## **Источник**

Видео: https://www.youtube.com/watch?v=tDbRVZqwxn4&feature=emb\_title

Текст: https://habr.com/ru/company/oleg-bunin/blog/433318/

Основные выводы: при MPA можно вообще не думать об оптимизации для сборки мусора, использовать Performance и Memory с консоли разраб для выявления нагрузок, если что-то чрезмерно нагружает, то искать возможности оптимизации.

## **Зачем нужна сборка мусора?**

Рассмотрим на примере Яндекс.Карт. Яндекс.Карты — это огромный и сложный сервис, который использует много JS и практически все существующие браузерные API, кроме мультимедийных, а среднее время сессии 5–10 минут. Обилие JavaScript создает множество объектов. Перетаскивание карты, добавление организаций, поисковая выдача и много других событий, происходящих каждую секунду, создает лавину объектов. Добавьте к этому React и объектов становится еще больше.  
  
Однако, JS-объекты занимают на карте всего 30–40 Мб. Для долгих сессий Яндекс.Карт и постоянного выделения новых объектов это мало.

Причина небольшого объема объектов в том, что они успешно собираются сборщиком мусора и память используется заново.

Сегодня мы поговорим про сборку мусора с четырех сторон:

* **Теория**. Начнем с неё, чтобы говорить на одном языке и понимать друг друга.
* **Суровая реальность.** В конечном итоге компьютер исполняется машинный код, в котором нет всех привычных нам абстракций. Попробуем разобраться, как сборка мусора работает на низком уровне.
* **Браузерная реальность.** Посмотрим, как сборка мусора реализована в современных движках и браузерах, и какие мы можем сделать из этого выводы.
* **Повседневность** — поговорим о практическом применении полученных знаний в повседневной жизни.

Все утверждения подкрепим примерами, как можно и как не нужно делать.

## **Зачем все это знать?**

Сборка мусора — незаметная для нас вещь, однако зная как она устроена вы будете:

* Иметь представление об инструменте, который используете, что полезно в работе.
* Понимать, где оптимизировать уже выпущенные приложения и как спроектировать будущие, чтобы они работали лучше и быстрее.
* Знать, как не совершать распространенные ошибки и перестать тратить ресурсы на бесполезные и вредные «оптимизации».

## **Теория**

Джоэл Спольски однажды сказал:

Все нетривиальные абстракции дырявы.

Сборщик мусора — это одна большая нетривиальная абстракция, которую латают со всех сторон. К нашему счастью, она течет очень редко.  
  
Давайте начнем с теории, но без скучных определений. Разберем работу сборщика на примере простого кода:

window.Foo = **class** **Foo** {

**constructor**() {

**this**.x = { y: 'y’ };

}

work(name) {

let z = 'z';

return function () {

console.log(name, this.x.y, z);

this.x = null;

}.bind(this);

}

};

* В коде есть **класс**.
* У класса есть **constructor**.
* **Метод work** возвращает связанную функцию.
* Внутри функции используется **this** и пара переменных из замыкания.

Посмотрим, как этот код будет себя вести, если мы будем его запускать таким образом:

**var** foo = **new** Foo(); //Cоздаем объекта класса

window.worker = foo.work('Brendan Eich'); // Возьмем функцию полученную от bind, вызвав метод

window.foo = null; // Обнулим все

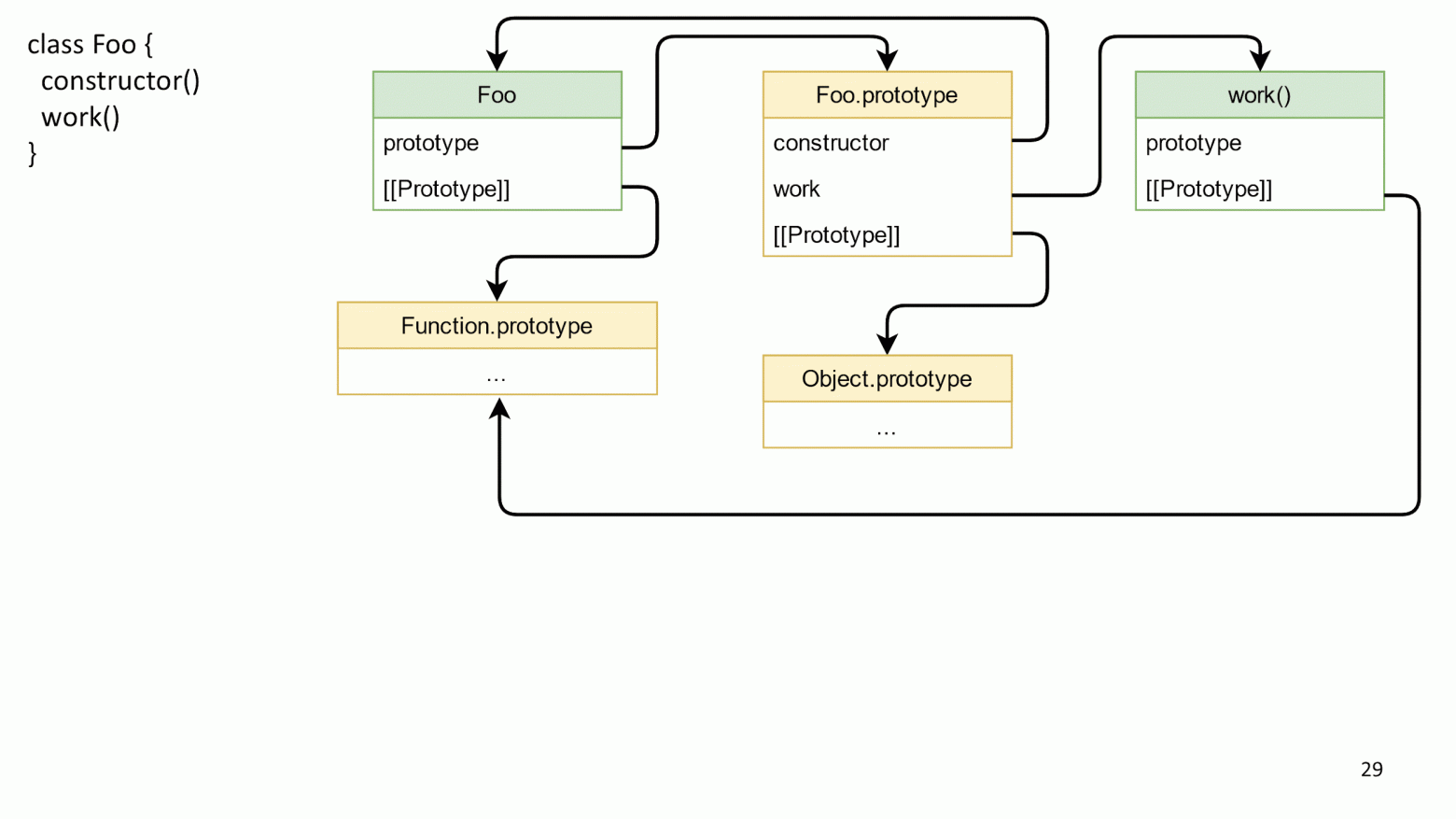
window.Foo = null; // Вызовем метод, который тоже что-то обнулит

window.worker();

window.worker = null; // Обнулим вообще все, что осталось

Разберем код и его составляющие подробнее и начнем с класса.

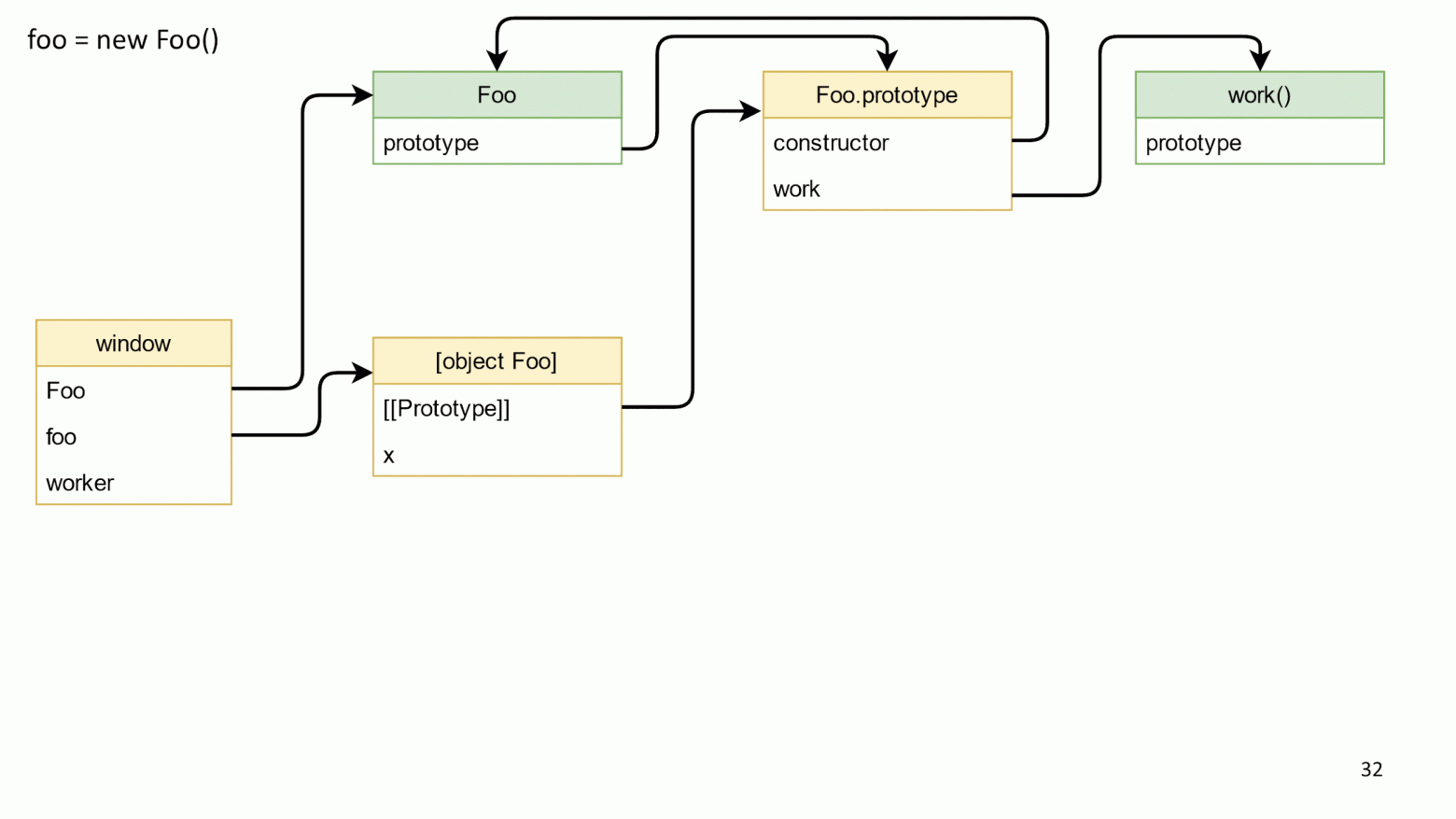
### **Объявление класса**

  
  
Можно считать, что классы в ECMAScript 2015 — это просто синтаксический сахар для функций. У всех функций есть:

