Javascript
Coding Best Practices

Matteo Ronchi - 2018

Javascript

Closures

A closure is an inner function that has access to the outer (enclosing) function's variables

-- Dan Wellman (javascriptissexy.com)

Closures can be used for many purposes:

- Namespaces
- (Revealing) Module Pattern
- IIFE *(immediately-invoked function expression)*

Code

Grab It: WorkWave/js-best-practices

Javascript

Namespaces

Namespaces can be considered a logical grouping of units of code under a unique identifier.
The identifier can be referenced in many namespaces [..]

-- Addy Osmani

Namespaces

- help us avoid collisions with other objects or variables in the global namespace
- help to organize blocks of functionality
- it's a safeguard preventing our code from breaking if other script use the same method names

Javascript doesn't have built-in support for namespaces

But we can use several techniques to implement namespaces (objects, IIFE, closures)

Javascript

Namespace Patterns

Namespace Patterns

- Single global variables
- Prefix namespacing
- Object literal notation
- Nested namespacing
- IIFE

Code

Javascript

Functional Javascript principles

Core concepts

- Pure functions
- Function composition
- Avoid shared state
- Avoid mutating state
- Avoid side effects

Functional programming (FP) is the process of building software by composing pure functions, avoiding shared state, mutable data, and side-effects.
Functional programming is declarative rather than imperative, and application state flows through pure functions.
Contrast with object oriented programming, where application state is usually shared and colocated with methods in objects.

-- Eric Elliot

A pure function is a function which:

Given the same inputs, **always** returns the same output, and has **no side-effects**.

Pure functions **depend only on the inputs** of the function, and the output should be the exact same for the same input.

Side Effects

A side effect is any application state change that is observable outside the called function other than its return value.

Side effects include:

- Modifying any external variable or object property
- Logging to the console
- Writing to the screen
- Writing to a file
- Writing to the network
- Triggering any external process
- Calling any other functions with side-effects

