# JavaScript. The Core: 2nd Edition

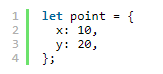
## [Object](http://dmitrysoshnikov.com/ecmascript/javascript-the-core-2nd-edition/#object)

ECMAScript is an object-oriented programming language with the prototype-based organization, having the concept of an object as its core abstraction.

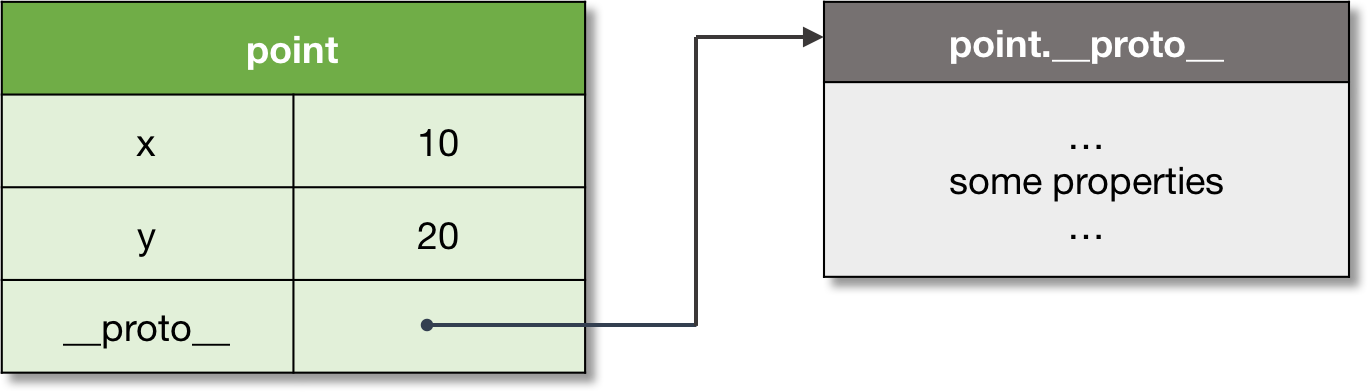
**Def. 1: Object:** An object is a collection of properties, and has a single prototype object. The prototype may be either an object or the null value.

Let’s take a basic example of an object. A prototype of an object is referenced by the internal [[Prototype]] property, which to user-level code is exposed via the \_\_proto\_\_ property.

For the code:



we have the structure with two explicit own properties and one implicit \_\_proto\_\_property, which is the reference to the prototype of point:



*Figure 1. A basic object with a prototype.*

**Note:** objects may store also symbols

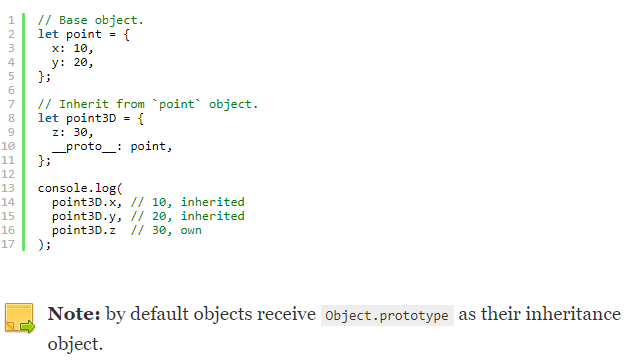
The prototype objects are used to implement inheritance with the mechanism of dynamic dispatch. Let’s consider the prototype chain concept to see this mechanism in detail.

## [Prototype](http://dmitrysoshnikov.com/ecmascript/javascript-the-core-2nd-edition/#prototype)

Every object, when is created, receives its prototype. If the prototype is not set explicitly, objects receive default prototype as their inheritance object.

**Def. 2: Prototype:** A prototype is a delegation object used to implement prototype-based inheritance.

The prototype can be set explicitly via either the \_\_proto\_\_ property, or Object.create method:



Any object can be used as a prototype of another object, and the prototype itself can have its own prototype. If a prototype has a non-null reference to its prototype, and so on, it is called the prototype chain.

**Def. 3: Prototype chain:** A prototype chain is a finite chain of objects used to implement inheritance and shared properties.