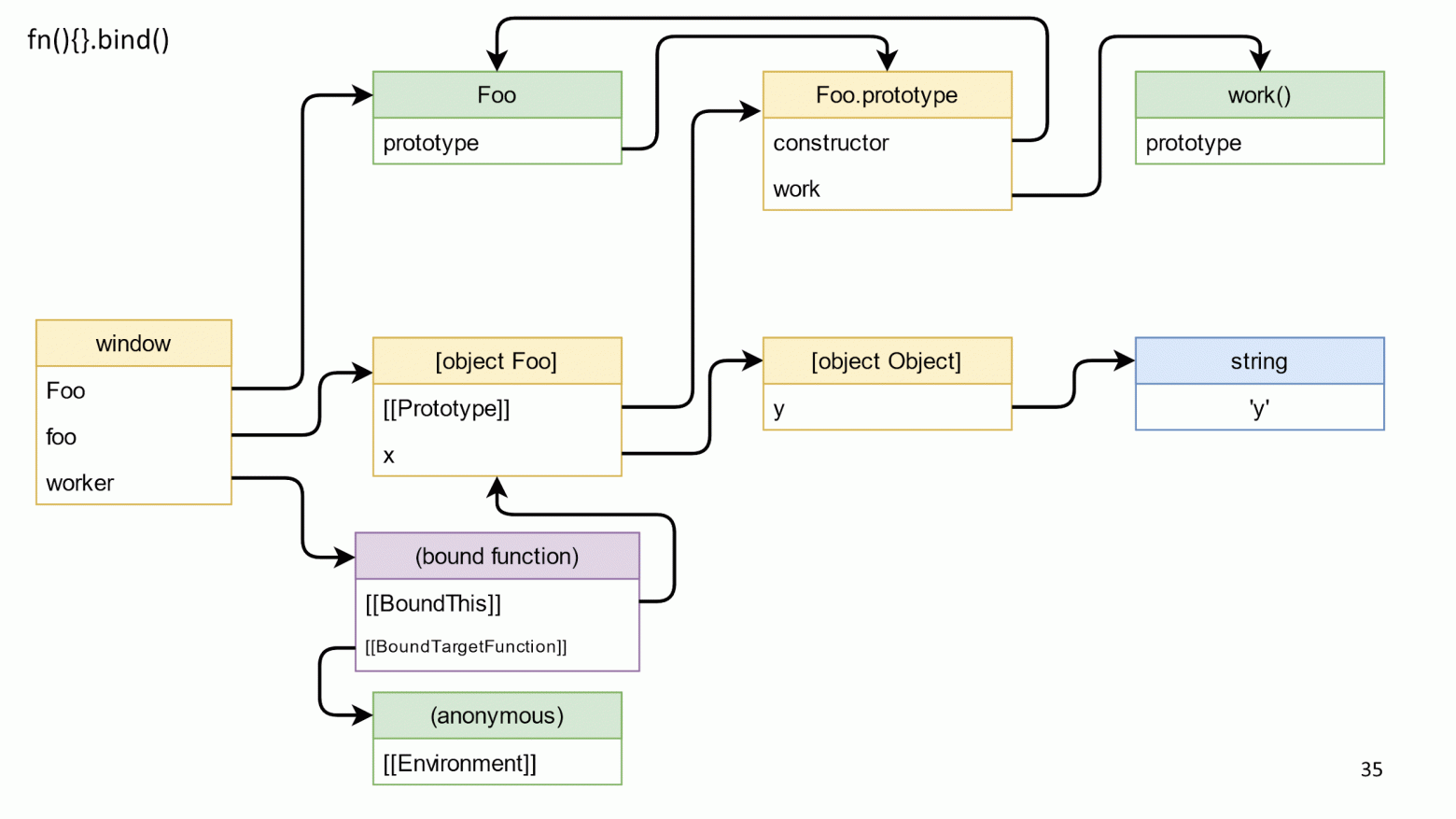
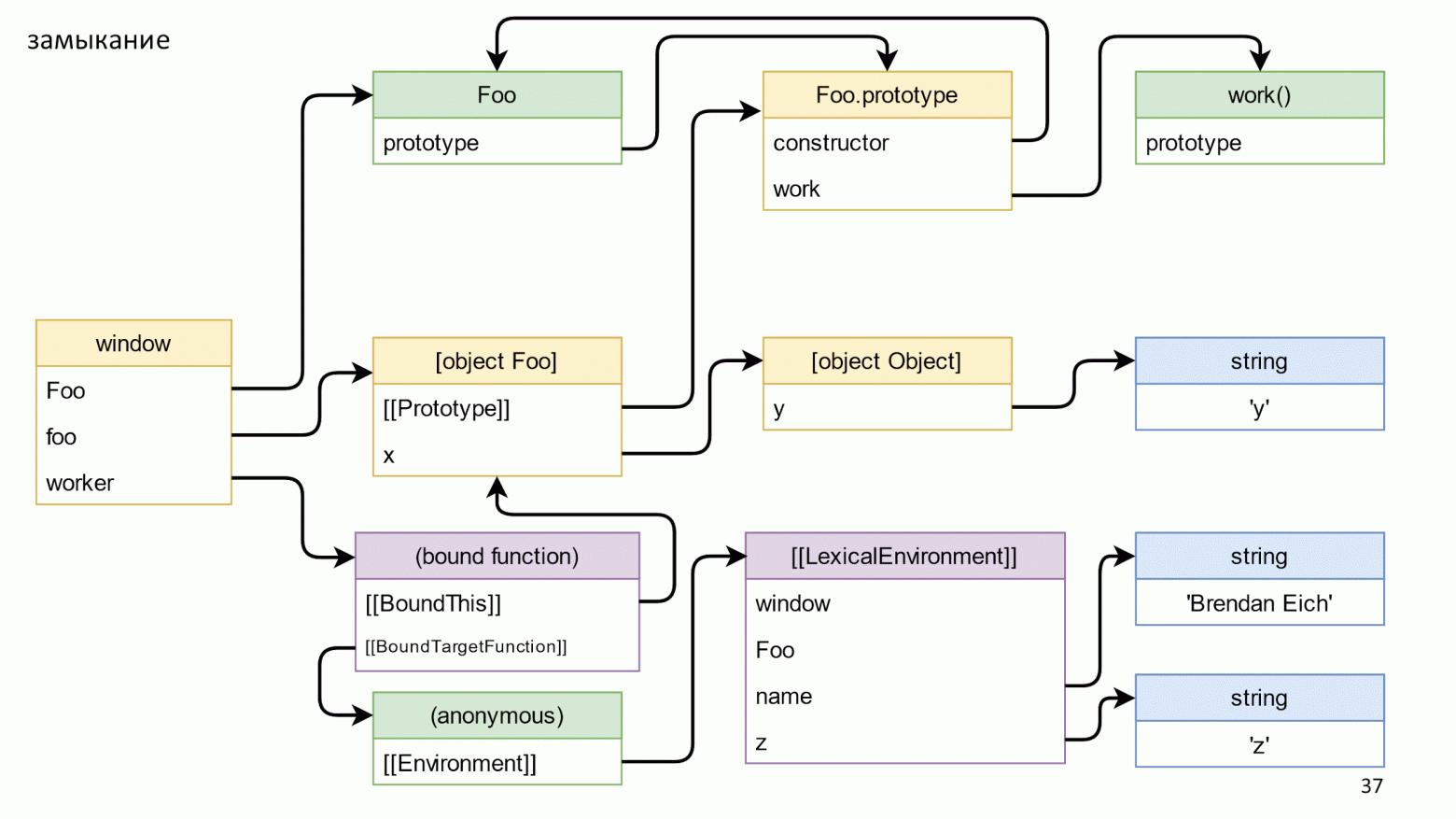
* **Function.[[Prototype]]** — реальный прототип функции.
* **Foo.prototype**— прототип для свежесозданных объектов.
* У Foo.prototype есть обратная ссылка на конструктор через поле constructor. Это объект, поэтому он наследует **Object.prototype**.
* **Метод work** отдельная функция на которую есть ссылка, похож на constructor, потому что они оба просто функции. Ему тоже можно задать прототип и вызывать его через new, но таким поведением редко кто пользуется.

Прототипы занимают очень много места на схеме, поэтому давайте запомним, что они есть, но далее уберем для простоты.

