Node.JS
Библиотеки и фреймворки
Часть 1

к.т.н. доцент каф. ВТ
Медведев М.С.

Node.JS

Принцип работы Node.js предполагает, быстрый ответ на входящие запросы.

Такой подход стоит применять не только к I/O: сложные расчеты, загружающие процессор должны быть вынесены в отдельные процессы, с которыми вы сможете взаимодействовать с помощью событий.

Node.JS

Тяжелые операции можно вынести из основного потока, используя такие абстракции, как WebWorkers.

Это подразумевает, что вы не сможете распараллелить работу вашего кода, без использования отдельного фонового потока, с которым вы сможете взаимодействовать с помощью событий.

Node.JS

Все объекты, которые могут бросать события (EventEmitter) поддерживают асинхронное взаимодействие на основе коллбеков.

Вы можете использовать это в работе с блокирующим кодом. Взаимодействие можно организовать с помощью файлов, сокетов, или дочерних процессов, каждый из которых будет представлен в Node.js инстансом EventEmitter.

Node.JS

Внутренняя реализация

Внутри, для работы событийного цикла, Nodejs использует libev.

В приложении к нему применяется библиотека libeio, которая использует очереди потоков для реализации асинхронного ввода/вывода.

Узнать об этом больше вы можете в документации к libev.

Node.JS

Шаблоны реализации асинхронности

Функции первого класса. Мы подходим к таким функциям как к данным, «разбрасывая» их вокруг и выполняя их по необходимости.

Функции первого класса трактуются как объекты, то есть могут быть переданы другим функциям и их можно вернуть из функций. Так же их можно присваивать переменным.

Node.JS

```
/* функция не имеет имени и находится в правой
части команды присваивания переменной.*/

var dog = function(num) {
    for (var i = 0; i < num; i++) {
        alert("Woof");
    }
};

dog(3); //Эту функцию можно вызвать через
переменную dog.
```

Node.JS

Функциональные композиции.

Так же известны как анонимные функции или замыкания, которые выполняются после того, как что-то произойдет в I/O.

Node.JS

Замыкания - это функции, ссылающиеся на независимые (свободные) переменные. Другими словами, функция, определённая в замыкании, «запоминает» окружение, в котором она была создана.

Независимые переменные — это все переменные, которые не были переданы как параметры и не были объявлены как локальные.

Node.JS

```
function numberGenerator() {
// Local “free” variable that ends up within the
closure
  var num = 1;
  function checkNumber() {
    console.log(num);
  }
  num++;
  return checkNumber;
}

var number = numberGenerator();
number(); // 2
```

Node.JS

В примере выше функция *numberGenerator* создаёт локальную переменную num (число), а также локальную функцию *checkNumber* (функцию, печатающую num в консоль разработчика).

Локальная функция *checkNumber* сама по себе не объявляет локальных переменных, но благодаря механизму замыкания ей доступна переменная из внешнего окружения функции *numberGenerator*.

Node.JS

В результате она может пользоваться переменной *num*, созданной во время вызова функции *numberGenerator*, даже после возврата из вызова *numberGenerator*.

Node.JS

```
function sayHello() {
  var say = function() { console.log(hello); }

// Local variable that ends up within the closure
  var hello = 'Hello, world!';
  return say;
}

var sayHelloClosure = sayHello();
sayHelloClosure(); // 'Hello, world!'
```