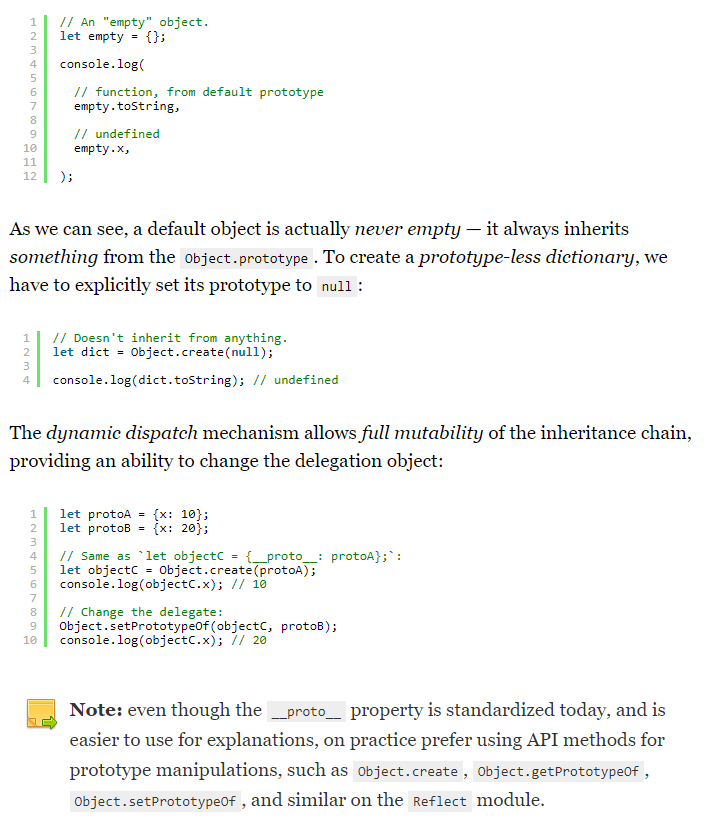
The rule is very simple: if a property is not found in the object itself, there is an attempt to resolve it in the prototype; in the prototype of the prototype, etc. — until the whole prototype chain is considered.

Technically this mechanism is known as dynamic dispatch or delegation.

**Def. 4: Delegation:** a mechanism used to resolve a property in the inheritance chain. The process happens at runtime, hence is also called **dynamic dispatch**.

**Note:** in contrast with static dispatch when references are resolved at compile time, dynamic dispatch resolves the references at runtime.

And if a property eventually is not found in the prototype chain, the undefined value is returned:



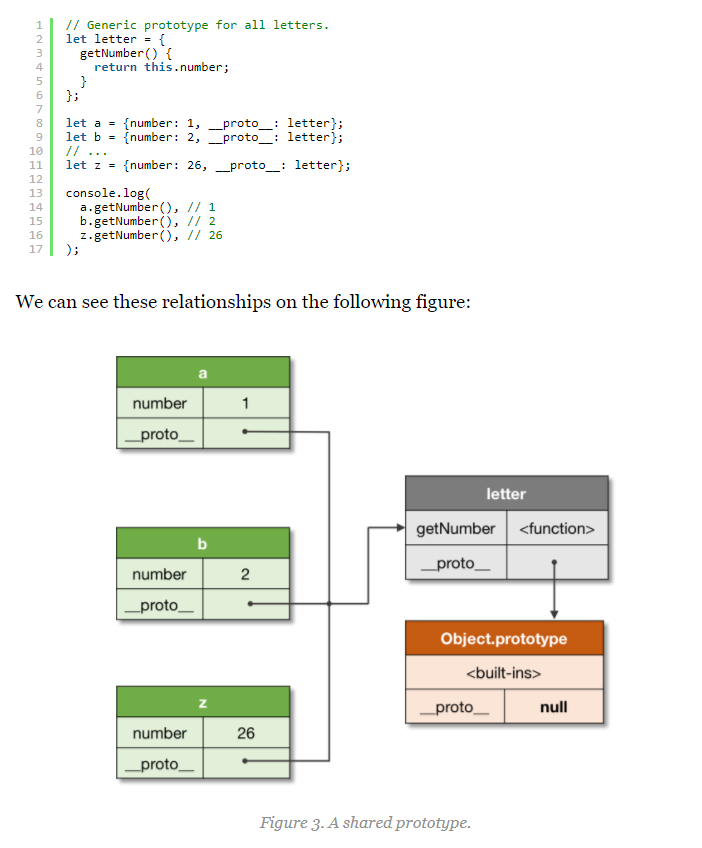
On the example of Object.prototype, we see that the same prototype can be shared across multiple objects. On this principle the class-based inheritance is implemented in ECMAScript. Let’s see the example, and look under the hood of the “class” abstraction in JS.

