**Global Object**

When javascript interpreter starts it creates a new global object and gives it an initial set of properties that define:

* global properties like **undefined**, **Infinity**, and **NaN**.
* global functions like **isNaN()**, **parseInt()**, and **eval()**.
* constructor functions like **Date()**, **RegExp()**, **String()**, **Object()**, and **Array()**.
* global objects like **Math** and **JSON**.

In client-side JavaScript, the Window object serves as the global object for all JavaScript code contained in the browser window it represents.

**Objects**

Objects in JavaScript are class-free. There is no constraint on the names of new prop- erties or on the values of properties. Objects are useful for collecting and organizing data. Objects can contain other objects, so they can easily represent tree or graph structures.

**Object Literals**

Object literals provide a very convenient notation for creating new object values.

**Retrieval**

There is 2 ways to access to the values of the properties of a object.

* string expression in a [ ] suf-fix.
* . notation

**Reference**

Objects are passed around by reference. They are never copied.

**Prototype**

Every JavaScript object has a second JavaScript object (or null, but this is rare) associated with it. This second object is known as a prototype, and the first object inherits properties from the prototype.

**Object.prototype** is one of the rare objects that has no prototype: it does not inherit any properties. Other prototype objects are normal objects that do have a prototype.

**Reflection**

It is easy to inspect an object to determine what properties it has by attempting to retrieve the properties and examining the values obtained.

**Delete**

The delete operator can be used to remove a property from an object. It will remove a property from the object if it has one. It will not touch any of the objects in the proto- type linkage.

**Variable Scope**

The scope of a variable is the region of your program source code in which it is defined. A global variable has global scope; it is defined everywhere in your JavaScript code. On the other hand, variables declared within a function are defined only within the body of the function. They are local variables and have local scope. Function parameters also count as local variables and are defined only within the body of the function.

**Functions**

Functions in JavaScript are objects.

Since functions are objects, they can be used like any other value. Functions can be stored in variables, objects, and arrays. Functions can be passed as arguments to functions, and functions can be returned from functions. Also, since functions are objects, functions can have methods.

* Method Invocation Pattern
* Function Invocation Pattern
* Constructor Invocation Pattern
* Apply Invocation Pattern

**Recursion**

A recursive function is a function that calls itself, either directly or indirectly. Recur- sion is a powerful programming technique in which a problem is divided into a set of similar subproblems, each solved with a trivial solution.

**Closure**

A ***closure*** is the combination of a function and the lexical environment within which that function was declared.

**Callbacks**

A ***callback*** is a function that is to be executed after another function has finished executing.

Callbacks are a way to make sure certain code doesn’t execute until other code has already finished execution.

**IIFE (Immediately Invoked Function Expression)**

is a [JavaScript](https://developer.mozilla.org/en-US/docs/Glossary/JavaScript) [function](https://developer.mozilla.org/en-US/docs/Glossary/function) that runs as soon as it is defined.

IIFEs are very useful because they don't pollute the global object, and they are a simple way to isolate variables declarations.

**Event delegation**

Event delegation allows you to avoid adding event listeners to specific nodes, instead the event listener is added to one parent.  That event listener analyzes bubbled events to find a match on child elements.

**Promises**

Promises are one way to deal with asynchronous code in JavaScript, without writing too many callbacks in your code.

A promise has 3 states.

1. Pending
2. Fulfilled
3. Rejected

**Chaining promises**

A promise can be returned to another promise, creating a chain of promises.

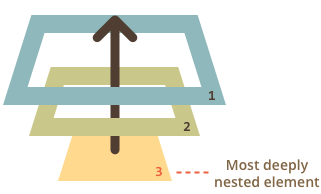
**Orchestrating promises**

If you need to synchronize different promises, **Promise.all()** helps you define a list of promises, and execute something when they are all resolved.