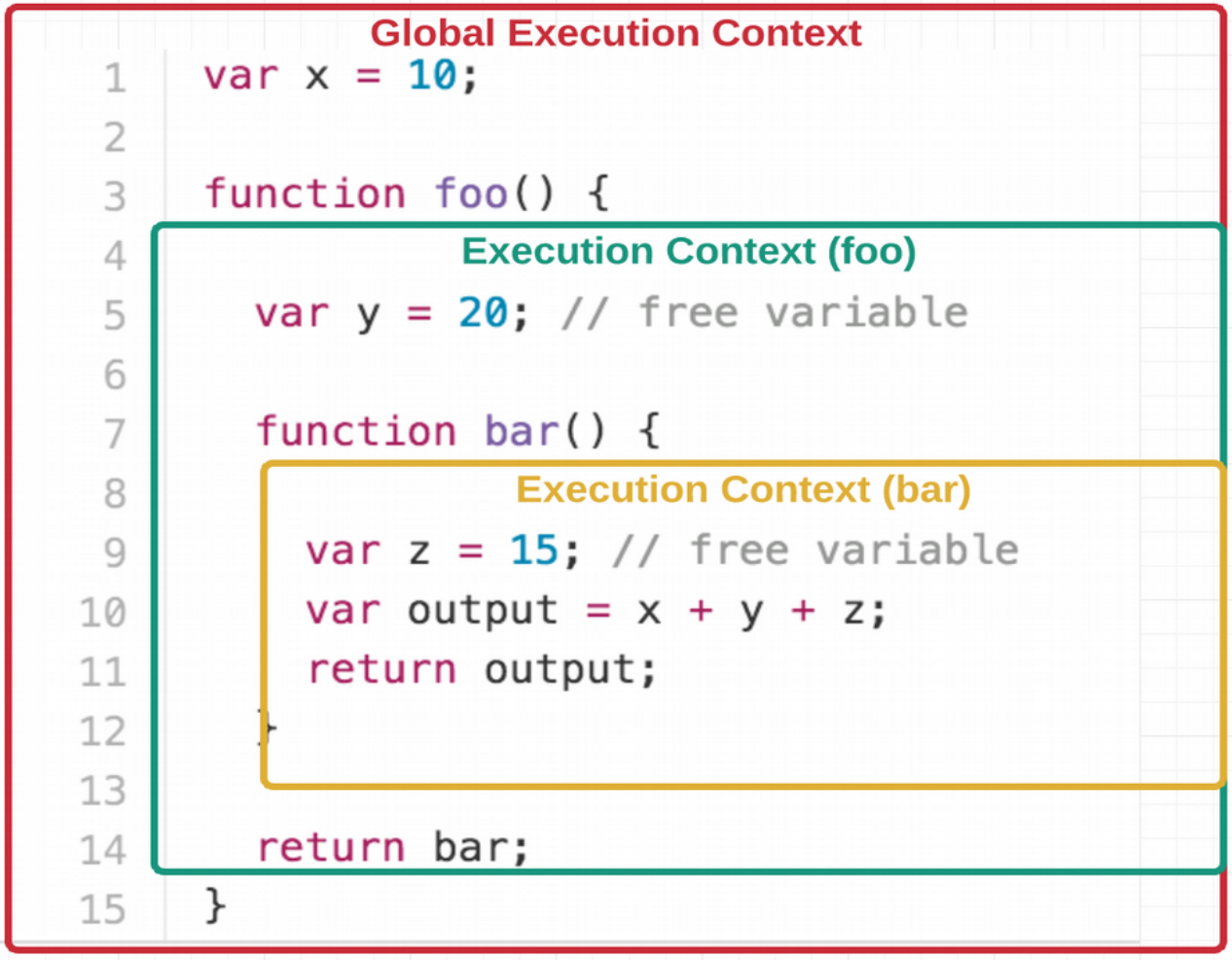
Node.JS

Контекст выполнения

это абстрактная концепция, в рамках которой спецификация языка Javascript, известная как ECMAScript, объясняет модель выполнения кода после запуска.

Контекст бывает либо **глобальным**, с которого начинается исполнения скрипта, либо **контекстом выполнения вызова**, который начинается с момента входа в тело функции.

Node.JS


Global Execution Context
Execution Context (foo)
Execution Context (bar)
1 var x = 10;
2
3 function foo() {
4
5 var y = 20; // free variable
6
7 function bar() {
8
9 var z = 15; // free variable
10 var output = x + y + z;
11 return output;
12 }
13
14 return bar;
15 }

Node.JS

В каждый момент времени активен только один контекст выполнения. Именно поэтому Javascript называют “однопоточным”, имея ввиду, что только одна инструкция исполняется в один момент времени.

Активный контекст выполнения находится на вершине стека. Он снимается со стека, когда выполнение кода активного контекста завершается, и выполнение продолжается в коде предыдущего контекста, который теперь оказался на вершине стека.

Node.JS

Если в данный момент исполняется код из некоторого контекста выполнения, это ещё не значит, что он должен завершить выполнение до запуска другого потока выполнения.

