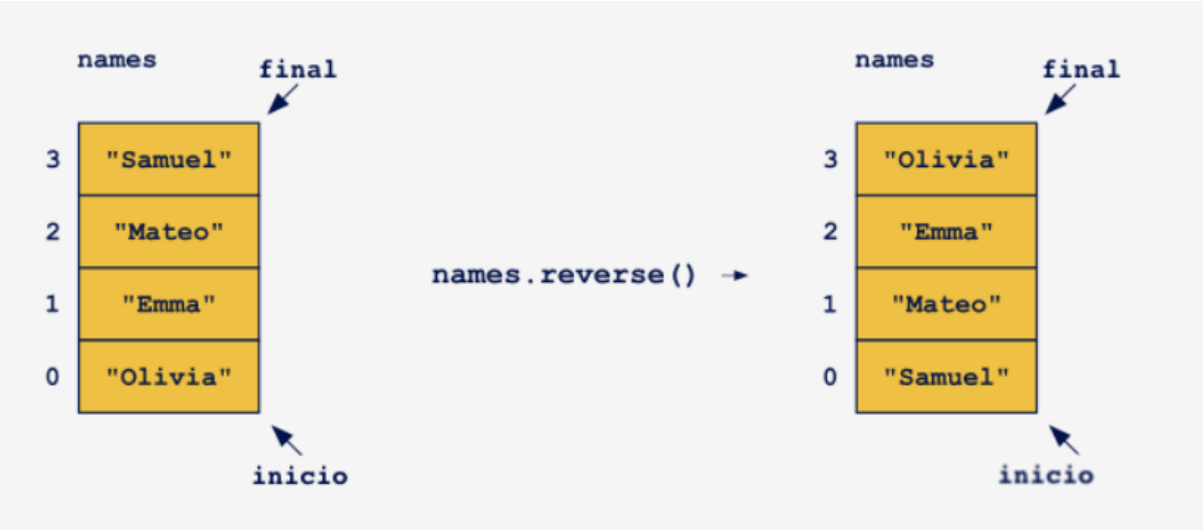
- shift: El método `shift` funciona de manera similar a pop, solo que esta vez eliminamos el elemento del inicio del arreglo (con el índice 0). El método devuelve el elemento eliminado y todos los demás elementos se desplazan hacia la izquierda, llenando el espacio vacío. La longitud del arreglo original se reduce en 1.

```
let names = ["Olivia", "Emma", "Mateo", "Samuel"];
console.log(names.length); // -> 4

let name = names.shift();
console.log(names.length); // -> 3
console.log(name); // -> Olivia
console.log(names); // -> ["Emma", "Mateo", "Samuel"]
```



- reverse: El método `reverse` invierte el orden de los elementos de un arreglo. El método devuelve un arreglo con los elementos en orden inverso.

```
let names = ["Olivia", "Emma", "Mateo", "Samuel"];

names.reverse();
console.log(names); // -> ["Samuel", "Mateo", "Emma", "Olivia"]
```



- slice: El método `slice` te permite crear un nuevo arreglo a partir de elementos seleccionados del arreglo original. Llamar al método no afecta el arreglo original. El método toma uno o dos valores enteros como argumentos.

  Las combinaciones básicas son:

- Un argumento mayor que cero: se copian todos los elementos del índice dado como argumento hasta el final del arreglo.
- Dos argumentos mayores que cero: se copia el elemento del índice especificado como primer argumento al elemento especificado como segundo argumento.
- Dos argumentos, el primero positivo, el segundo negativo: se copian todos los elementos desde el índice especificado hasta el final del arreglo, excepto el número especificado de los últimos elementos (por ejemplo, el argumento -3 significa que no copiamos los últimos tres elementos).
- Un argumento negativo: el número especificado de los últimos elementos se copian al final del arreglo (por ejemplo, -2 significa que se copian los dos últimos elementos).

```
let names = ["Olivia", "Emma", "Mateo", "Samuel"];

let n1 = names.slice(2);
console.log(n1); // -> ["Mateo", "Samuel"]

let n2 = names.slice(1,3);
console.log(n2); // -> ["Emma", "Mateo"]

let n3 = names.slice(0, -1);
console.log(n3); // -> ["Olivia", "Emma", "Mateo"]

let n4 = names.slice(-1);
console.log(n4); // -> ["Samuel"]

console.log(names); // -> ["Olivia", "Emma", "Mateo","Samuel"]
```

- concat: El método `concat` crea un nuevo arreglo adjuntando elementos del arreglo dado como argumento a los elementos originales del arreglo. El método no cambia ni el arreglo original ni el arreglo especificado como argumento.

```
let names = ["Olivia", "Emma", "Mateo", "Samuel"];
let otherNames = ["William", "Jane"];
let allNames = names.concat( otherNames);

console.log(names); // -> ["Olivia", "Emma", "Mateo", "Samuel"]
console.log(otherNames); // -> ["William", "Jane"]
console.log(allNames); // -> ["Olivia", "Emma", "Mateo","Samuel", "William", "Jane"]
```