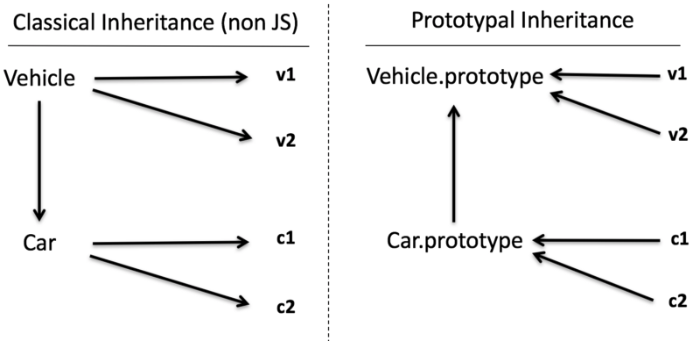
**Event bubbling**

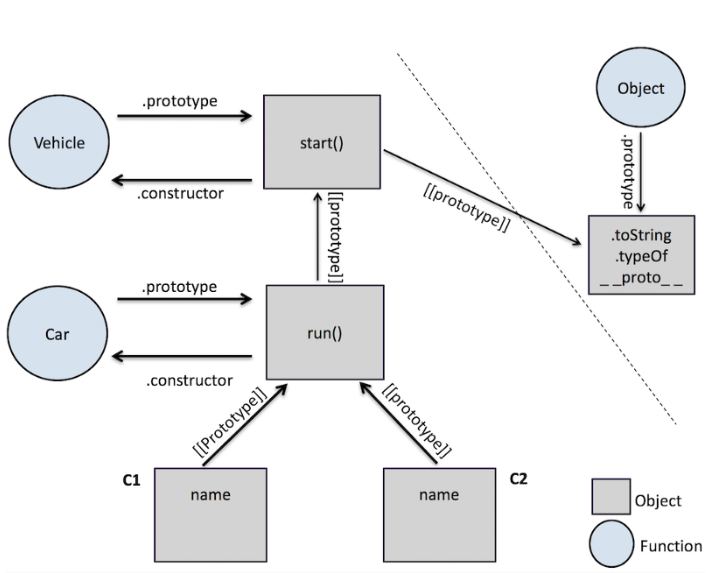
The bubbling principle is simple.

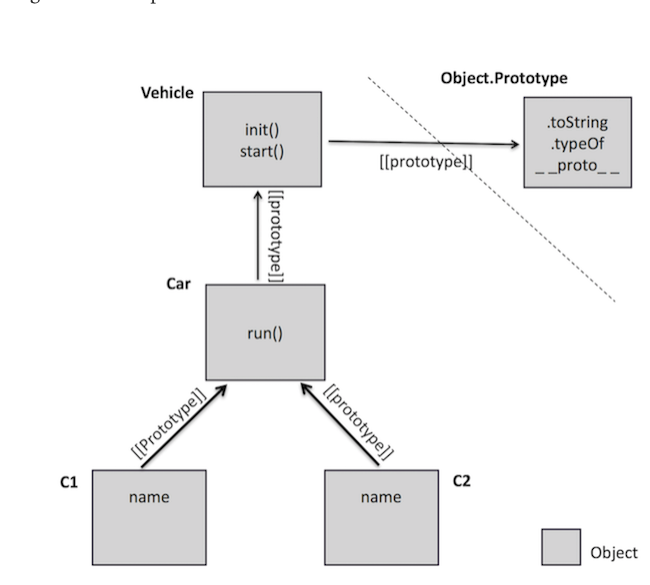
When an event happens on an element, it first runs the handlers on it, then on its parent, then all the way up on other ancestors.



**Inheritance**







**Throttle vs debounce**

To ***throttle***a function means to ensure that the function is called at most once in a specified time period (for instance, once every 10 seconds).

This means throttling will prevent a function from running if it has run **“recently”**.

***Debounced*** *function*will ignore all calls to it until the calls have stopped for a specified time period. Only then will it call the original function. For instance, if we specify the time as two seconds, and the debounced function is called 10 times with an interval of one second between each call, the function will not call the original function until two seconds after the last (tenth) call.

Use **debounce** when you want your function to postpone its next execution until after X milliseconds have elapsed since the last time it was invoked. Use **throttle** when you need to ensure that events fire at given intervals.