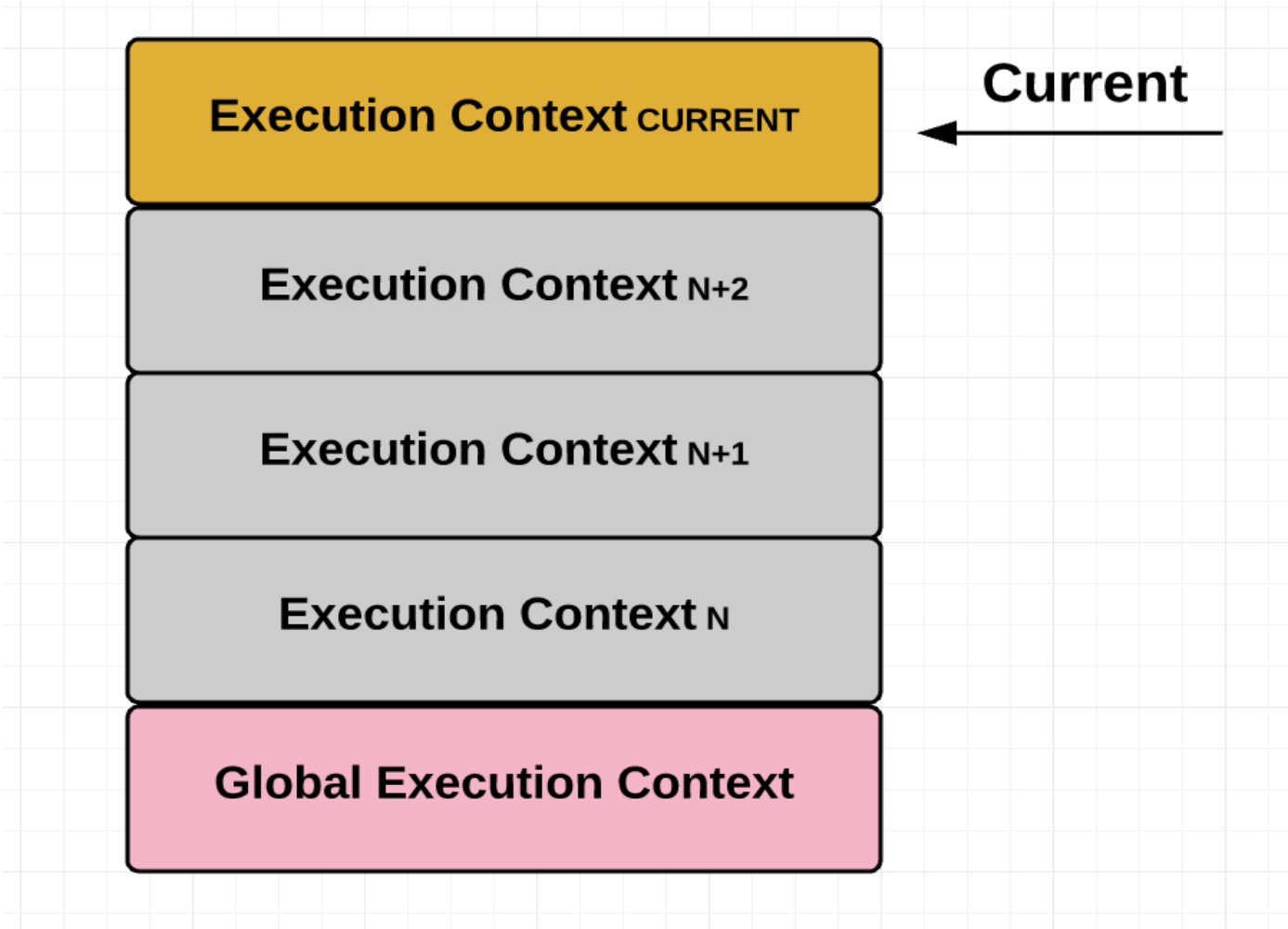
! Есть временные интервалы, на которых активный контекст засыпает, и другой контекст выполнения становится активным контекстом.

