## What are microtasks?

A **microtask** is a short function which is executed after the function or program which created it exits and only if the JavaScript execution stack is empty, but before returning control to the event loop being used by the task that calls the script's execution environment.

## What is a microtask queue?

Microtask queue is an internal queue PromiseJob used for proper task management.

## What is their role in Promises and how are they different from callbacks?

Promises have been introduced in [E](https://www.ecma-international.org/ecma-262/6.0/)S6 (2015) to allow for more readable asynchronous code than is possible with callbacks.

The main difference between callbacks and promises is that with callbacks you tell the executing function what to do when the asynchronous task completes, whereas with promises the executing function returns a special object to you (the promise) and then you tell the promise what to do when the asynchronous task completes.

## Explain with examples how private, protected variables can be implemented in classes and how can they be used in subclasses?

Private Vriables:

PrivateParts.createKey(); is used to store it on the underscore variable, the underscore variable that stores the result of the createKey method is actually a function that takes the this object and returns its private instance.

var Car = (function() {

var \_ = PrivateParts.createKey();

function Car(mileage) {

*// Store the mileage property privately.*

\_(this).mileage = mileage;

}

Car.prototype.drive = function(miles) {

if (typeof miles == 'number' && miles > 0) {

\_(this).mileage += miles;

} else {

throw new Error('drive only accepts positive numbers');

}

}

Car.prototype.readMileage = function() {

return \_(this).mileage;

}

return Car;

}());

Protected Variables:

The below citizen class defines both public and protected methods and uses the passed key function to store data on the instance.

To subclass citizen, simply call its subclass method. As you’ll see, two of the methods in this subclass (init and allowedToVote) are overridden and call super, and the vote method is simply inherited as you’d expect from a subclass.

var ctor = require('mozart');

var Citizen = ctor(function(prototype, \_, \_protected) {

// == PUBLIC ==

prototype.init = function(name, age) {

\_(this).name = name;

\_(this).age = age;

};

prototype.vote = function(politician) {

if (\_(this).allowedToVote()) {

console.log(\_(this).name + ' voted for ' + politician);

} else {

throw new Error(\_(this).name + ' is not allowed to vote.');

}

};

// == PROTECTED ==

\_protected.allowedToVote = function() {

return this.age > 18;

};

});

var Criminal = Citizen.subclass(function(prototype, \_, \_protected) {

prototype.init = function(name, age, crime) {

\_(this).crime = crime;

prototype.super.init.call(this, name, age);

};

\_protected.allowedToVote = function() {

return \_(this).crime != 'felony'

&& \_protected.super.allowedToVote.call(this);

};

});

var joe = new Criminal('Joe', 27, 'felony');

joe.vote('Obama')

Public Variables:

public scope is available to everything that is able to access a class' instantiated object. If this sounds obvious, it's because that's the only way JavaScript does things. If you create an object in JavaScript, you can access all of its properties:

const object = { publicProperty: "I'm Public!" };

So long as you can access object, you can also access publicProperty and any other property that is attached in this way.