## [Class](http://dmitrysoshnikov.com/ecmascript/javascript-the-core-2nd-edition/#class)

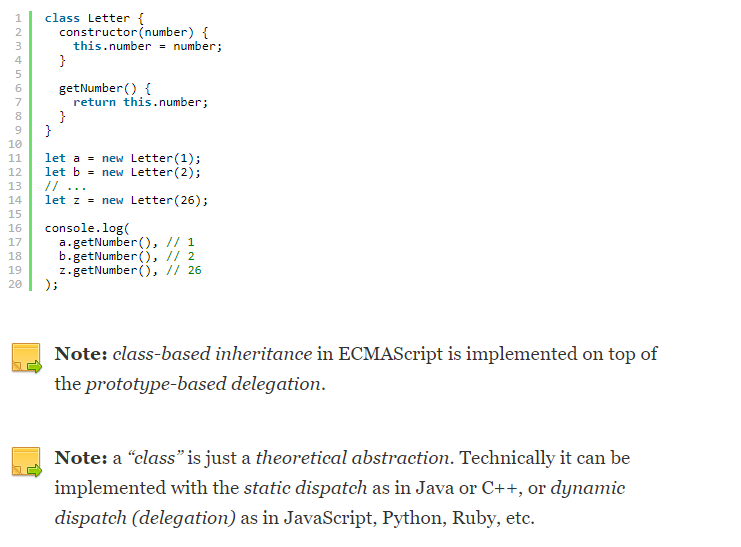
When several objects share the same initial state and behavior, they form a classification.

**Def. 5: Class:** A class is a formal abstract set which specifies initial state and behavior of its objects.

In case we need to have multiple objects inheriting from the same prototype, we could of course create this one prototype, and explicitly inherit it from the newly created objects:



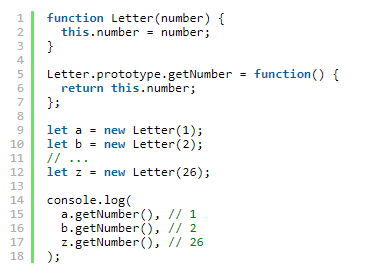
However, this is obviously cumbersome. And the class abstraction serves exactly this purpose — being a syntactic sugar (i.e. a construct which semantically does the same, but in a much nicer syntactic form), it allows creating such multiple objects with the convenient pattern:



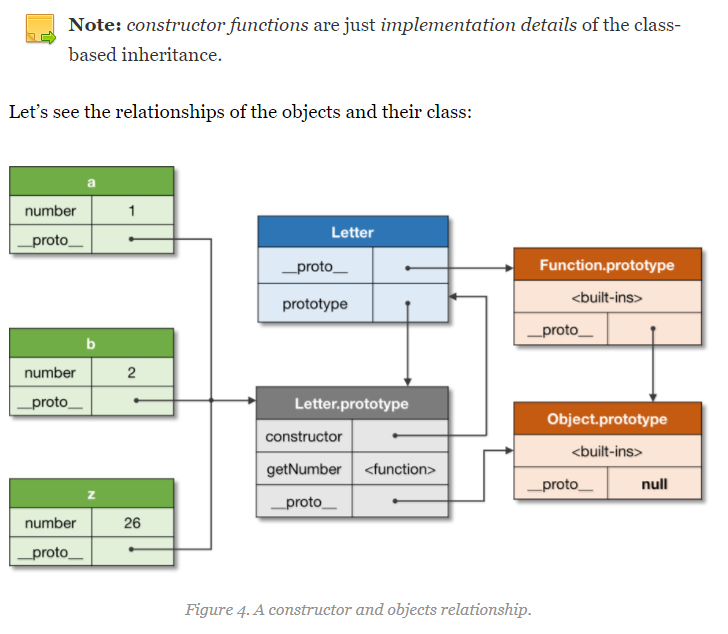
Technically a “class” is represented as a “constructor function + prototype” pair. Thus, a constructor function creates objects, and also automatically sets the prototype for its newly created instances. This prototype is stored in the <ConstructorFunction>.prototype property.

**Def. 6: Constructor:** A constructor is a function which is used to create instances, and automatically set their prototype.

It is possible to use a constructor function explicitly. Moreover, before the class abstraction was introduced, JS developers used to do so not having a better alternative (we can still find a lot of such legacy code allover the internets):



And while creating a single-level constructor was pretty easy, the inheritance pattern from parent classes required much more boilerplate. Currently this boilerplate is hidden as an implementation detail, and that exactly what happens under the hood when we create a class in JavaScript.



The figure above shows that every object has an associated prototype. Even the constructor function (class) Letter has its own prototype, which is Function.prototype. Notice, that Letter.prototype is the prototype of the Letter instances, that is a, b, and z.

**Note:** the actual prototype of any object is always the \_\_proto\_\_reference. And the explicit prototype property on the constructor function is just a reference to the prototype of its instances; from instances it’s still referred by the \_\_proto\_\_

You can find a detailed discussion on generic OPP concepts (including detailed descriptions of the class-based, prototype-based, etc) in the [ES3. 7.1 OOP: The general theory](http://dmitrysoshnikov.com/ecmascript/chapter-7-1-oop-general-theory/) article.

Now when we understand the basic relationships between ECMAScript objects, let’s take a deeper look at JS runtime system. As we will see, almost everything there can also be presented as an object.