Node.JS

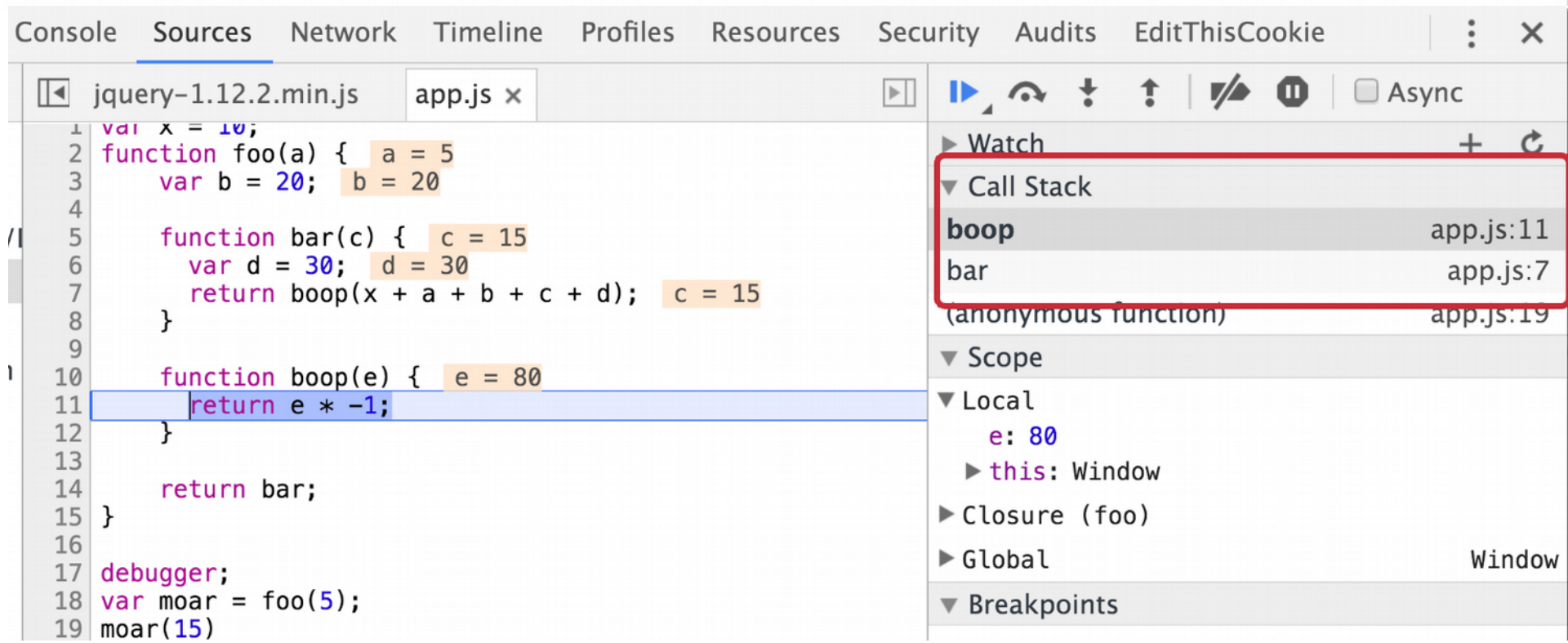
Заснувший контекст выполнения может позднее проснуться на той точке, в которой он ранее заснул.

Каждый раз при подобной замене одного контекста выполнения другим будет создаваться новый контекст выполнения, который попадёт на вершину стека и станет активным контекстом.