Code

Javascript

Memory management

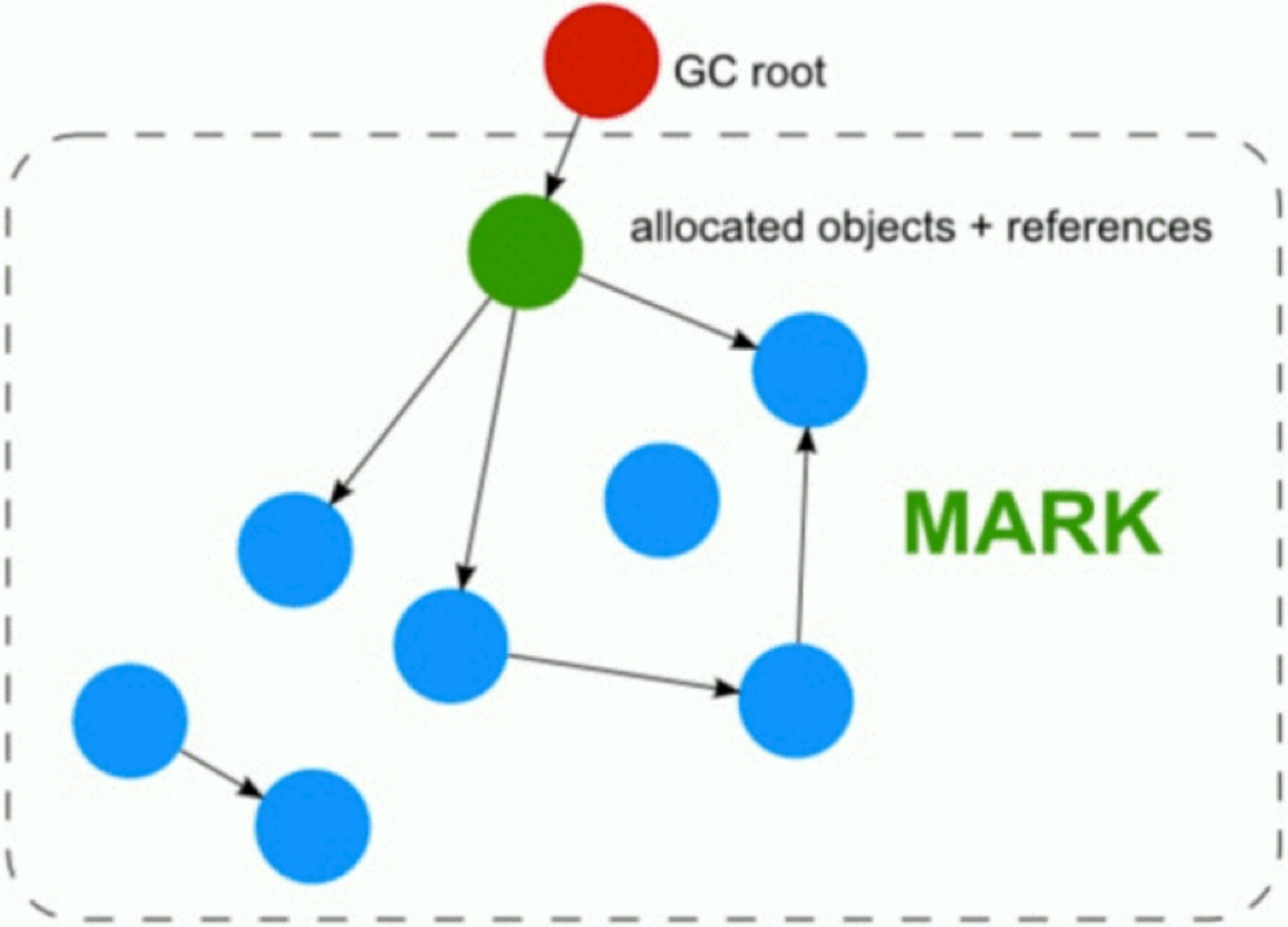
Memory life cycle

1. Allocate the memory you need
2. Use the allocated memory (read, write)
3. Release the allocated memory when it is not needed anymore

Mark-and-sweep algorithm

1. The GC builds a list of **roots**, global variables to which a reference is kept in code. A **root** is **always present**, so the GC consider it and all of its children to be always present.
2. All roots and their children are recursively inspected and marked as active. Everything that can be reached from a root is not considered garbage.
3. All pieces of memory not marked as active can now be considered garbage and can be freed and returned to the OS.

Mark and sweep (MARK)



Javascript

Common Memory Leaks

global variables

when a undeclared variable is referenced, a new variable gets created in the global object.

```
function foo(arg) {
    bar = "some text";
}
```

is the equivalent of:

```
function foo(arg) {
    window.bar = "some text";
}
```

You can avoid all this by adding ***'use strict'****; at the beginning of your JavaScript file which would switch on a much stricter mode of parsing JavaScript which prevents the unexpected creation of global variables.*

Forgotten timers or callbacks

```
var someResource = getData();

var intervalId = setInterval(function() {
    var node = document.getElementById('Node');
    if(node) {
        node.innerHTML = JSON.stringify(someResource));
    } else {
      // without this condition 'someResource' will not GC
      clearInterval(intervalId)
    }
}, 1000);
```

Observers

```
var element = document.getElementById('button');

function onClick(event) {
    element.innerHtml = 'text';
}

element.addEventListener('click', onClick);
// Do stuff

element.removeEventListener('click', onClick);
element.parentNode.removeChild(element);
// Now when element goes out of scope,
// both element and onClick will be collected even in old browsers that don't
// handle cycles well.
```

Out of DOM references

```
var elements = {
    button: document.getElementById('button'),
};

function doStuff() {
  button.click();
}

function removeButton() {
    // The button is a direct child of body.
    document.body.removeChild(document.getElementById('button'));

    // we still have a reference to #button in the global object.
    // The button element is still in memory and cannot be collected by the GC.
}
```

Closures

```
var theThing = null;

var replaceThing = function () {
  var originalThing = theThing;
  var unused = function () {
    if (originalThing)
      console.log("hi");
  };
  theThing = {
    longStr: new Array(1000000).join('*'),
    someMethod: function () {
      console.log(someMessage);
    }
  };
};

setInterval(replaceThing, 1000);
```

Javascript

DOM Interactions

DOM manipulation with vanilla JS ***is not*** *rocket science*

Code

Useful Links

- the-poor-misunderstood-innerText
- essential-js-namespacing
- essential js design patterns
- javascript-namespacing
- immediately-invoked-function-expressions-and-parentheses
- immediately-invoked-function-expression

Useful Links

- memory-management-and-garbage-collection
- four-types-of-leaks-in-your-javascript-code
- how-javascript-works-memory-management
- what-is-a-closure
- understand-javascript-closures-with-ease
- what-is-functional-programming
- functional-programming-for-javascript-people

Thanks