Javascript Arrays, Object & Functions

Agenda

- Creating & Using Arrays
- Creating & Using Objects
- Creating & Using Functions

Creating & Using Arrays

- Arrays are a type of object that are ordered by the index of each item it contains.

- The index starts at zero and extends to however many items have been added, which is a property of the array known as the "length" of the array.

- Similar to objects, an array can be created with the array constructor or the shorthand syntax known as array literal.

Creating Arrays...

Array Constructor

```
var foo = new Array;
```

Array Literal

```
var bar = [];
```

Creating Arrays with Dimensions

Array Constructor

```
var foo = new Array(100);
```

Array Literal

```
var bar = [];
```

Using Arrays...

- Insertion into arrays can be via:
  - [] notation (like Java)
  - Using ‘push’ which appends to the end of the array

```
var foo = [];

foo.push("a");
foo.push("b");

alert( foo[ 0 ] ); // => a
alert( foo[ 1 ] ); // => b

alert( foo.length ); // => 2

foo.pop();

alert( foo[ 0 ] ); // => a
alert( foo[ 1 ] ); // => undefined

alert( foo.length ); // => 1
```

Definition of an Array

- Arrays are zero-indexed, ordered lists of values.

- They are a handy way to store a set of related items of the same type (such as strings), though in reality, an array can include multiple types of items, including other arrays.

- To create an array, either use the object constructor or the literal declaration, by assigning the variable a list of values after the declaration

```
// A simple array with constructor
var myArray1 = new Array( "hello", "world" );
// literal declaration, the preferred way
var myArray2 = [ "hello", "world" ];
```

Push

- If the values are unknown, it is also possible to declare an empty Array, and add elements either through functions or through accessing by index:

- 'push' is a function that adds an element on the end of the array and expands the array respectively. You also can directly add items by index. Missing indices will be filled with 'undefined'.

```
// Creating empty arrays and adding
values
var myArray = [];

// adds "hello" on index 0
myArray.push("hello");

// adds "world" on index 1
myArray.push("world");

// adds "!" on index 2
myArray[ 2 ] = "!";
```

Missing Indices

- Missing indices will be filled with 'undefined'.

- If the size of the array is unknown, 'push' is far more safe.

```
// Leaving indices
var myArray = [];

myArray[ 0 ] = "hello";
myArray[ 1 ] = "world";
myArray[ 3 ] = "!";

console.log( myArray );
// [ "hello", "world", undefined, "!" ];
```

- You can both access and assign values to array items with the index

```
// Accessing array items by index
var myArray = [ "hello", "world", "!" ];

console.log( myArray[2] ); // "!"
```

Array Methods and Properties

length

- The .length property is used to determine the amount of items in an array.

```
// Length of an array
var myArray = [ "hello", "world", "!" ];

console.log( myArray.length ); // 3
```

- You will need the .length property for looping through an array:

```
// For loops and arrays - a classic
var myArray = [ "hello", "world", "!" ];

for ( let i = 0; i < myArray.length; i = i + 1 )
{
  console.log( myArray[i] );
}
```

- Except when using for/in loops:

```
// For loops and arrays - alternate method
var myArray = [ "hello", "world", "!" ];

for ( let i in myArray )
{
  console.log( myArray[ i ] );
}
```

concat

- Concatenate two or more arrays with .concat:

```
// Concatenating Arrays
var myArray = [ 2, 3, 4 ];
var myOtherArray = [ 5, 6, 7 ];

// [ 2, 3, 4, 5, 6, 7 ]
var wholeArray = myArray.concat( myOtherArray );
```

join

- .join creates a string representation of the array. Its parameter is a string that works as a separator between elements (default separator is a comma):

```
// Joining elements
var myArray = [ "hello", "world", "!" ];

console.log( myArray.join(" ") );  // "hello world !";
console.log( myArray.join() );     // "hello,world,!"
console.log( myArray.join("") );   // "helloworld!"
console.log( myArray.join("!!") ); // "hello!!world!!!";
```

pop

- .pop removes the last element of an array. It is the opposite method of .push:

```
// pushing and popping
var myArray = [];

myArray.push( 0 ); // [ 0 ]
myArray.push( 2 ); // [ 0 , 2 ]
myArray.push( 7 ); // [ 0 , 2 , 7 ]
myArray.pop();     // [ 0 , 2 ]
```

reverse

- The elements of the array are in reverse order after calling this method:

```
// reverse
var myArray = [ "world" , "hello" ];

// [ "hello", "world" ]
myArray.reverse();
```

shift

- Removes the first element of an array.