Node.JS

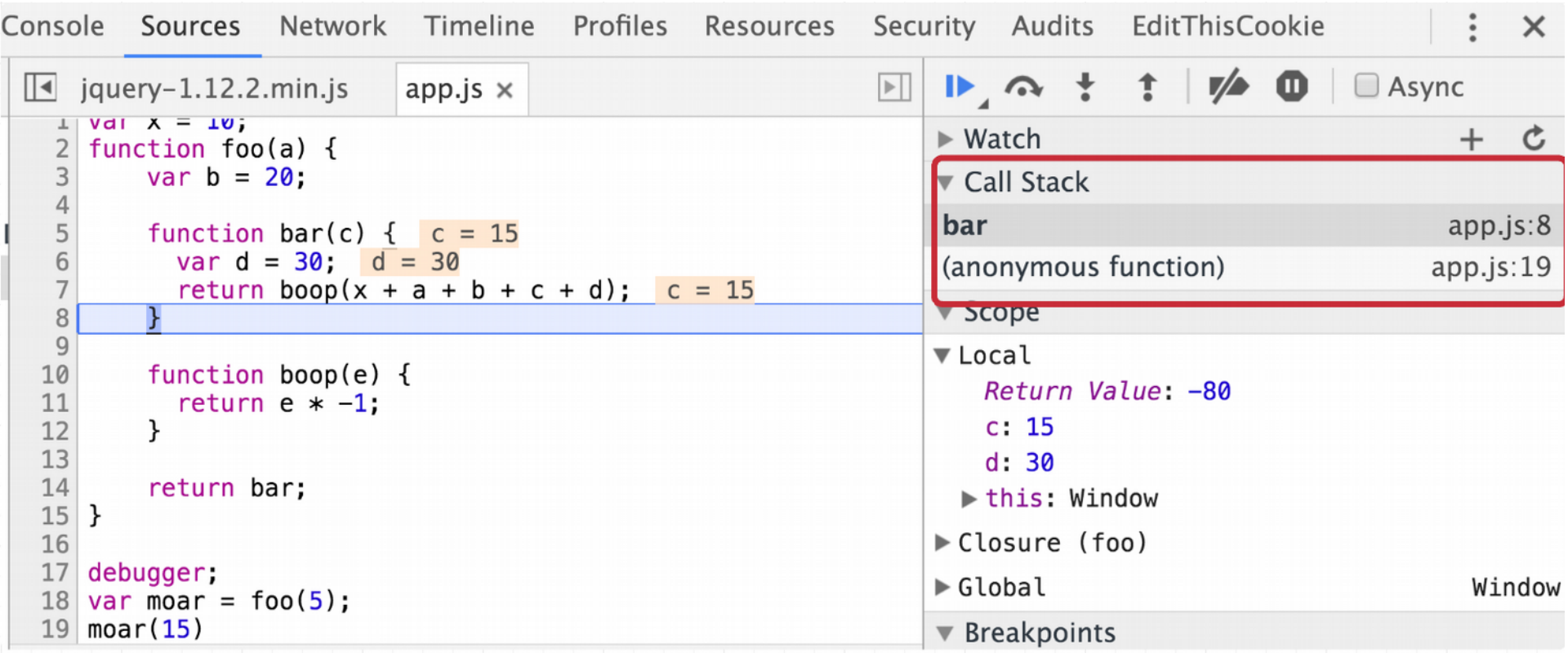

Execution Context CURRENT
Current
Execution Context N+2
Execution Context N+1
Execution Context N
Global Execution Context

Node.JS



В момент выполнения функции boop вы можете увидеть активный контекст выполнения в отладчике браузера:

Node.JS



Когда вызов функции boop возвращает управление, его контекст выполнения выталкивается из стека, и выполнение продолжается в контексте выполнения вызова bar

Node.JS

Мы имеем множество контекстов выполнения, работающих один за другим—часто с остановкой посреди исполнения и последующим продолжением.

Таким образом нужен способ отслеживать состояние, чтобы мы могли управлять порядком и процессом выполнения этих контекстов.

Node.JS

Каждый контекст выполнения имеет различные элементы состояния, которые можно использовать для отслеживания прогресса выполнения кода, достигнутого в каждом контексте выполнения:

1) Состояние вычисления (code evaluation state): всё состояние контекста, необходимое для исполнения, засыпания и продолжения вычисления кода, связанного с одним контекстом выполнения

Node.JS

2) Функция (function): объект функции, с вызовом которой связан контекст выполнения, либо null, если контекст выполнения связан со скриптом или модулем

3) Область (realm): набор внутренних объектов, глобальное окружение ECMAScript и связанные с ним ресурсы, а также весь код на ECMAScript, который загружается в области видимости глобального окружения

Node.JS

4) Лексическое окружение: используется для сопоставления “идентификаторов-ссылок”, используемых в коде внутри контекста выполнения

5) Таблица переменных (variable environment)— таблица, связанная с лексическим окружением, в которой в качестве ключей занасены все имена переменных, используемые в инструкциях объявления переменных

Лексическое окружение

термин, описывающий связывание идентификаторов с переменными и функциями, основанное на лексической вложенности кода, написанного на ECMAScript (Javascript).