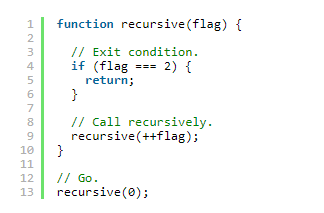
## [Execution context](http://dmitrysoshnikov.com/ecmascript/javascript-the-core-2nd-edition/#execution-context)

To execute JS code and track its runtime evaluation, ECMAScript spec defines the concept of an execution context. Logically execution contexts are maintained using a stack (the execution context stack as we will see shortly), which corresponds to the generic concept of a call-stack.

**Def. 7: Execution context:** An execution context is a specification device that is used to track the runtime evaluation of the code.

There are several types of ECMAScript code: the global code, function code, evalcode, and module code; each code is evaluated in its execution context. Different code types, and their appropriate objects may affect the structure of an execution context: for example, generator functions save their generator object on the context.

Let’s consider a recursive function call:

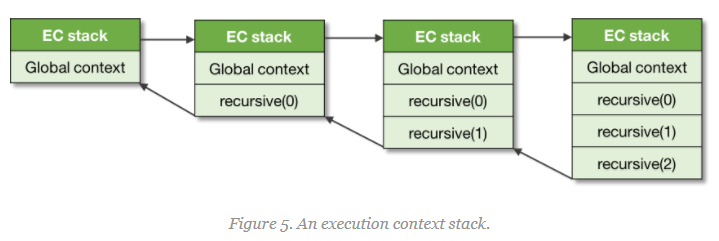


When a function is called, a new execution context is created, and pushed onto the stack — at this point it becomes an active execution context. When a function returns, its context is popped from the stack.

A context which calls another context is called a caller. And a context which is being called, accordingly, is a callee. In our example the recursive function plays both roles: of a callee and a caller — when calls itself recursively.

**Def. 8: Execution context stack:** An execution context stack is a LIFO structure used to maintain control flow and order of execution.

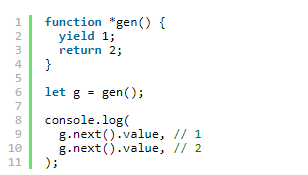
For our example from above we have the following stack “push-pop” modifications:



As we can also see, the global context is always at the bottom of the stack, it is created prior execution of any other context.

You can find more details on execution contexts in the [appropriate chapter](http://dmitrysoshnikov.com/ecmascript/chapter-1-execution-contexts/).

In general, the code of a context runs to completion, however as we mentioned above, some objects — such as generators, may violate LIFO order of the stack. A generator function may suspend its running context, and remove it from the stack before completion. Once a generator is activated again, its context is resumed and again is pushed onto the stack:



The yield statement here returns the value to the caller, and pops the context. On the second next call, the same context is pushed again onto the stack, and is resumed. Such context may outlive the caller which creates it, hence the violation of the LIFO structure.

We shall now discuss the important components of an execution context; in particular we should see how ECMAScript runtime manages variables storage, and scopes created by nested blocks of a code. This is the generic concept of lexical environments, which is used in JS to store data, and solve the “Funarg problem” — with the mechanism of closures.

## [Environment](http://dmitrysoshnikov.com/ecmascript/javascript-the-core-2nd-edition/#environment)

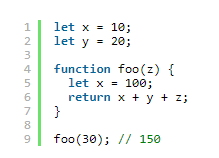
Every execution context has an associated lexical environment.

**Def. 9: Lexical environment:** A lexical environment is a structure used to define association between identifiers appearing in the context with their values. Each environment can have a reference to an optional parent environment.

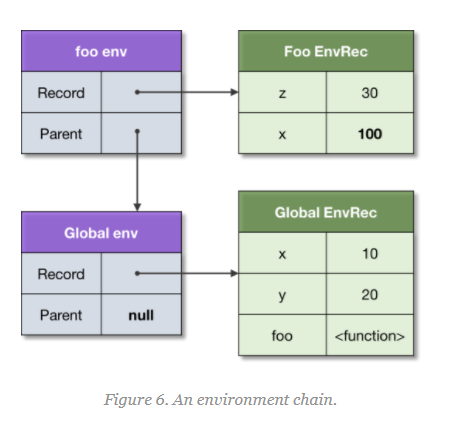
So an environment is a storage of variables, functions, and classes defined in a scope.

Technically, an environment is a pair, consisting of an environment record (an actual storage table which maps identifiers to values), and a reference to the parent (which can be null).

For the code:



The environment structures of the global context, and a context of the foo function would look as follows:



Logically this reminds us of the prototype chain which we’ve discussed above. And the rule for identifiers resolution is very similar: if a variable is not found in the own environment, there is an attempt to lookup it in the parent environment, in the parent of the parent, and so on — until the whole environment chain is considered.