- With .pop and .shift, you can simulate a queue:

```
// queue with shift() and pop()
var myArray = [];

myArray.push( 0 ); // [ 0 ]
myArray.push( 2 ); // [ 0 , 2 ]
myArray.push( 7 ); // [ 0 , 2 , 7 ]
myArray.shift();   // [ 2 , 7 ]
```

forEach

- In modern browsers it is possible to traverse through arrays with a .forEach method, where you pass a function that is called for each element in the array.
- The function takes up to three arguments:
  - Element - The element itself.
  - Index - The index of this element in the array.
  - Array - The array itself.
- All of these are optional, but you will need at least the 'element' parameter in most cases

```
// native forEach
function printElement( elem ) {
  console.log( elem );
}


myArray = [ 1, 2, 3, 4, 5 ];

// prints all elements to the console
myArray.forEach( printElement );
```

Element & Index Parameters

```
function printElementAndIndex( elem, index ) {
  console.log( "Index " + index + ": " + elem );
}


// prints "Index 0: 1" "Index 1: 2" "Index 2: 3" ...
myArray.forEach( printElementAndIndex );
```

Creating & Using Objects

- The simplest way to create an object is either through:
    - the *object constructor*
    - the shorthand syntax known as *object literal.*
- Objects are *unordered key/value pairs*.
    - The *key* is formally known as a property and the value can be any valid JavaScript type, even another object.
- To create or access a property on an object, we use
    - *dot notation*
    - *bracket notation*

Creating Objects ...

Object Constructor

```
let person1 = new Object;

person1.firstName = "John";
person1.lastName = "Doe";
```

Object Literal

```
let person2 = {
  firstName: "Jane",
  lastName: "Doe"
};
```

Using Objects...

Dot Notation

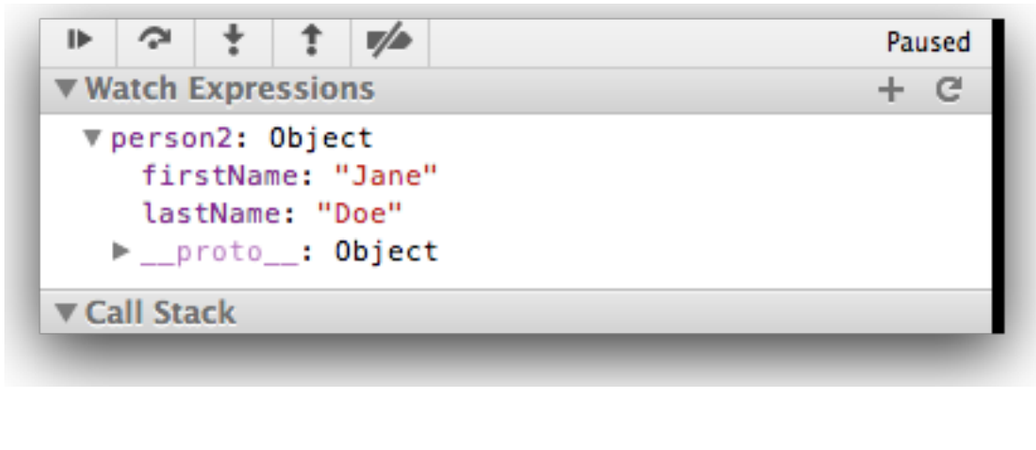
```
person1.firstName = "John";
person1.lastName = "Doe";
```

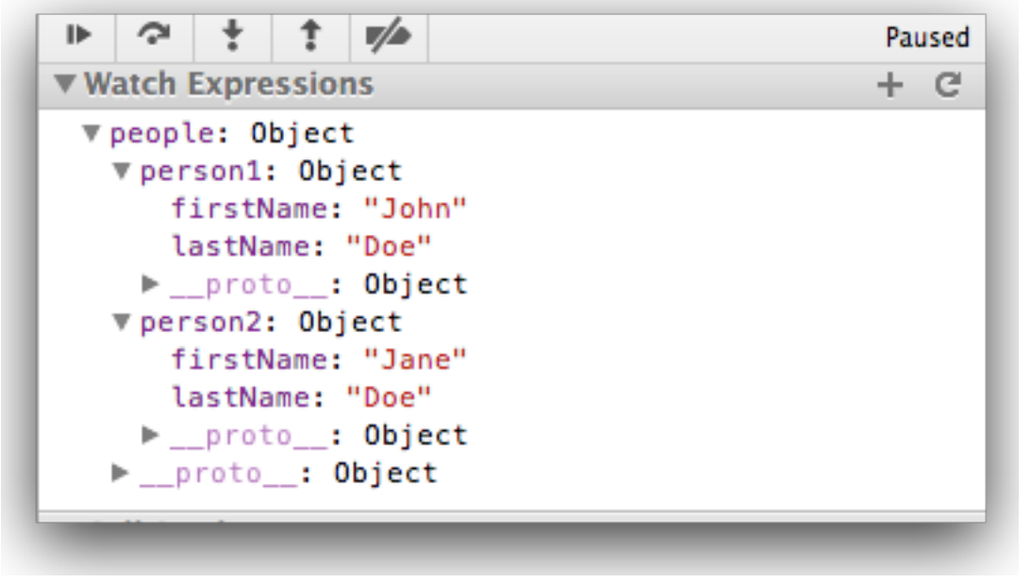
Bracket Notation

```
person['firstName'] = "Mary";
person['lastName'] = "Smith";
```