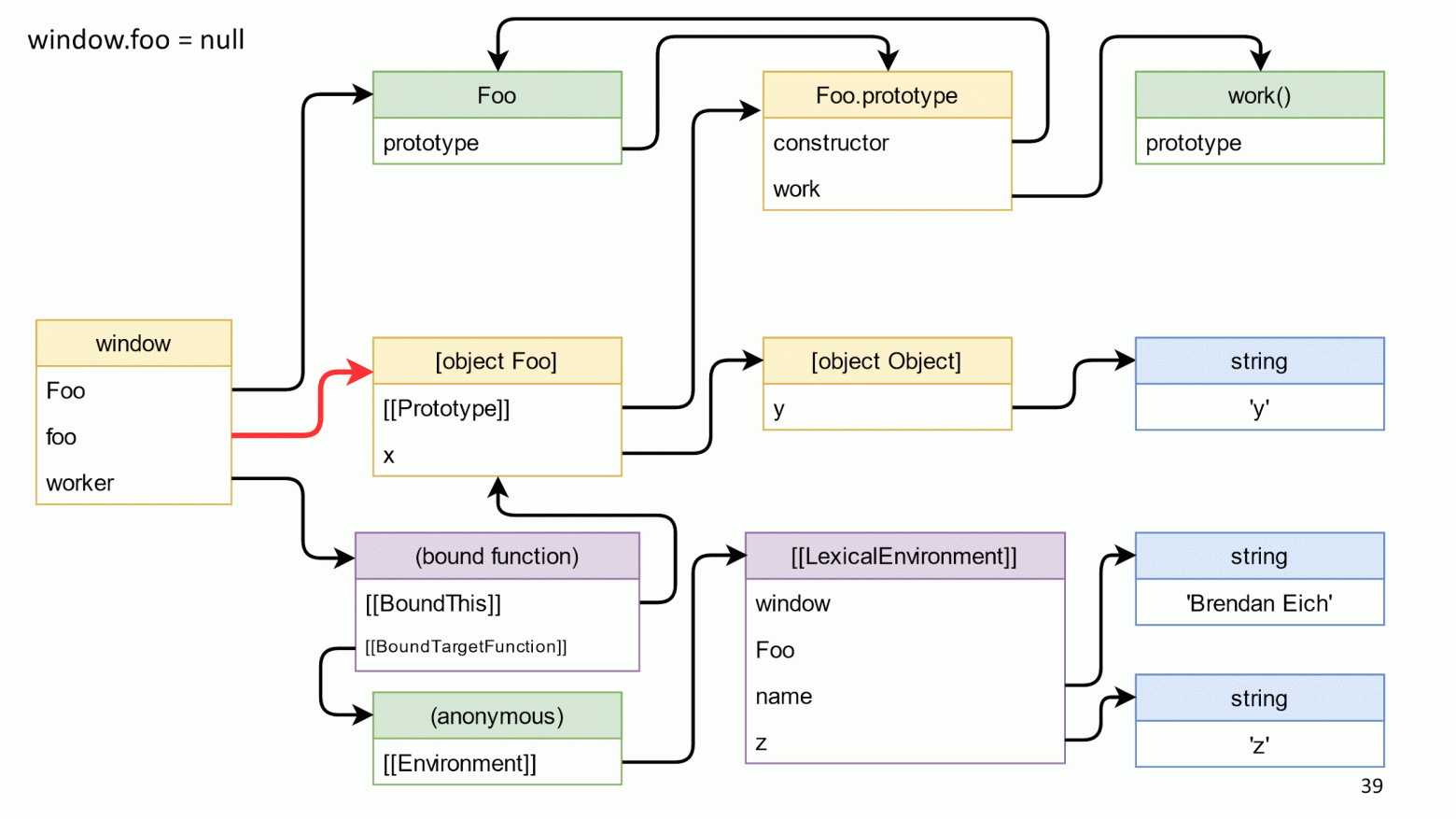
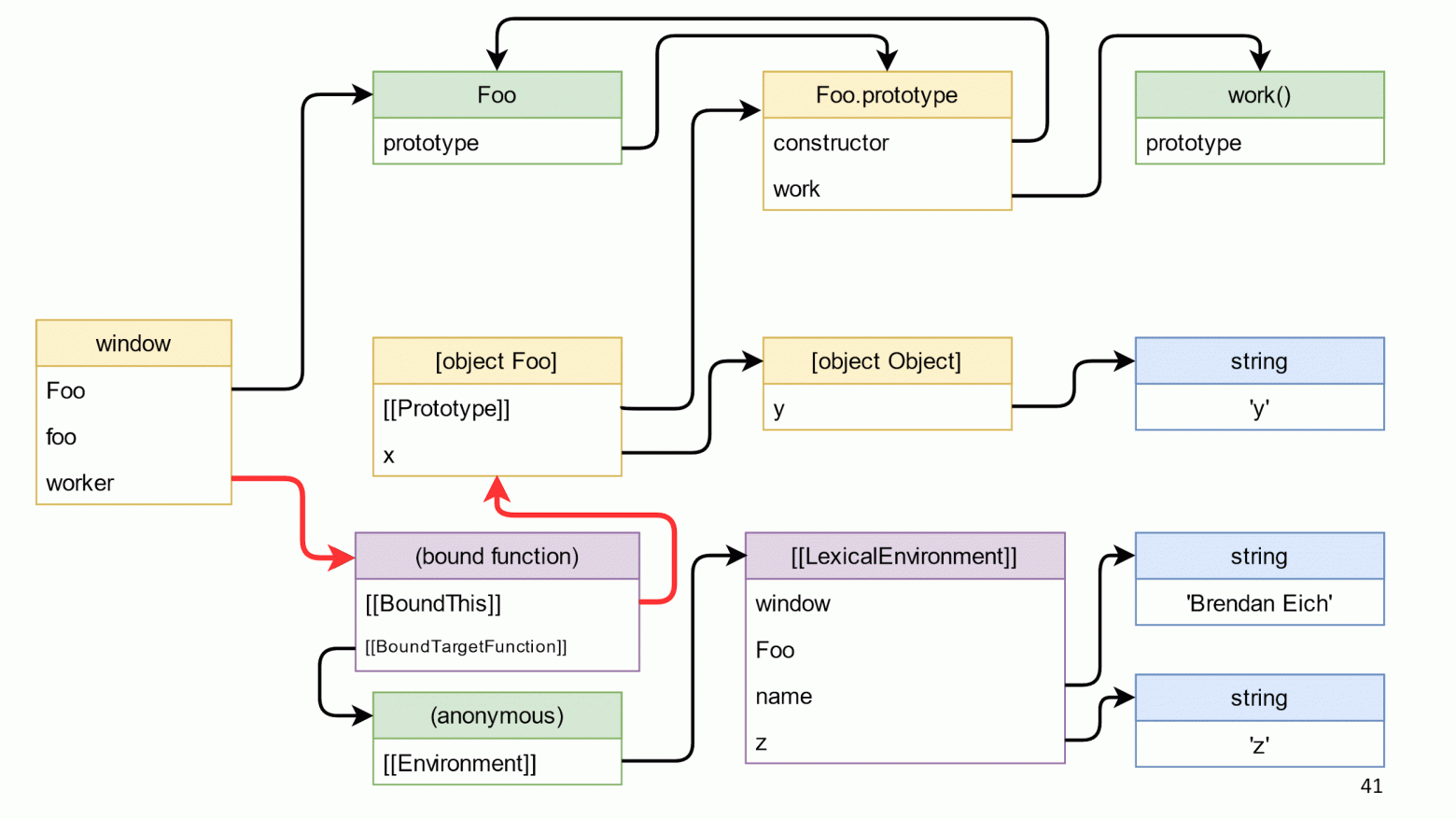
### **Создание объекта класса**



* Кладем наш класс в window, потому что классы по умолчанию туда не попадают.
* Создаем объект класса.
* Создание объекта автоматически выставляет прототип у объекта класса в Foo.prototype. Поэтому, когда вы попытаетесь вызвать на объекте метод work, он будет знать о каком work речь.
* Наш конструктор создает в объекте поле x из объекта со строкой.

Вот что получилось:  
  
  
  
Метод возвращает связанную функцию — это такой специальный «магический» объект в JS, который состоит из связанного this и функции, которую надо вызывать. У связанной функции тоже есть прототип и другой прототип, но нам интересно замыкание. По спецификации замыкание хранится в Environment. Скорее всего вам привычнее слово Scope, но **в спецификациях поле называется именно Environment**.  
  
  
  
В Environment хранится ссылка на LexicalEnvironment. Это сложный объект, сложнее чем на слайде, в нем хранятся ссылки на все, к чему можно обратиться из функции. Например, window, Foo, name и z. Там же хранятся ссылки даже на то, что вы явно не используете. Например, вы можете, применить eval и случайно использовать неиспользуемые объекты, но JS при этом не должен ломаться.  
  
Итак, мы построили все объекты и сейчас будем все рушить.

### **Удаляем ссылку на объект**

Начнем с удаления ссылки на объект, эта ссылка на схеме выделена красным.  
  
  
  
Удаляем и ничего не происходит, потому что от **window** до объекта есть путь через **bound function** функцию.  
  
  
  
Это подталкивает нас к типичной ошибке.

### **Типичная ошибка — забытая подписка**

externalElement.addEventListener('click', () => {

**if** (**this**.shouldDoSomethingOnClick) {

**this**.doSomething();

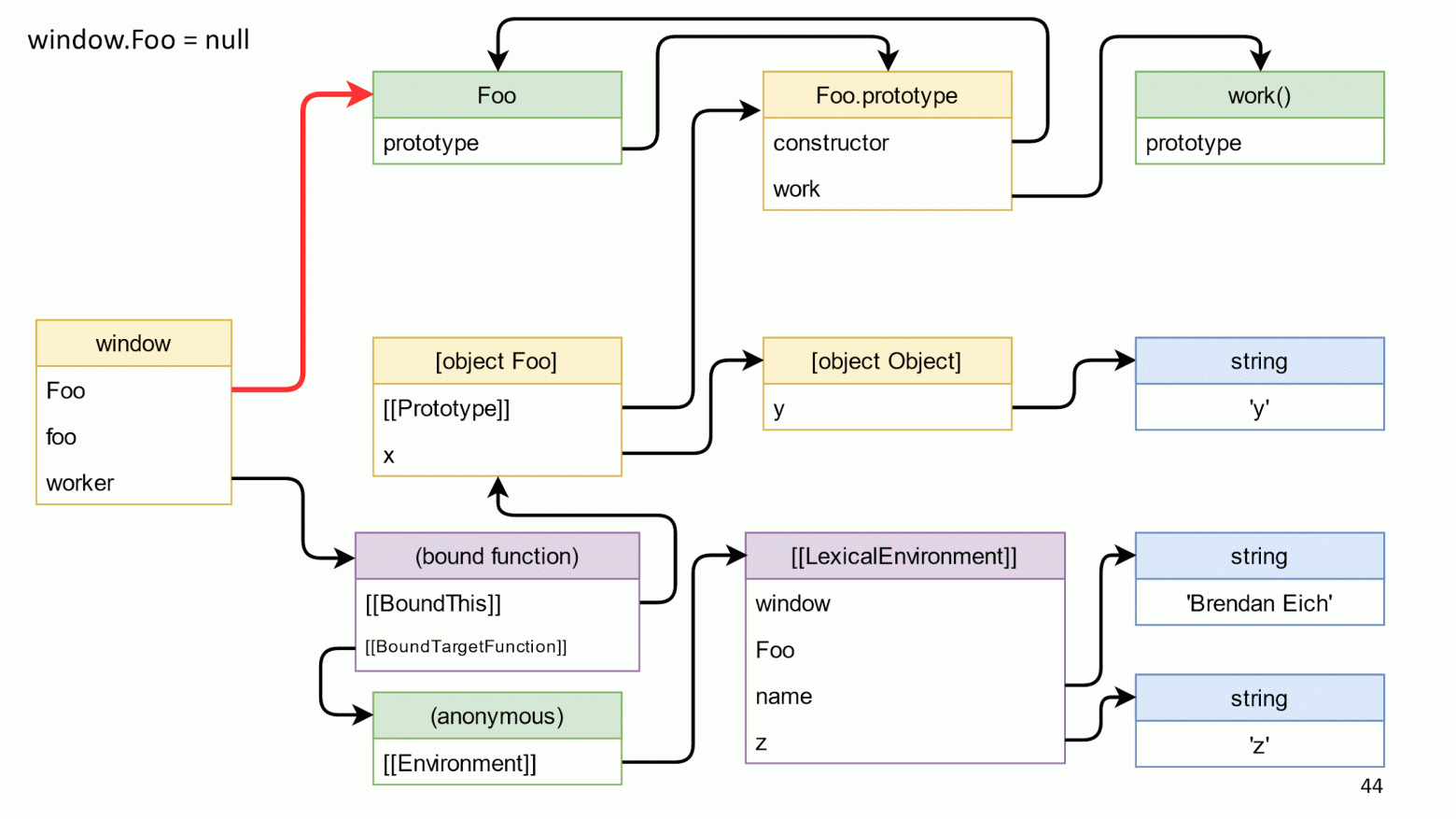
}

})

Возникает, когда вы подписываетесь: используя **this,** явно через bind или через стрелочные функции; используете что-то в замыкании. Потом забываете отписаться, и время жизни вашего объекта или того, что есть в замыкании, становится таким же, как время жизни подписки. Например, если это элемент DOM, который вы не трогаете руками, то, скорее всего, это время до конца жизни страницы.  
  
Для решения это проблемы:

* Отписывайтесь.
* Продумайте время жизни подписки, и кто ей владеет.
* Если по каким-то причинам вы не можете отписаться, то занулите ссылки (whatever = null), или почистите все поля у объекта. Если у вас утечет объект, он будет маленький и его не жалко.
* Используйте WeakMap, возможно это поможет в каких-то ситуациях.

### **Удаляем ссылку на класс**

Идем дальше и попробуем удалить подсвеченную красным ссылку на класс.  
  
  
  
Удаляем ссылку и у нас ничего не меняется. Причина в том, что класс доступен через BoundThis, в котором есть ссылка на прототип, а в прототипе есть ссылка обратно на constructor.

### **Типичная ~~ошибка~~ бесполезная работа**

Зачем нужны все эти демонстрации? Потому, что существует обратная сторона проблемы, когда люди воспринимают совет занулять ссылки слишком буквально и зануляют вообще все подряд.