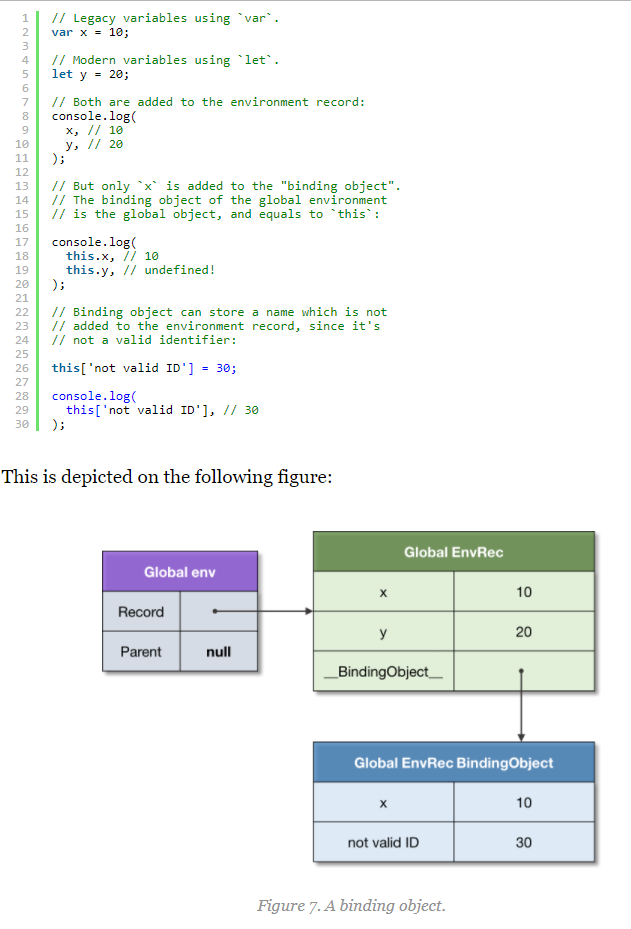
**Def. 10: Identifier resolution:** the process of resolving a variable (binding) in an environment chain. An unresolved binding results to ReferenceError.

This explains why variable x is resolved to 100, but not to 10 — it is found directly in the own environment of foo; why we can access parameter z — it’s also just stored on the activation environment; and also why we can access the variable y— it is found in the parent environment.

Similarly to prototypes, the same parent environment can be shared by several child environments: for example, two global functions share the same global environment.

Environment records differ by type. There are **object** environment records and **declarative** environment records. On top of the declarative record there are also **function** environment records, and **module** environment records. Each type of the record has specific only to it properties. However, the generic mechanism of the identifier resolution is common across all the environments, and doesn’t depend on the type of a record.

An example of an object environment record can be the record of the global environment. Such record has also associated binding object, which may store some properties from the record, but not the others, and vice-versa. The binding object can also be provided as this value.



Notice, the binding object exists to cover legacy constructs such as var-declarations, and with-statements, which also provide their object as a binding object. These are historical reason when environments were represented as simple objects. Currently the environments model is much more optimized, however as a result we can’t access binding as properties anymore.

We have already seen how environments are related via the parent link. Now we shall see how an environment can outlive the context which creates it. This is the basis for the mechanism of closures which we’re about to discuss.

## [Closure](http://dmitrysoshnikov.com/ecmascript/javascript-the-core-2nd-edition/#closure)

Functions in ECMAScript are first-class. This concept is fundamental to functional programming, which aspects are supported in JavaScript.

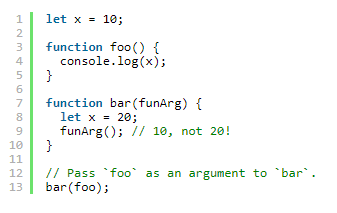
**Def. 11: First-class function:** a function which can participate as a normal data: be stored in a variable, passed as an argument, or returned as a value from another function.

With the concept of first-class functions so called [“Funarg problem”](https://en.wikipedia.org/wiki/Funarg_problem) is related (or “A problem of a functional argument”). The problem arises when a function has to deal with free variables.

**Def. 12: Free variable:** a variable which is neither a parameter, nor a local variable of this function.

Let’s take a look at the Funarg problem, and see how it’s solved in ECMAScript.

Consider the following code snippet:



For the function foo the variable x is free. When the foo function is activated (via the funArg parameter) — where should it resolve the x binding? From the outer scope where the function was created, or from the caller scope, from where the function is called? As we see, the caller, that is the bar function, also provides the binding for x — with the value 20.

The use-case described above is known as **downward funarg problem**, i.e. an ambiguity at determining a correct environment of a binding: should it be an environment of the creation time, or environment of the call time?