Лексическое окружение состоит из таблицы символов и ссылки на внешнее лексическое окружение (нулевой для глобального окружения).

Node.JS

Лексическое окружение отвечает за управление данными в виде переменных и функций (или, если кратко, “символов”) в коде.
Оно задаёт смысл доступных идентификаторов.
Например, если мы имеем строку кода `console.log(x / 10)`, то смысл использования идентификатора `x` появляется, если есть связь между ним и символом.

Лексическое окружение задаёт связь идентификаторов с переменными и функциями через таблицу символов.

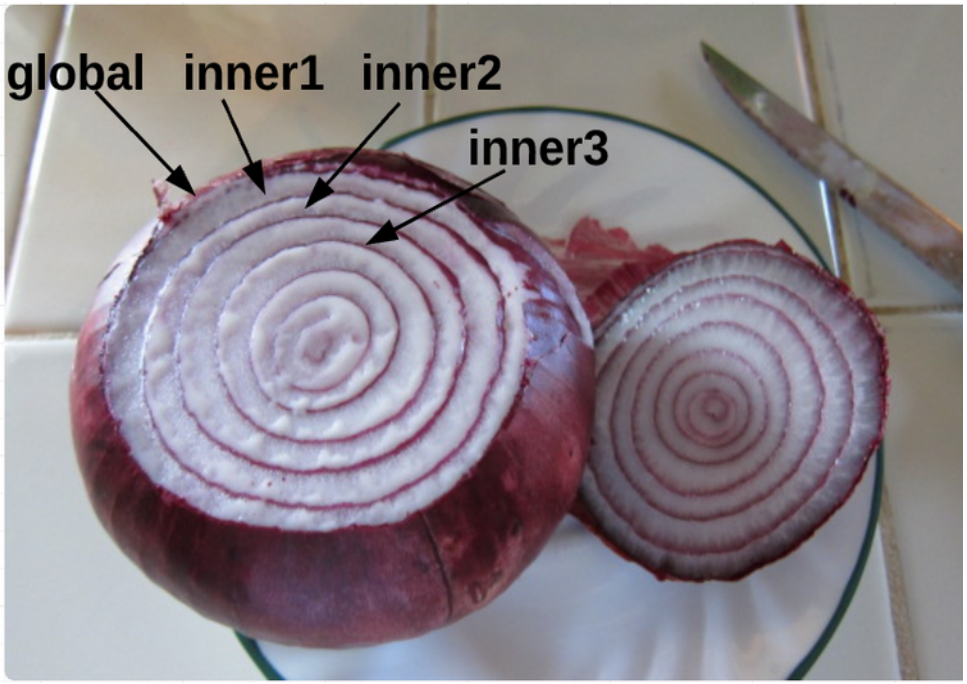
Node.JS

Таблица символов в виде ассоциативного массива (хеш-массива или бинарного дерева) даёт простой способ хранения записей обо всех идентификаторах и их привязки к символам внутри лексического окружения.

Каждое лексическое окружение имеет таблицу символов.

Node.JS

Вложенное окружение ссылается на внешнее окружение, а это внешнее окружение, в свою очередь, может иметь ссылку на окружающее его окружение. И только глобальное окружение не имеет внешнего окружения.



Node.JS

Каждый раз при вызове функции создаётся новое лексическое окружение.

Каждый контекст выполнения имеет лексическое окружение. Это окружение хранит переменные и их значения, имеет ссылку на внешнее окружение

Node.JS

Цепочка областей видимости (Scope Chain)
Окружение получает доступ к символам родительского окружения, а родительское окружение имеет доступ к своему родителю.
Объединённое множество идентификаторов, доступ к которым имеет текущее окружение, носит имя “область видимости” (scope).

В процессе движения от родительского окружения к дочернему число доступных идентификаторов возрастает, и сами области видимости складываются в “цепочку областей видимости”.

Node.JS

Цепочка областей видимости (Scope Chain)
Окружение получает доступ к символам родительского окружения, а родительское окружение имеет доступ к своему родителю.
Объединённое множество идентификаторов, доступ к которым имеет текущее окружение, носит имя “область видимости” (scope).