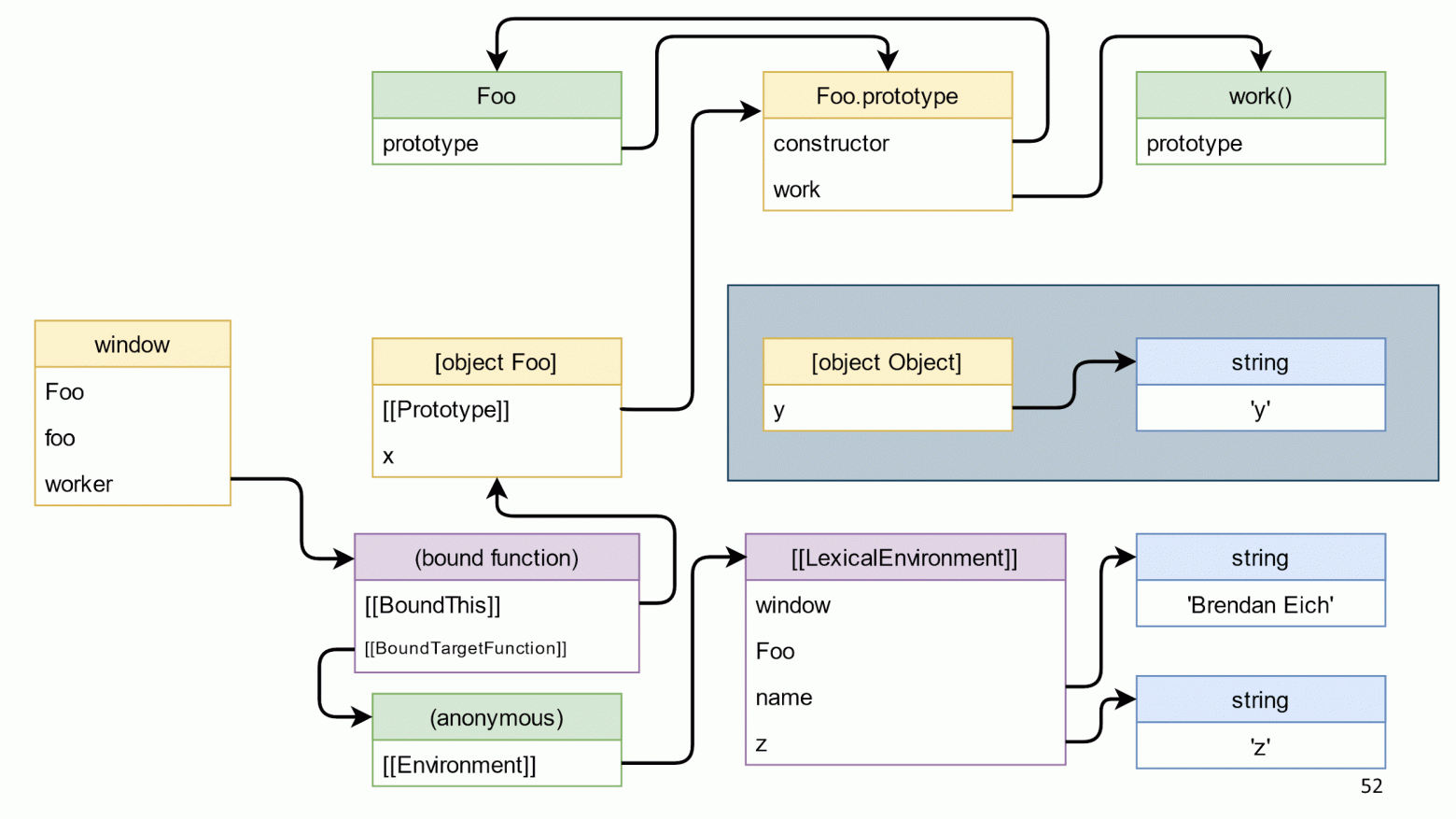
destroy() {

**this**.\_x = null;

**this**.\_y = null;

// еще 10 this.\_foobar = null

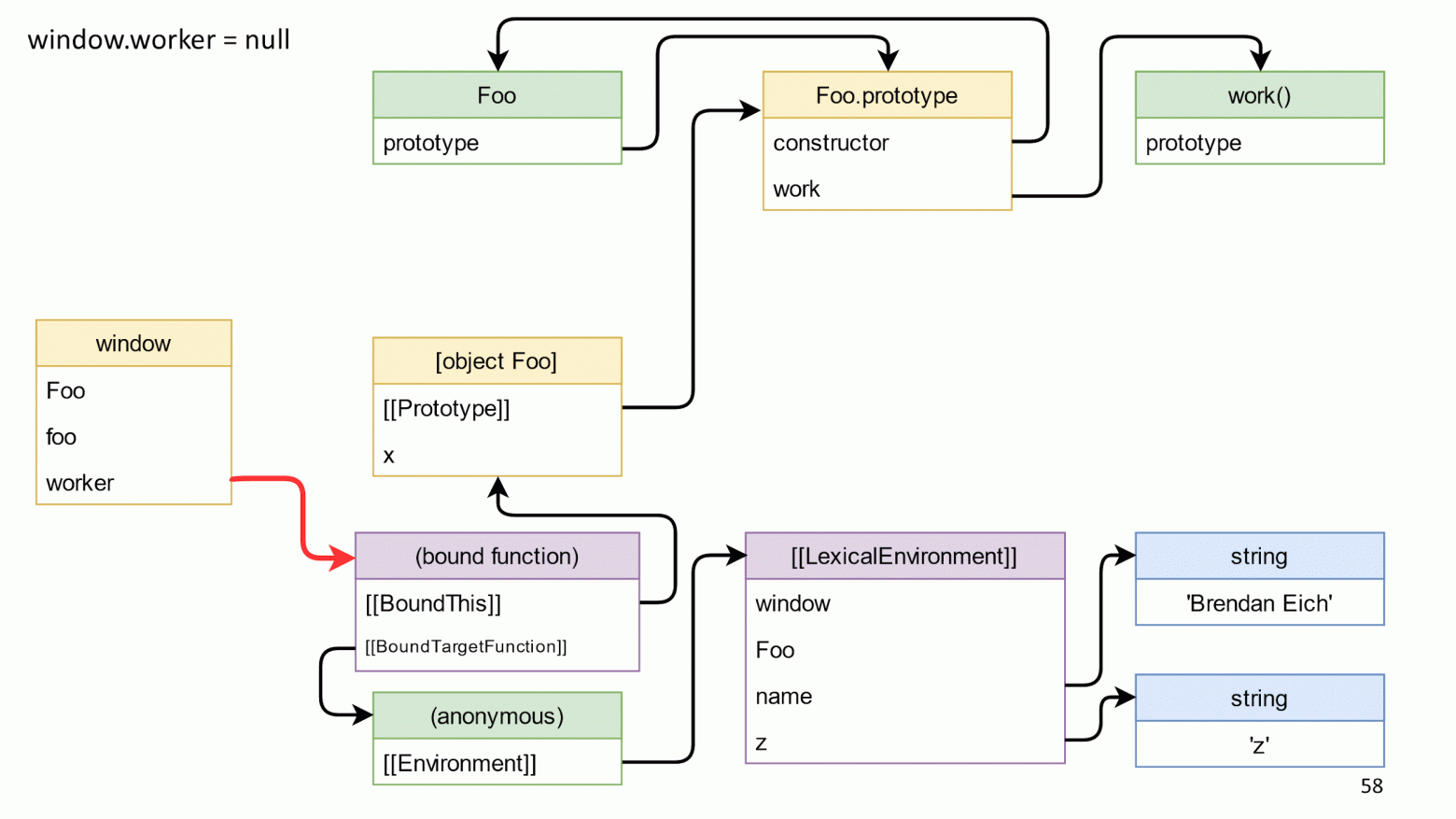
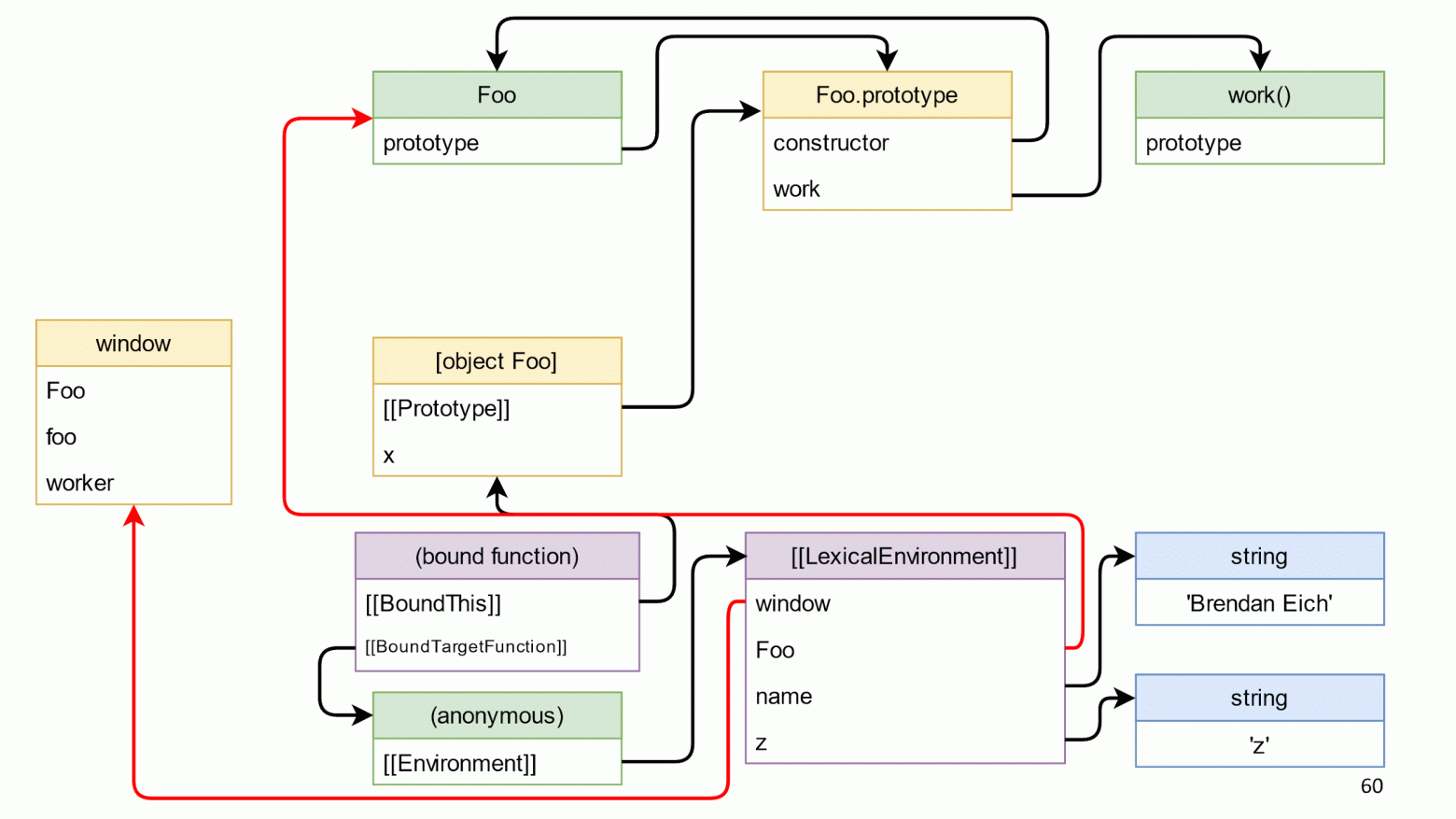
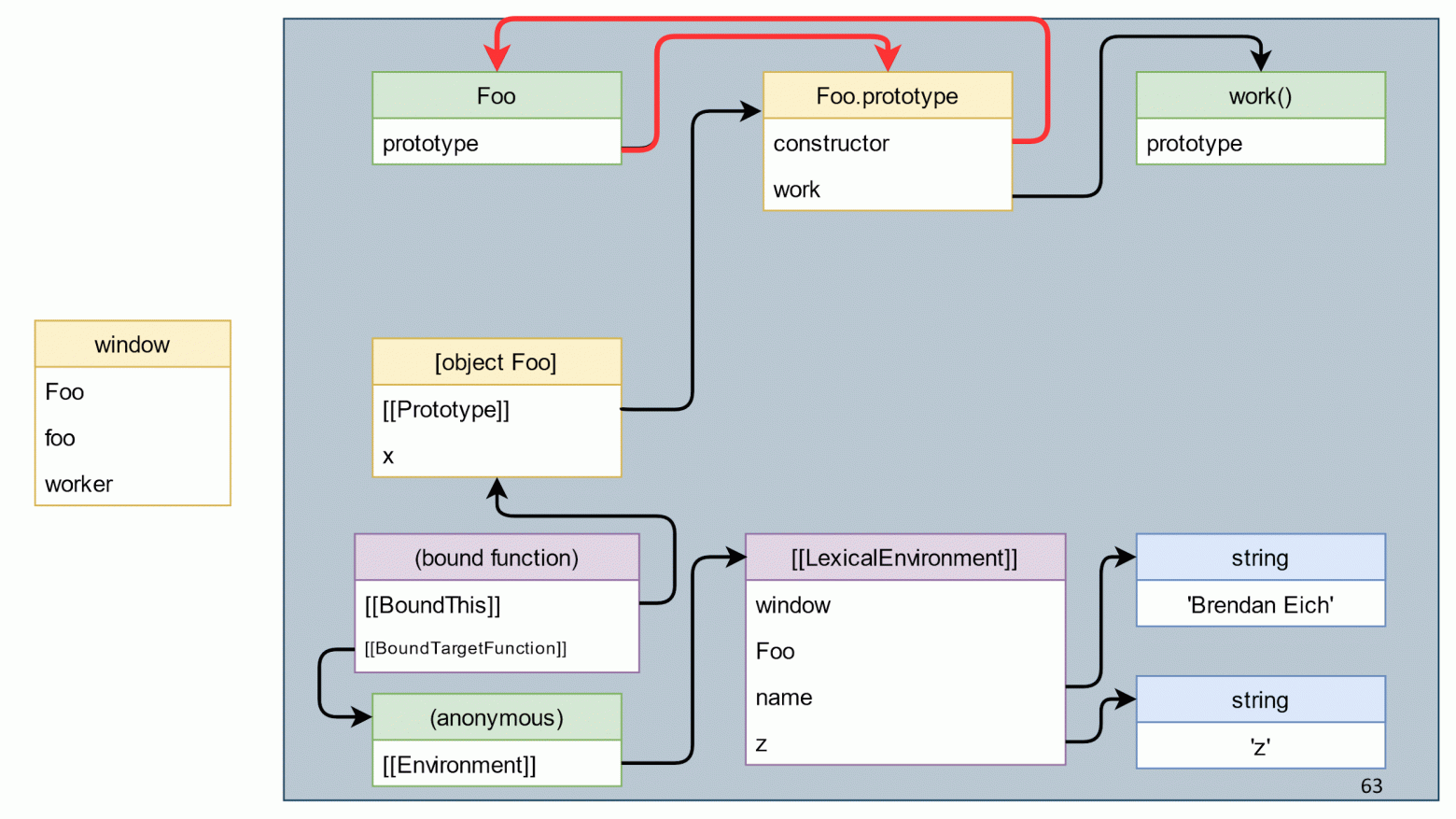
}