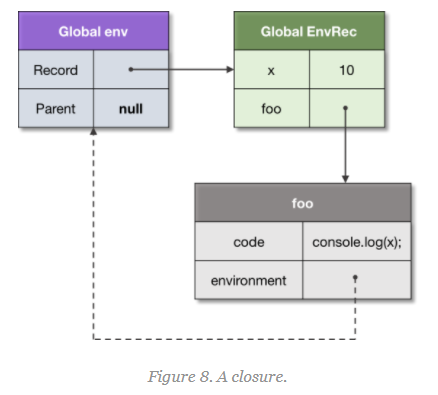
This is solved by an agreement of using static scope, that is the scope of the creation time.

**Def. 13: Static scope:** a language implements static scope, if only by looking at the source code one can determine in which environment a binding is resolved.

The static scope sometimes is also called lexical scope, hence the lexical environments naming.

Technically the static scope is implemented by capturing the environment where a function is created.

In our example, the environment captured by the foo function, is the global environment:



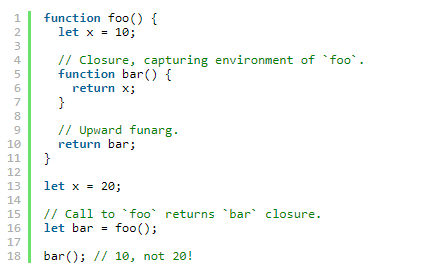
We can see that an environment references a function, which in turn reference the environment back.

**Def. 14: Closure:** A closure is a function which captures the environmentwhere it’s defined. Further this environment is used for identifier resolution.

**Note:** a function is called in a fresh activation environment which stores local variables, and arguments. The parent environment of the activation environment is set to the closured environment of the function, resulting to the lexical scope semantics.

The second sub-type of the Funarg problem is known as the **upward funarg problem**. The only difference here is that a capturing environment outlives the context which creates it.

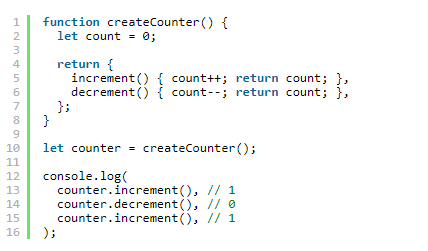
Let’s see the example:



Again, technically it doesn’t differ from the same exact mechanism of capturing the definition environment. Just in this case, hadn’t we have the closure, the activation environment of foo would be destroyed. But we captured it, so it cannot be deallocated, and is preserved — to support static scope semantics.

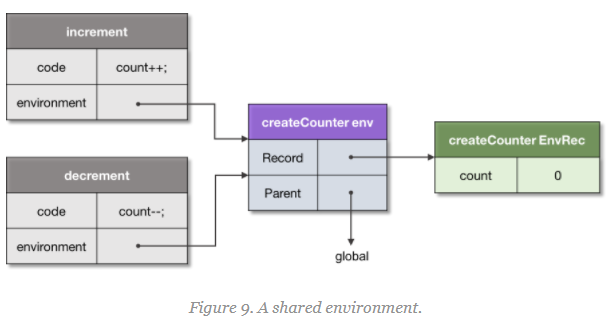
Often there is an incomplete understanding of closures — usually developers think about closures only in terms of the upward funarg problem (and practically it really makes more sense). However, as we can see, technical mechanism for downwardand upward funarg problem is exactly the same — and is the mechanism of static scope.

As we mentioned above, similarly to prototypes, the same parent environment can be shared across several closures. This allows accessing and mutating the shared data:



Since both closures, increment and decrement, are created within the scope containing the count variable, they share this parent scope. That is, capturing always happens “by-reference” — meaning the reference to the whole parent environment is stored.

We can see this on the following picture:



Some languages may capture by-value, making a copy of a captured variable, and do not allow changing it in the parent scopes. However in JS, to repeat, it is always the reference to the parent scope.

**Note:** implementations may optimize this step, and do not capture the whole environment. Capturing only used free-variables, they though still maintain invariant of mutable data in parent scopes.

You can find a detailed discussion on closures and the Funarg problem in the [appropriate chapter](http://dmitrysoshnikov.com/ecmascript/chapter-6-closures/).

So all identifiers are statically scoped. There is however one value which is dynamically scoped in ECMAScript. It’s the value of this.