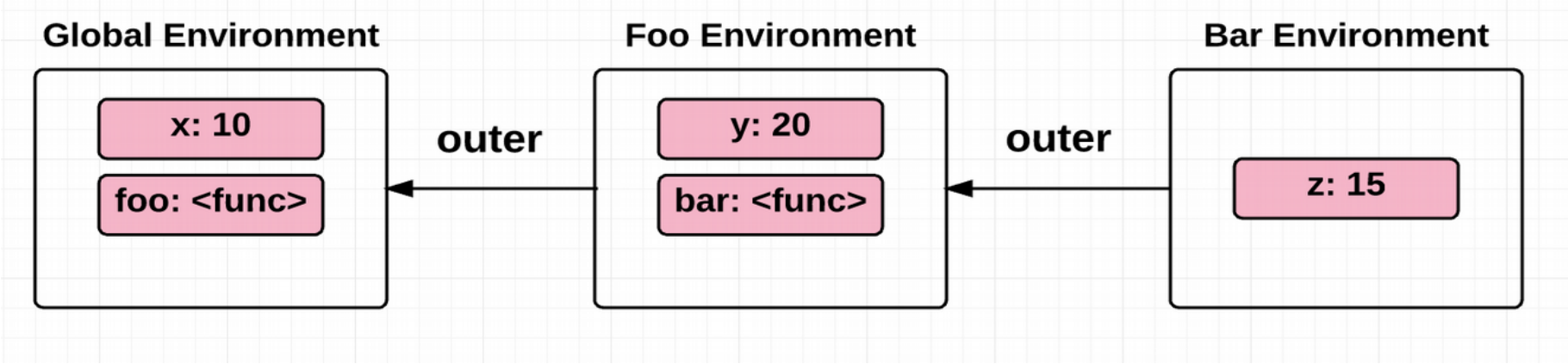
В процессе движения от родительского окружения к дочернему число доступных идентификаторов возрастает, и сами области видимости складываются в “цепочку областей видимости”.

Node.JS

```
var x = 10;

function foo() {
  var y = 20; // free variable
  function bar() {
    var z = 15; // free variable
    return x + y + z;
  }
  return bar;
}
```

Node.JS


Global Environment
x: 10
foo: <func>
outer
Foo Environment
y: 20
bar: <func>
outer
Bar Environment
z: 15

Node.JS

Иерархия цепочки областей видимости (т.е. цепочки окружений, связанных с функциями) сохраняется в момент создания функции перед началом интерпретации скрипта.

То есть иерархия определена статически расположением в исходном коде (это также называется “лексическим связыванием”, англ. lexical scoping).

Замыкания (Closures)

Каждая функция имеет контекст выполнения, который включает в себя окружение, которое определяет набор переменных в функции и хранит ссылку на своё родительское окружение.

Ссылка на родительское окружение делает все переменные из родительских областей видимости доступными для вложенных функций, независимо от того, была ли функция вызвана кодом внутри или снаружи данной области видимости

Замыкания (Closures)

Таким образом, функция “запоминает окружение (или область видимости, англ. scope), поскольку функция фактически сохраняет ссылку на окружение (и тем самым удерживает в памяти таблицу переменных функции).

Node.JS

```
var x = 10;

function foo() {
  var y = 20; // free variable
  function bar() {
    var z = 15; // free variable
    return x + y + z;
  }
  return bar;
}

var test = foo();

test(); // 45
```

Замыкания (Closures)

Таким образом, функция “запоминает окружение (или область видимости, англ. scope), поскольку функция фактически сохраняет ссылку на окружение (и тем самым удерживает в памяти таблицу переменных функции).

Node.JS

```
var result = [];

for (var i = 0; i < 5; i++) {
  result[i] = function () {
    console.log(i);
  };
}

result[0](); // 5, expected 0
result[1](); // 5, expected 1
result[2](); // 5, expected 2
result[3](); // 5, expected 3
result[4](); // 5, expected 4
```

Node.JS

Каждое обновление переменной i обновляет область видимости, которую делят между собой все пять анонимных функций.

Это можно исправить путём добавления нового контекста исполнения, в котором копия значения i поступит извне как параметр функции.