Это довольно бесполезная работа. Если объект состоит только из ссылок на другие объекты и там нет никаких ресурсов, то никакой destroy()не нужен. Достаточно потерять ссылку на объект, и он умрет сам по себе.  
  
Универсального совета нет. Когда надо — зануляйте, а когда не надо — не зануляйте. Зануление не ошибка, а просто бесполезная работа.  
  
Идем дальше. Вызовем метод bound function и он удалит ссылку от [object Foo] до [object Object]. Это приведет к тому, что в схеме появятся объекты, которые лежат особняком в синем прямоугольнике.  
  
  
  
Эти объекты — JS мусор. Он прекрасно собирается. Однако есть мусор, который не поддается сборщику.

### **Мусор, который не собирается**

В многих браузерных API вы можете создать и уничтожить объект. Если объект не уничтожить, то никакой сборщик не сможет его собрать.  
  
Объекты с парными функциями create/delete:

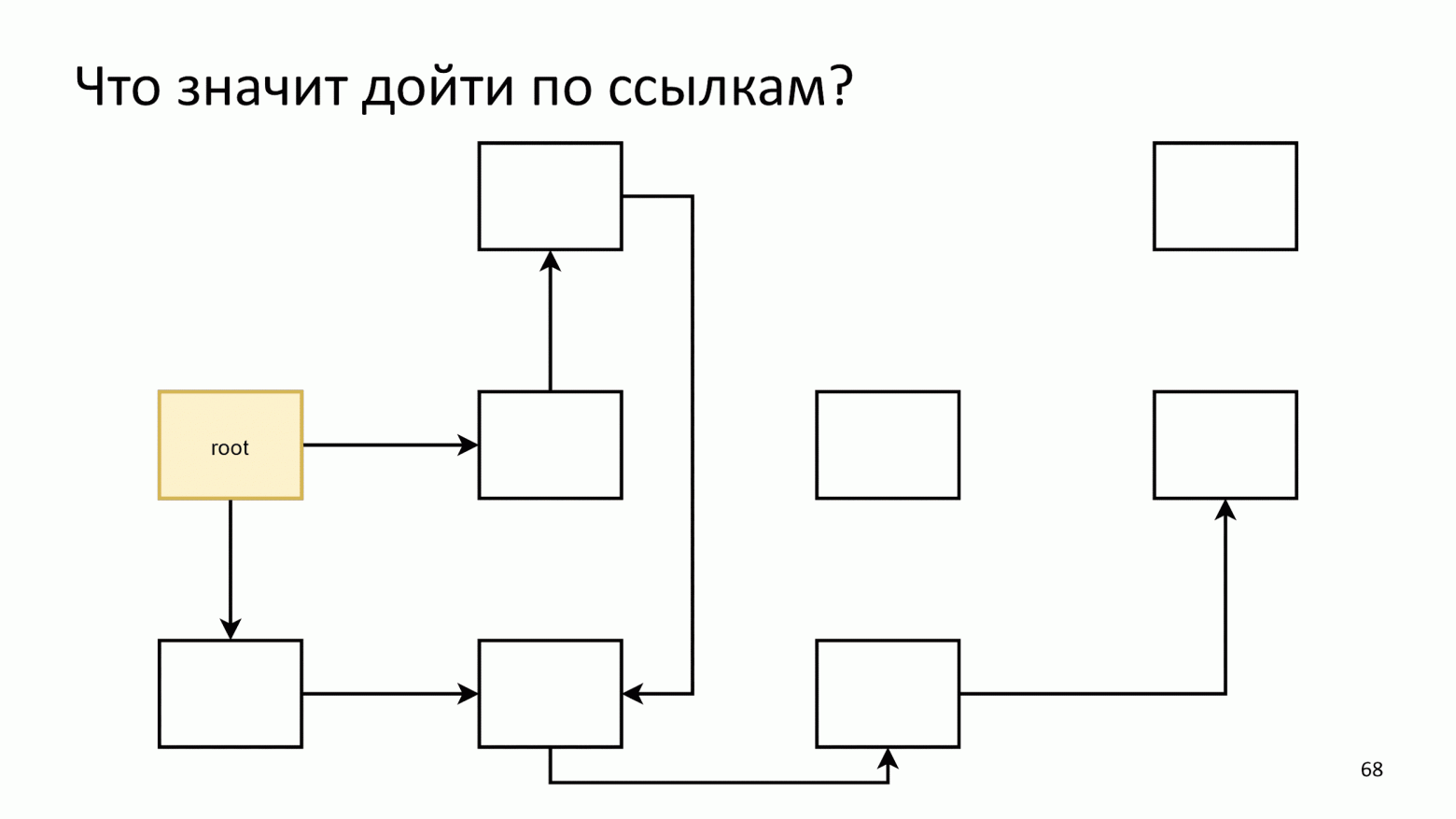
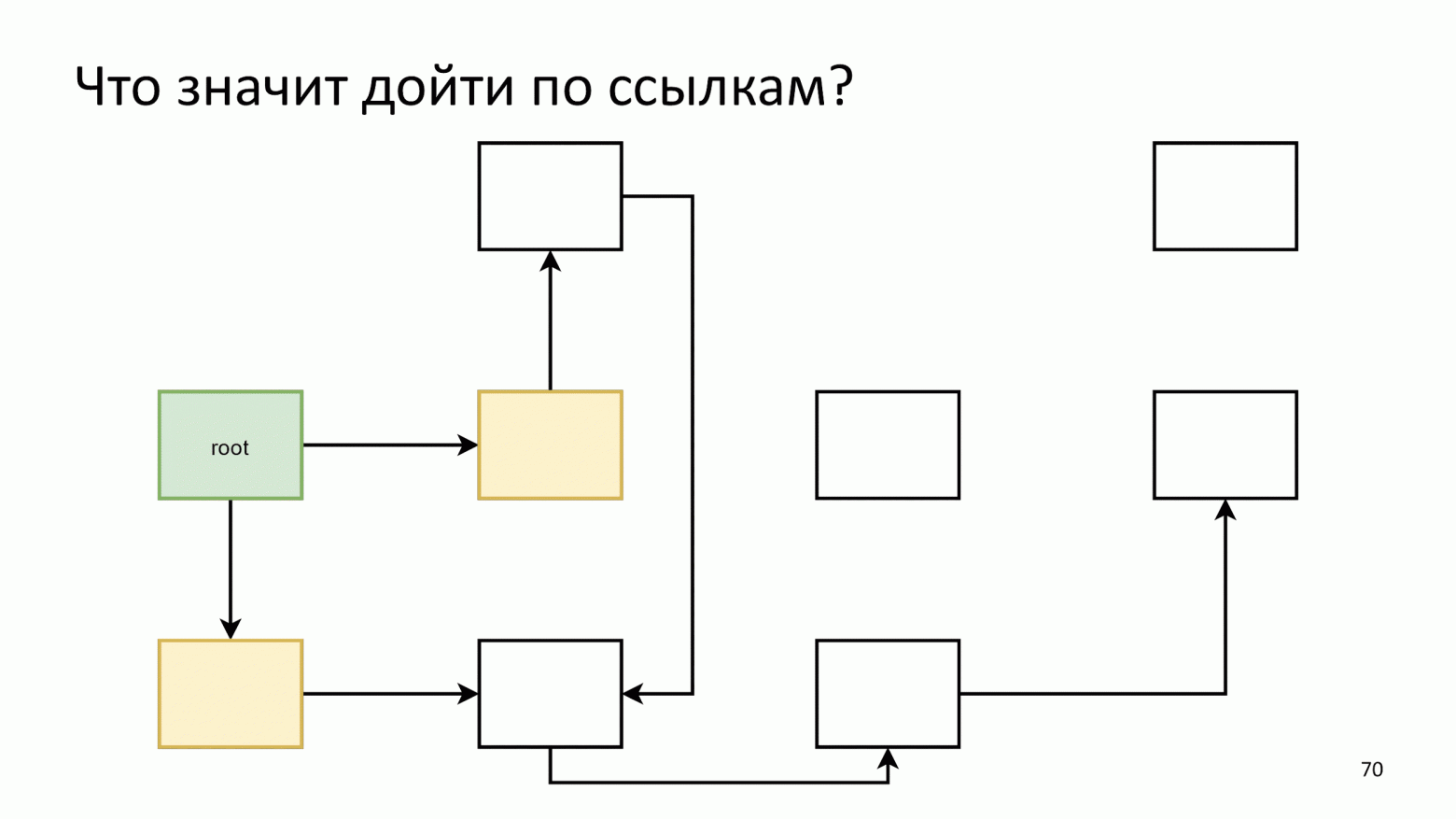
* createObjectURL(), revokeObjectURL();
* WebGL: create/delete Program/Shader/Buffer/Texture/etc;
* ImageBitmap.close();
* indexDb.close().