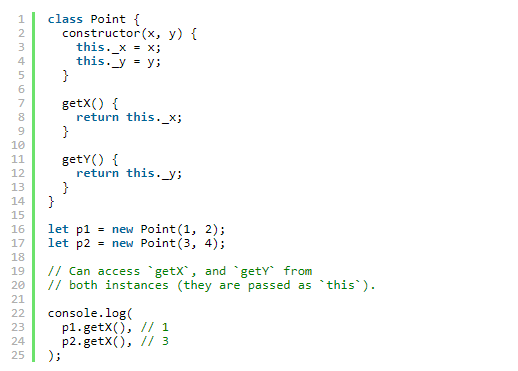
## [This](http://dmitrysoshnikov.com/ecmascript/javascript-the-core-2nd-edition/#this)

The this value is a special object which is dynamically and implicitly passed to the code of a context. We can consider it as an implicit extra parameter, which we can access, but cannot mutate.

The purpose of the this value is to executed the same code for multiple objects.

**Def. 15: This:** an implicit context object accessible from a code of an execution context — in order to apply the same code for multiple objects.

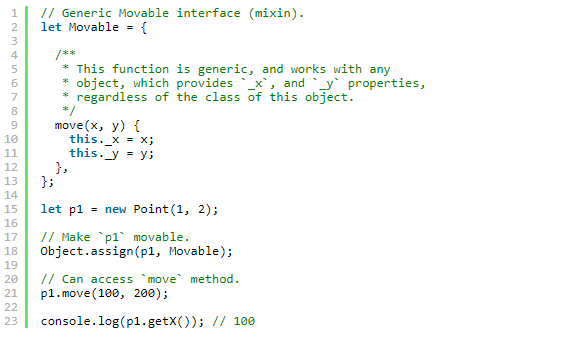
The major use-case is the class-based OOP. An instance method (which is defined on the prototype) exists in one exemplar, but is shared across all the instances of this class.



When the getX method is activated, a new environment is created to store local variables and parameters. In addition, function environment record gets the [[ThisValue]] passed, which is bound dynamically depending how a function is called. When it’s called with p1, the this value is exactly p1, and in the second case it’s p2.

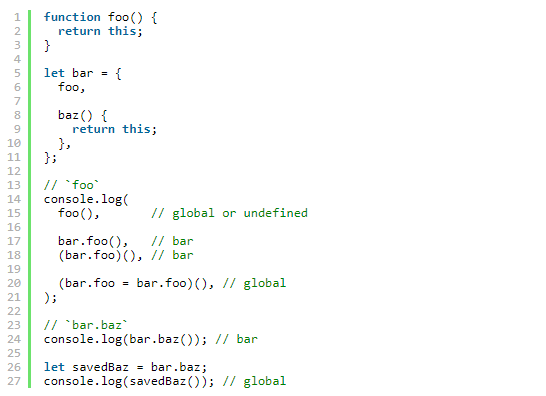
Another application of this, is generic interface functions, which can be used in mixins or traits.

In the following example, the Movable interface contains generic function move, which expects the users of this mixin to implement \_x, and \_y properties:



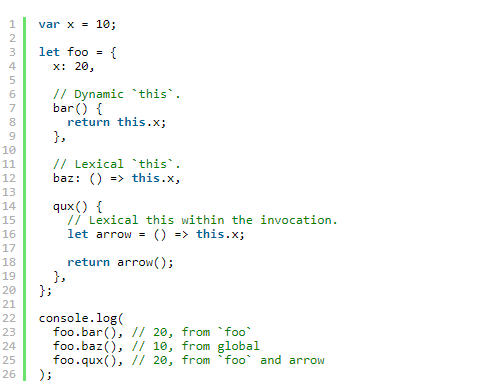
As an alternative, a mixin can also be applied at prototype level instead of per-instance as we did in the example above.

Just to show the dynamic nature of this value, consider this example, which we leave to a reader as an exercise to solve:



Since only by looking at the source code of the foo function we cannot tell what value of this will it have in a particular call, we say that this value is dynamically scoped.

The **arrow functions** are special in terms of this value: their this is lexical (static), but not dynamic. I.e. their function environment record does not provide this value, and it’s taken from the parent environment.



Like we said, in the global context the this value is the global object (the binding object of the global environment record). Previously there was only one global object. In current version of the spec there might be multiple global objects which are part of code realms. Let’s discuss this structure.

## [Realm](http://dmitrysoshnikov.com/ecmascript/javascript-the-core-2nd-edition/#realm)

Before it is evaluated, all ECMAScript code must be associated with a realm. Technically a realm just provides a global environment for a context.

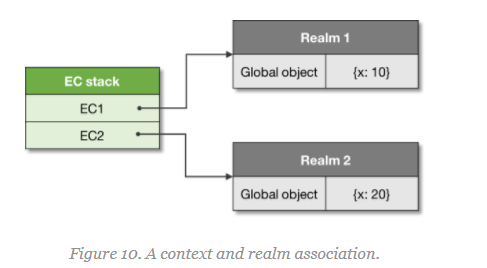
**Def. 16: Realm:** A code realm is an object which encapsulates a separate global environment.