Node.JS

```
var result = [];

for (var i = 0; i < 5; i++) {
  result[i] = (function inner(x) {
    // additional enclosing context
    return function() {
      console.log(x);
    }
  })(i);
}
```

Node.JS

```
var result = [];

for (let i = 0; i < 5; i++) {
  result[i] = function () {
    console.log(i);
  };
}
```

Node.JS

Счетчики коллбеков.
Повесив функции обратного вызова к определенным событиям, вы не можете гарантировать порядок их выполнения. Так что, если вам необходимо дождаться выполнения нескольких запросов, то самый простой способ решения такой задачи — считать каждую завершенную операцию, и, таким образом, проверять, все ли необходимые операции были завершены.

Node.JS

Счетчики коллбеков.

Это пригодиться, если обязательно нужно дождаться результатов.

Например, считая количество выполненных запросов к базе данных в коллбеке, мы можем определить, когда все наши запросы будут выполнены, и только тогда пойти дальше. Запросы к базам данных запустятся параллельно, потому что I/O библиотека поддерживает это.

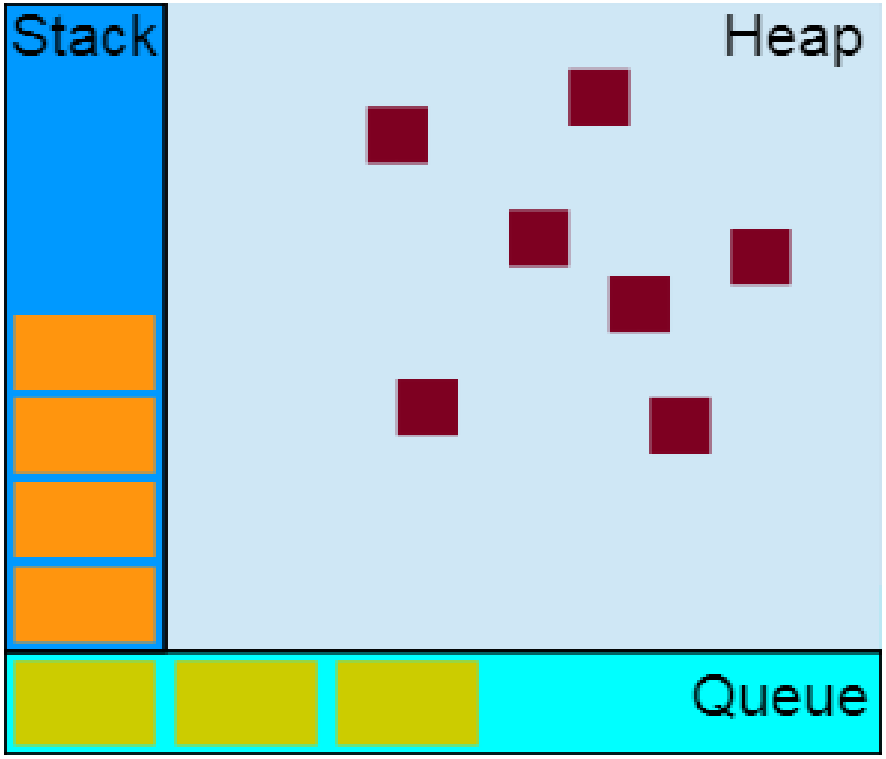
Node.JS

Событийные циклы.

Вы можете обернуть блокирующий код в событийную абстракцию, запустив его в дочернем процессе, и забрать данные из этого процесса по окончанию его работы.

Node.JS

Концепция жизненного цикла



Node.JS

Стек

Вызов любой функции создает контекст выполнения. При вызове вложенной функции создается новый контекст, а старый сохраняется в специальной структуре. Так формируется стек контекстов

Куча

Объекты размещаются в куче. Куча — это ссылка на определённую неструктурированную область памяти.

Node.JS

Очередь событий — это список событий, подлежащих обработке. Каждое событие ассоциируется с некоторой функцией.

Когда на стеке освобождается достаточно места, событие извлекается из очереди и обрабатывается.

Обработка события состоит в вызове этой функции (и, таким образом, создании начального контекста выполнения). Обработка события заканчивается, когда стек снова становится пустым.