Например, если вы забудете удалить ObjectURL с видео на 200 Мб, то в памяти эти 200 Мб будут находиться до конца жизни страницы и даже дольше, потому что между вкладками есть обмен данными. Аналогично в WebGL, indexDb и другими браузерными API, с подобными ресурсами.  
  
К счастью, в нашем примере в синем прямоугольнике просто JavaScript объекты, поэтому это просто мусор, который можно удалить.  
  
Следующим этапом мы почистим последнюю ссылку слева направо. Это ссылка на метод, который мы получили, на связанную функцию.  
  
  
  
После ее удаления, у нас не останется ссылок между левой и правой частью? На самом деле, еще есть ссылки из замыкания.  
  
  
  
Важно, что ссылок слева направо нет, поэтому все, кроме window, — мусор, и он умрет.  
  
**Важное примечание**: в мусоре бывают циклические ссылки, то есть объекты, ссылающиеся друг на друга. Наличие таких ссылок ни на что не влияет, потому что сборщик мусора собирает не отдельные объекты, а весь мусор целиком.  
  
  
  
Мы посмотрели примеры и теперь на интуитивном уровне понимаем, что такое мусор, но давайте дадим полное определение понятия.

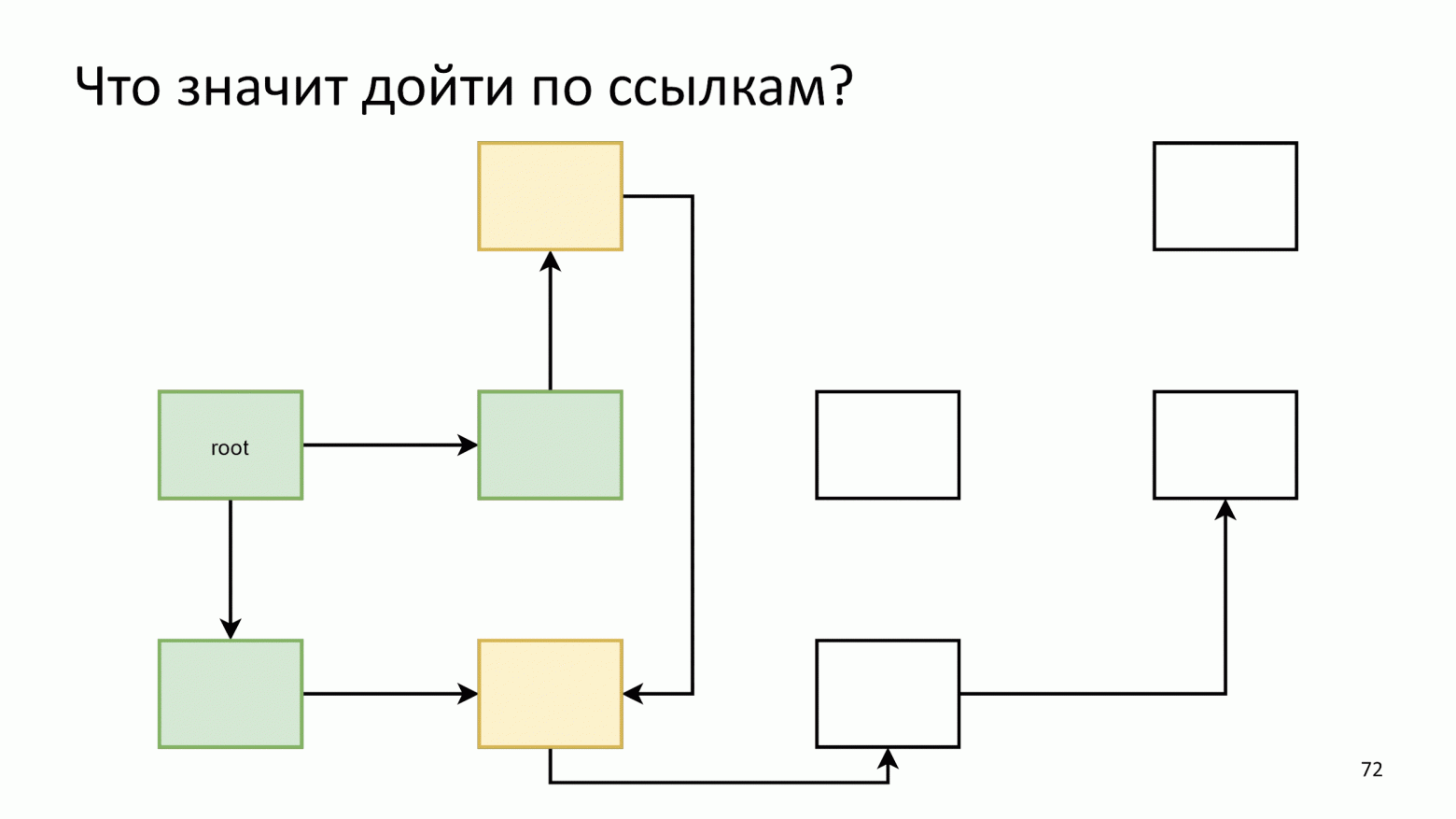
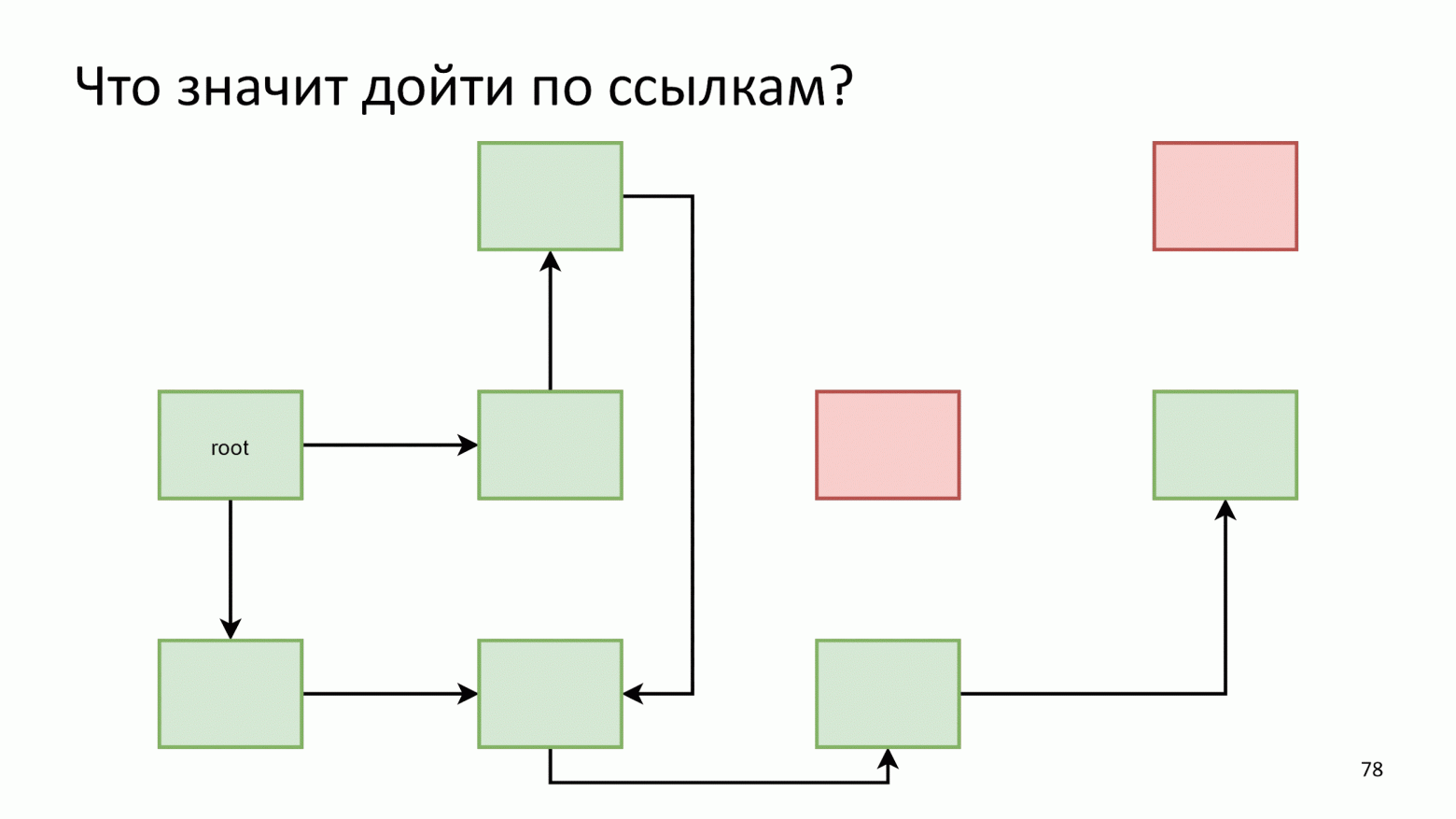
Мусор — всё, что не является живым объектом.

Все стало очень понятно. Но что такое живой объект?  
  
**Живой объект — такой объект, до которого можно дойти по ссылкам от корневого объекта.**  
  
Появляется два новых понятия: «дойти по ссылкам» и «корневой объект». Один корневой объект мы уже знаем — это window, поэтому давайте начнем со ссылок.

### **Что значит дойти по ссылкам?**

Есть множество объектов, которые связаны друг с другом и ссылаются друг на друга. Мы будем пускать по ним волну, начиная с корневого объекта.  
  
Инициализируем первый шаг, а затем действуем по следующему алгоритму: скажем, что все, что на гребне волны — живые объекты и посмотрим, на что они ссылаются.  
  
  
  
Инициализируем первый шаг. Затем будем действовать по следующему алгоритму: скажем, что все желтое на гребне волны — живые объекты и посмотрим, на что они ссылаются.  
  
То, на что они ссылаются, сделаем новым гребнем волны:  
  
  
  
Закончили и начинаем заново:

* Оживляем.
* Смотрим на что ссылаются.
* Создаем новый гребень волны, оживляем объекты.
* Смотрим на что ссылаются.

  
  
Заметив, что одна стрелка указывает на уже живой объект, мы просто ничего не предпринимаем. Далее по алгоритму, пока не закончатся объекты для обхода. Тогда мы говорим, что нашли все живые объекты, а все остальное — это мусор.  
  
  
  
Этот процесс называют **marking**.

### **Что значит корневой объект?**

* Window.
* Практически все браузерные API.
* Все promise.
* Всё, что кладется в Microtask и Macrotask.
* Mutation observers, RAF, Idle-callbacks. Все, до чего можно дойти из того, что лежит в RAF, нельзя удалять, потому что если удалить объект, который используется в RAF, то наверняка что-то пойдет не так.