Tracing Objects

- Object structure and contents can be explored in detail in Chrome Developer Tools





Object Literals

- Objects contain one or more key-value pairs.
- The key portion can be any string.
- The value portion can be any type of value: a number, a string, an array, a function, or even another object.
- When one of these values is a function, it's called a method of the object. Otherwise, they are called properties.

```
// Creating an object literal
var myObject = {
  sayHello : function() {
    console.log("hello");
  },
  myName : "Rebecca"
};

myObject.sayHello(); // "hello"
console.log(myObject.myName); // "Rebecca"
```

Object Literals

- When creating object literals, note that the key portion of each key-value pair can be written as any valid JavaScript identifier, a string (wrapped in quotes), or a number:

```
// test
const myObject = {
  validIdentifier : 123,
  "some string"   : 456,
  99999           : 789
};
```

Creating Functions

- Functions contain blocks of code that need to be executed repeatedly.

- Functions can take zero or more arguments, and can optionally return a value.

- Functions can be created in a variety of way

```
// Function Declaration
function foo() {
  /* do something */
}
```

```
// Named Function Expression
var foo = function() {
  /* do something */
}
```

Using Functions

```
// A simple function
var greet = function(person, greeting) {
  const text = greeting + ", " + person;
  console.log(text);
};

greet("Rebecca", "Hello");
```

```
// A function that returns a value
var greet = function(person, greeting) {
  const text = greeting + ", " + person;
  return text;
};

console.log(greet("Rebecca", "hello")); // "hello, Rebecca"
```

Functions creating Functions

- the greet function returns a function!
- This functions is then called.

```
// A function that returns another function
var greet = function(person, greeting) {
  const text = greeting + ", " + person;
  return function() {
    console.log(text);
  };
};

var greeting = greet("Rebecca", "Hello");
greeting();
```

Immediately-Invoked Function Expression (IIFE)

- A common pattern in JavaScript is the immediately-invoked function expression.

- This pattern creates a function expression and then immediately executes the function.

- This pattern is useful for cases where you want to avoid polluting the global namespace with code — no variables declared inside of the function are visible outside of it.

```
// An immediately-invoked function expression
(function() {

  let foo = "Hello world";

})();

console.log( foo );   // undefined!
```

Functions as Arguments

- In JavaScript, functions are "first-class citizens" — they can be assigned to variables or passed to other functions as arguments.

- Challenging and difficult to read code!

```
// Passing an anonymous function as an argument
var myFn = function(fn) {
  var result = fn();
  console.log(result);
};

// logs "hello world"
myFn(function() {
  return "hello world";
});
```

Scope

- "Scope" refers to the variables that are available to a piece of code at a given time.

- A lack of understanding of scope can lead to frustrating debugging experiences.

- When a variable is declared inside of a function, it is only available to code inside of that function — code outside of that function cannot access the variable.

- On the other hand, functions defined inside that function will have access to to the declared variable.

More Scope...

- Furthermore, variables that are declared inside a function without the var/const/let keyword are not local to the function — JavaScript will traverse the scope chain all the way up to the global scope to find where the variable was previously defined.

- If the variable wasn't previously defined, it will be defined in the global scope, which can have unexpected consequences.

Scope Example 1

```
// Functions have access to variables defined in the same scope
var foo = “hello";
var sayHello = function() {
  console.log(foo);
};

sayHello(); // "hello"
console.log(foo); // "hello"
```

Scope Example 2

```
// Code outside the scope in which a variable was defined does not have access
// to the variable
var sayHello = function() {
  const foo = "hello";
  console.log(foo);
};

sayHello(); // hello

console.log(foo); // undefined
```

Scope Example 3

```
// Variables with the same name can exist in different scopes with different
// values
const foo = "world";

var sayHello = function() {
  const foo = "hello";
  console.log(foo);
};

sayHello(); // logs "hello"
console.log(foo); // logs "world"
```