When an execution context is created it’s associated with a particular code realm, which provides the global environment for this context. This association further stays unchanged.

**Note:** a direct realm equivalent in browser environment is the iframeelement, which exactly provides a custom global environment. In Node.js it is close to the sandbox of the [vm module](https://nodejs.org/api/vm.html).

Current version of the specification doesn’t provide an ability to explicitly create realms, but they can be created implicitly by the implementations. There is a [proposal](https://github.com/tc39/proposal-realms/) though to expose this API to user-code.

Logically though, each context from the stack is always associated with its realm:



Now we’re getting closer to the bigger picture of the ECMAScript runtime. Yet however we still need to see the entry point to the code, and the initialization process. This is managed by the mechanism of jobs and job queues.

## [Job](http://dmitrysoshnikov.com/ecmascript/javascript-the-core-2nd-edition/#job)

Some operations can be postponed, and executed as soon as there is an available spot on the execution context stack.

**Def. 17: Job:** A job is an abstract operation that initiates an ECMAScript computation when no other ECMAScript computation is currently in progress.

Jobs are enqueued on the **job queues**, and in current spec version there are two job queues: **ScriptJobs**, and **PromiseJobs**.

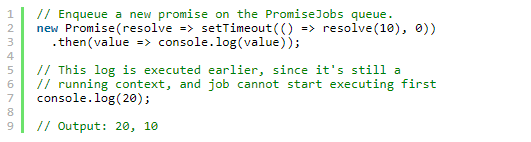
And initial job on the ScriptJobs queue is the main entry point to our program — initial script which is loaded and evaluated: a realm is created, a global context is created and is associated with this realm, it’s pushed onto the stack, and the global code is executed.

Notice, the ScriptJobs queue manages both, scripts and modules.

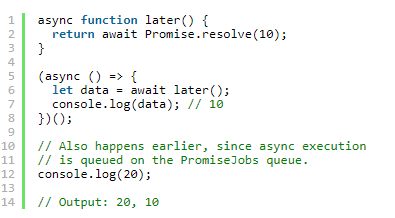
Further this context can execute other contexts, or enqueue other jobs. An example of a job which can be spawned and enqueued is a promise.

When there is no running execution context and the execution context stack is empty, the ECMAScript implementation removes the first pending job from a job queue, creates an execution context and starts its execution.

Example:



The ***async functions*** can await for promises, so they also enqueue promise jobs:



Now we’re very close to the final picture of the current JS Universe. We shall see now main owners of all those components we discussed, the Agents.

## [Agent](http://dmitrysoshnikov.com/ecmascript/javascript-the-core-2nd-edition/#agent)

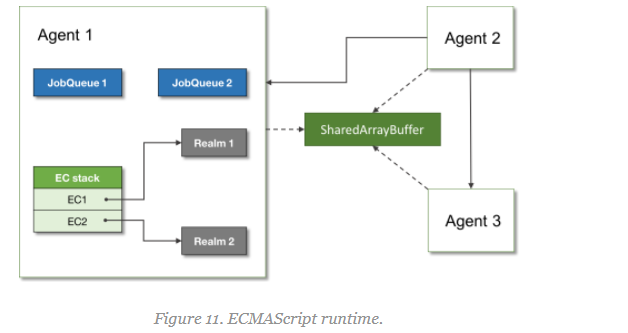
The concurrency and parallelism is implemented in ECMAScript using Agent pattern. The Agent pattern is very close to the [Actor pattern](https://en.wikipedia.org/wiki/Actor_model) — a lightweight process with message-passing style of communication.

**Def. 18: Agent:** An agent is an abstraction encapsulating execution context stack, set of job queues, and code realms.

Implementation dependent an agent can run on the same thread, or on a separate thread. A worker agent in the browser environment is an example of the Agentconcept.

The agents are state isolated from each other, and can communicate by sending messages. Some data can be shared though between agents, for example SharedArrayBuffers. Agents can also combine into agent clusters.

So below is the picture of the ECMAScript runtime:



And that is it; that’s what happens under the hood of the ECMAScript engine!

Now we come to an end. This is the amount of information on JS core which we can cover within an overview article. Like we mentioned, JS code can be grouped into modules, properties of objects can be tracked by Proxy objects, etc, etc. — there are many user-level details which you can find in different documentations on JavaScript language.

Here though we tried to represent the logical structure of an ECMAScript program itself, and hopefully it clarified these details. If you have any questions, suggestions or feedback, — as always I’ll be glad to discuss them in comments.

I’d like to thank the TC-39 representatives and spec editors which helped with clarifications for this article. The discussion can be found in [this Twitter thread](https://twitter.com/DmitrySoshnikov/status/930507793047592960).