Сборка может произойти в любой момент. Каждый раз, когда появляются фигурные скобки или function — создается новый объект. Памяти может не хватить, и сборщик пойдет искать свободную:

**function** **foo** (a, b, c) {

**function** **bar** (x, y, z) {

**const** x = {}; // nomem, run gc D:

// …

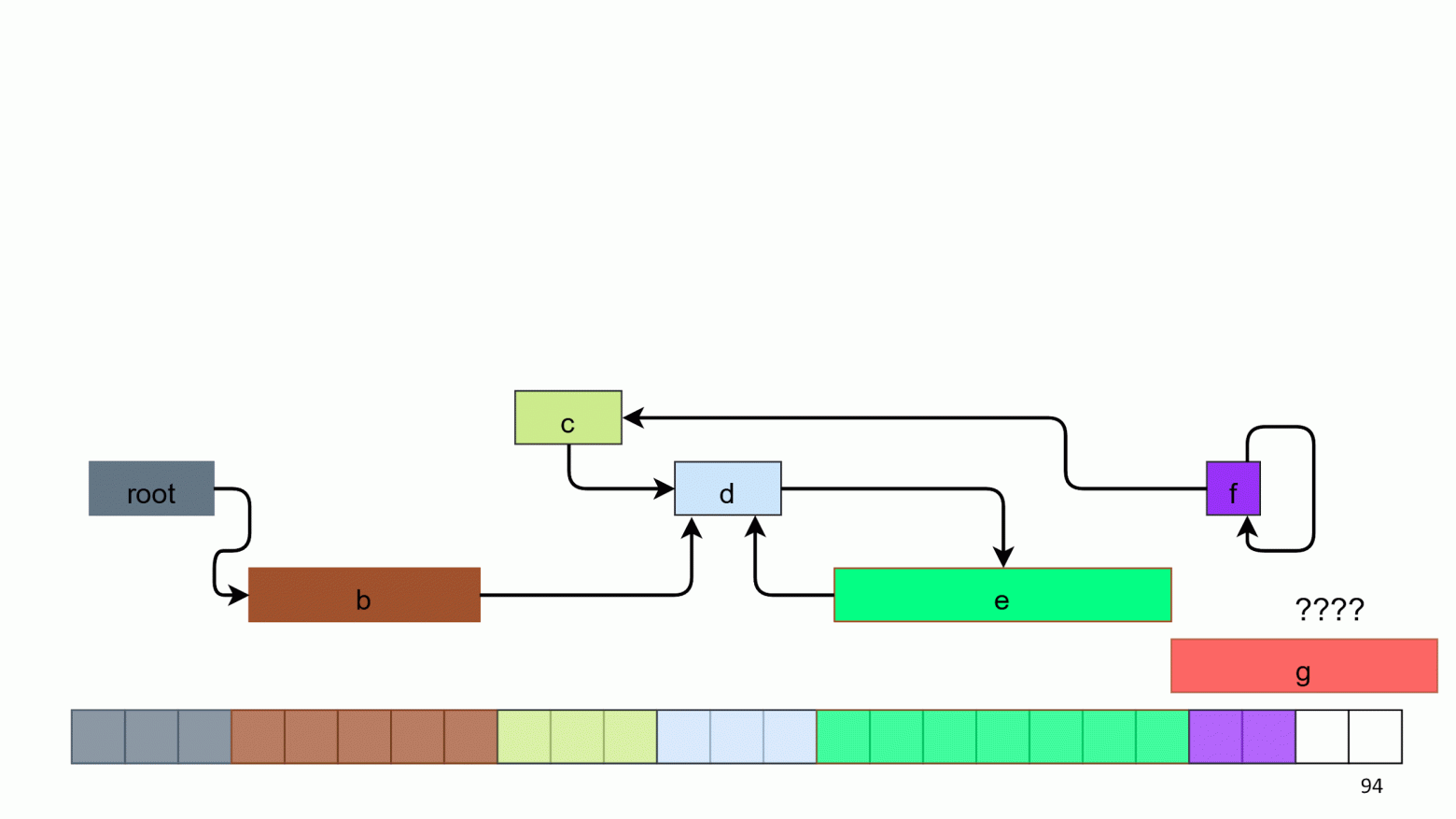
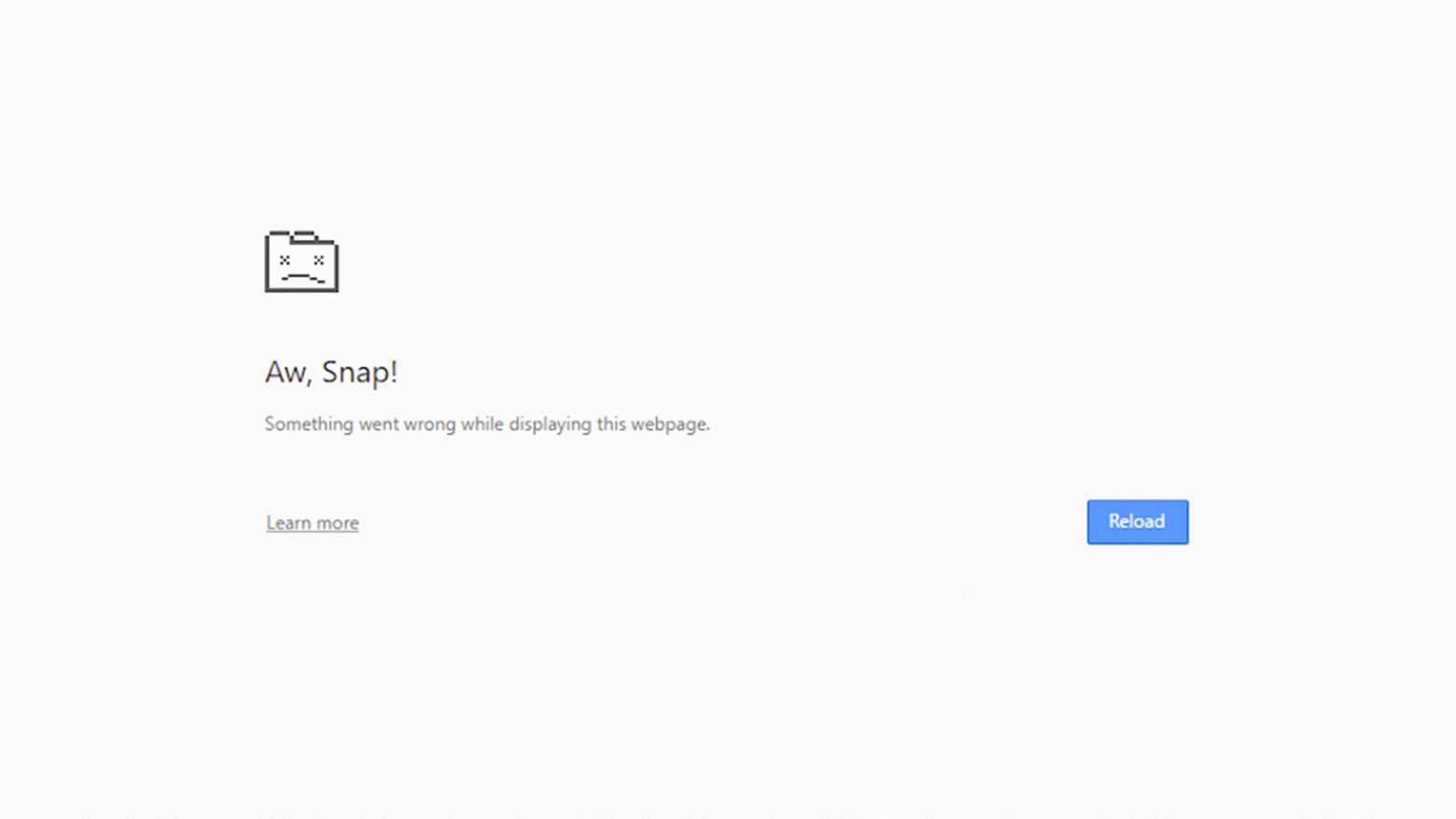
}

**while** (whatever()) bar();

}

В этом случае корневыми объектами будет все, что есть на стеке вызовов. Если вы, например, остановитесь на строчке с X и удалите то, на что ссылается Y, то ваше приложение упадет. JS не позволяет нам таких фривольностей, так что удалять объект из Y нельзя.  
  
Если предыдущая часть показалась сложной, то дальше будет еще сложнее.

## **Суровая реальность**

Поговорим про мир машин, в котором мы имеем дело с железом, с физическими носителями.  
  
Память — это один большой массив, в котором лежат просто числа, например: new Uint32Array(16 \* 2 \*\* 30).  
  
Давайте в памяти создадим объекты и будем их добавлять слева направо. Создаем один, второй, третий — они все разного размера. По пути проставляем ссылки.  
  
  
На седьмом объекте место закончилось, потому у нас 2 свободных квадратика, а надо 5.  
  
Что тут можно сделать? Первый вариант — аварийно завершить работу. На дворе 2018 год, у всех последние Макбуки и 16 Гб оперативки. Не бывает ситуаций, когда памяти нет!  
  
Однако, пускать все на самотек плохая идея, потому что в вебе это приводит к подобному экрану:  
  
  
  
Это не то поведение, которое мы хотим от программы, но в общем-то оно валидное. Есть категория сборщиков, которые называются **No-op**.

### **No-op collector**

Плюсы:

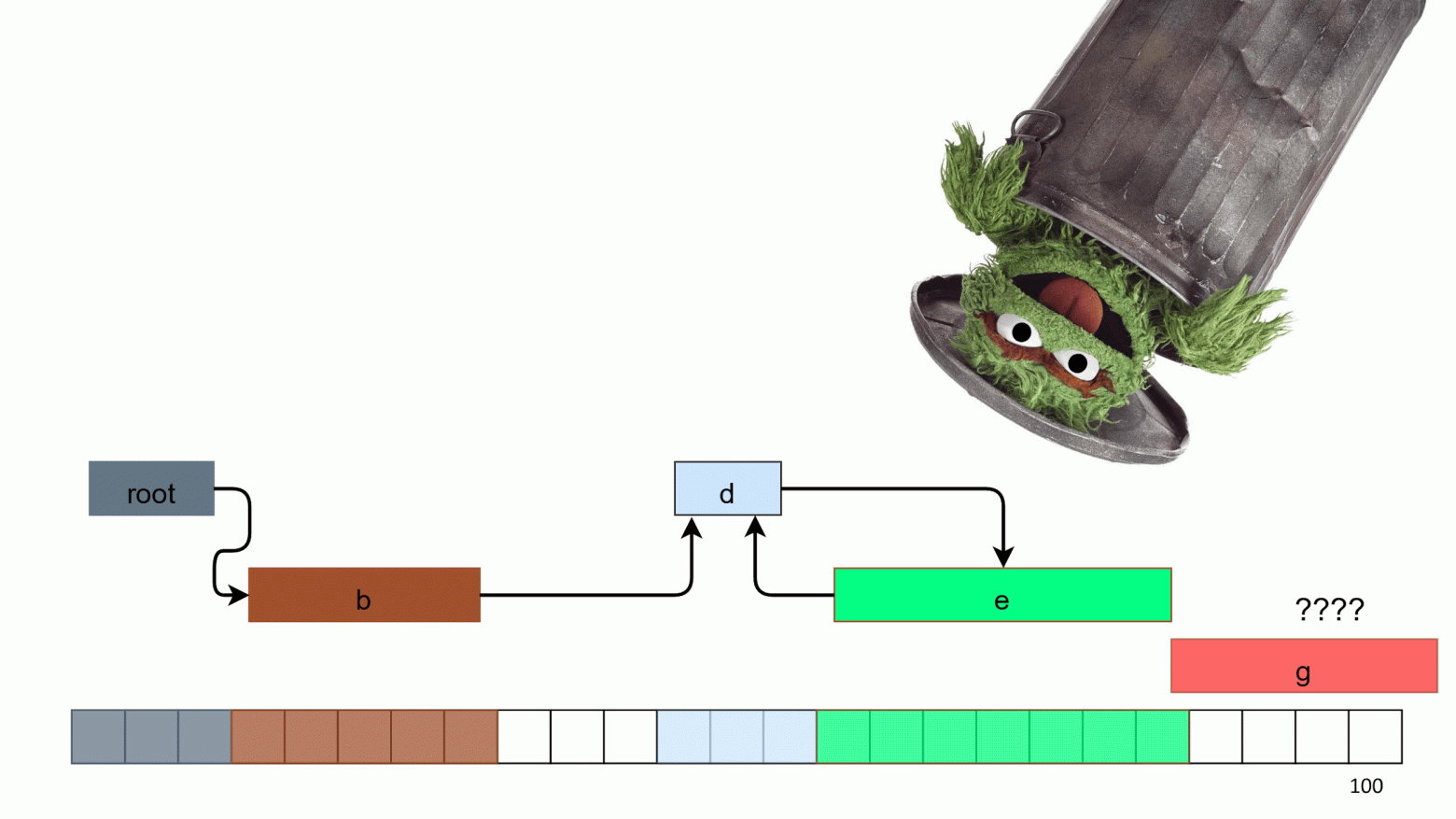
* Сборщик очень простой.
* Сборки мусора просто нет.
* Не надо ничего писать и думать о памяти.

Минусы:

* Все падает так, что больше никогда не поднимется.

Для фронтенда no-op collector неактуален, но на бэкенде используется. Например, имея несколько серверов за балансировщиками, приложению отдается 32 Гб оперативки и потом оно убивается целиком. Так проще и производительность только повышается за счет простого перезапуска, когда памяти становится мало.  
  
В вебе так нельзя и приходится чистить.

### **Поиск и удаление мусора**

Начнем чистку с поиска мусора. Мы уже знаем, как это делать. Мусор — объекты С и F на прошлой схеме, потому что до них нельзя дойти по стрелочкам от корневого объекта.  
  
Мы берем этот мусор, скармливаем его любителю мусора и готово.  
  
  
  
После чистки проблема не решается, так как в памяти остаются дырки. Обратите внимание, что свободных квадратиков 7, но 5 из них мы все еще не можем выделить. Произошла фрагментация и на этом сборка закончилась. Такой алгоритм с дырками называется **Mark and Sweep**.

### **Mark and Sweep**

Плюсы:

* Очень простой алгоритм. Один из первых, про который вы узнаете, если начнете изучать Garbage collector.
* Работает пропорционально количеству мусора, но справляется только, когда мусора мало.
* Если у вас только живые объекты, то он не тратит время и просто ничего не делает.

Минусы:

* Требует сложной логики поиска свободного места, потому что когда в памяти много дырок, то в каждую приходится примерять объект, чтобы понять — подходит он или нет.
* Фрагментирует память. Может произойти ситуация, что при свободных 200 Мб память разбита на маленькие кусочки и, как в примере выше, нет цельного куска памяти под объект.

Ищем другие идеи. Если посмотреть на картинку и подумать, то первая мысль — сдвинуть все влево. Тогда справа останется один большой и свободный кусок, в который спокойно поместится наш объект.  
  
Такой алгоритм есть и он называется**Mark and Compact**.

### **Mark and Compact**

Плюсы:

* Дефрагментирует память.
* Работает пропорционально количеству живых объектов, а значит, можно использовать, когда мусора практически нет.

Минусы:

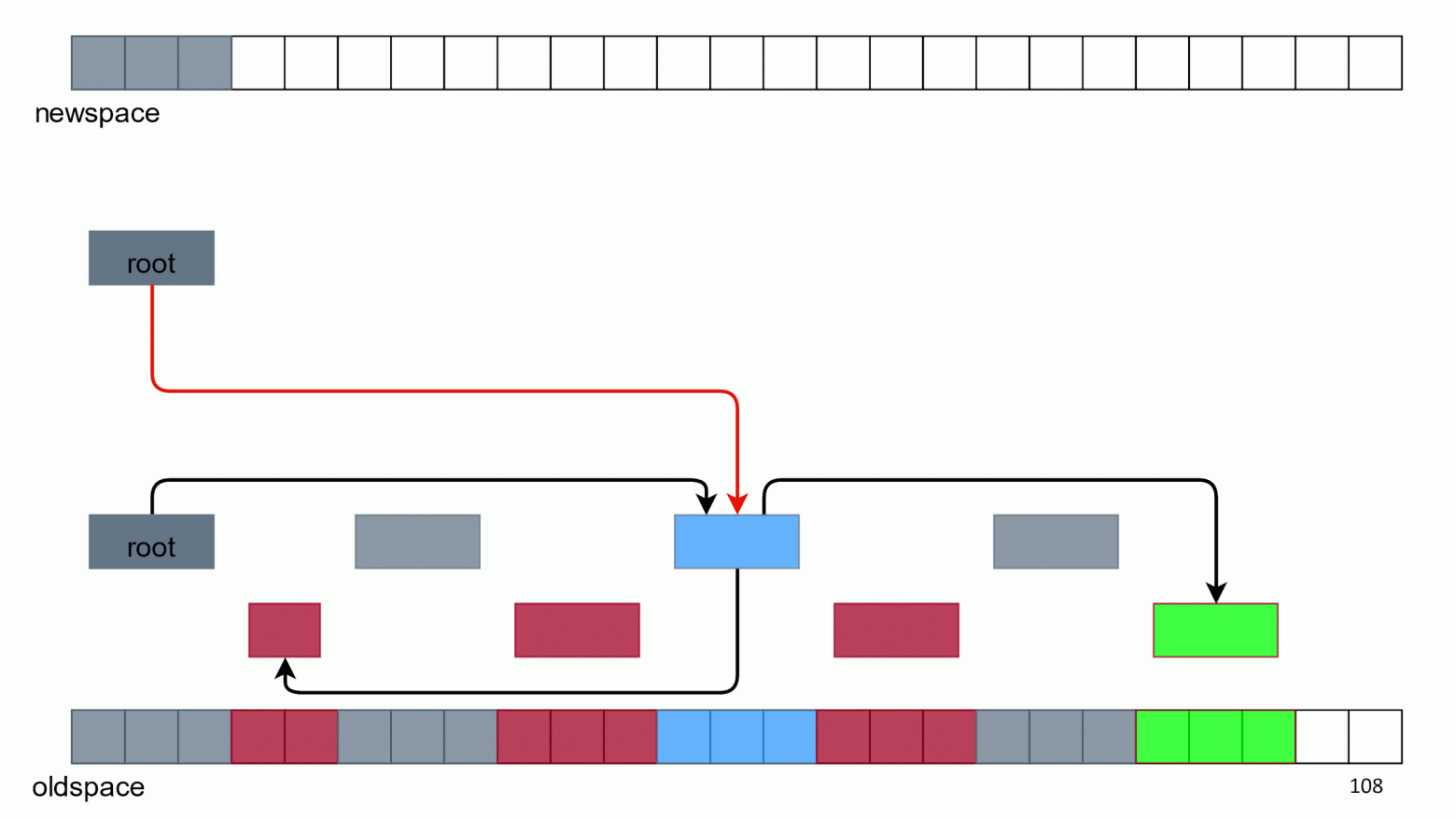
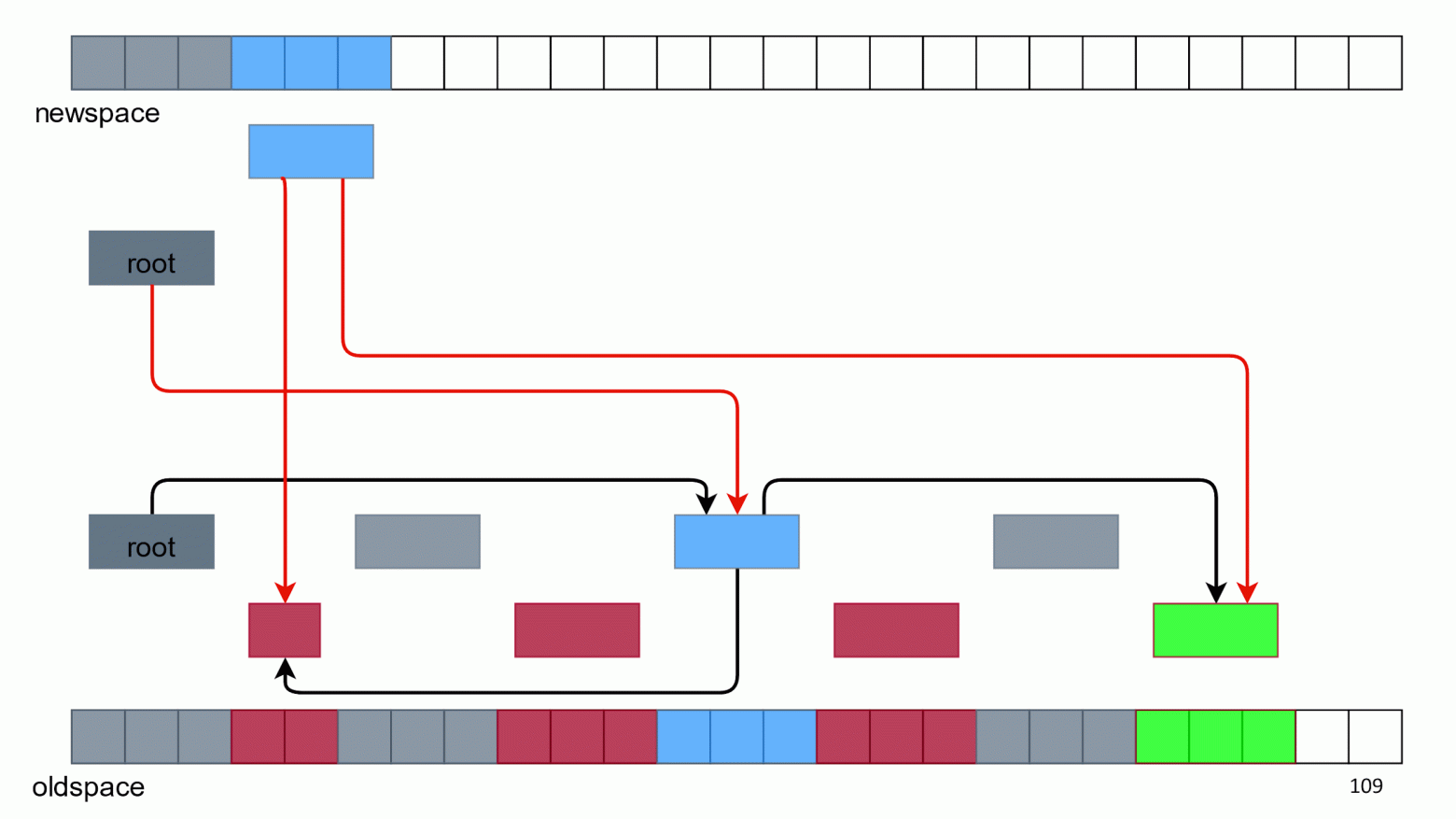
* Сложный в работе и реализации.
* Перемещает объекты. Мы подвинули объект, скопировали, теперь он находится в другом месте и вся эта операция довольно дорогая.
* Требует 2-3 прохода по всей памяти, в зависимости от реализации — алгоритм медленный.

Тут мы подходим еще к одной идее.

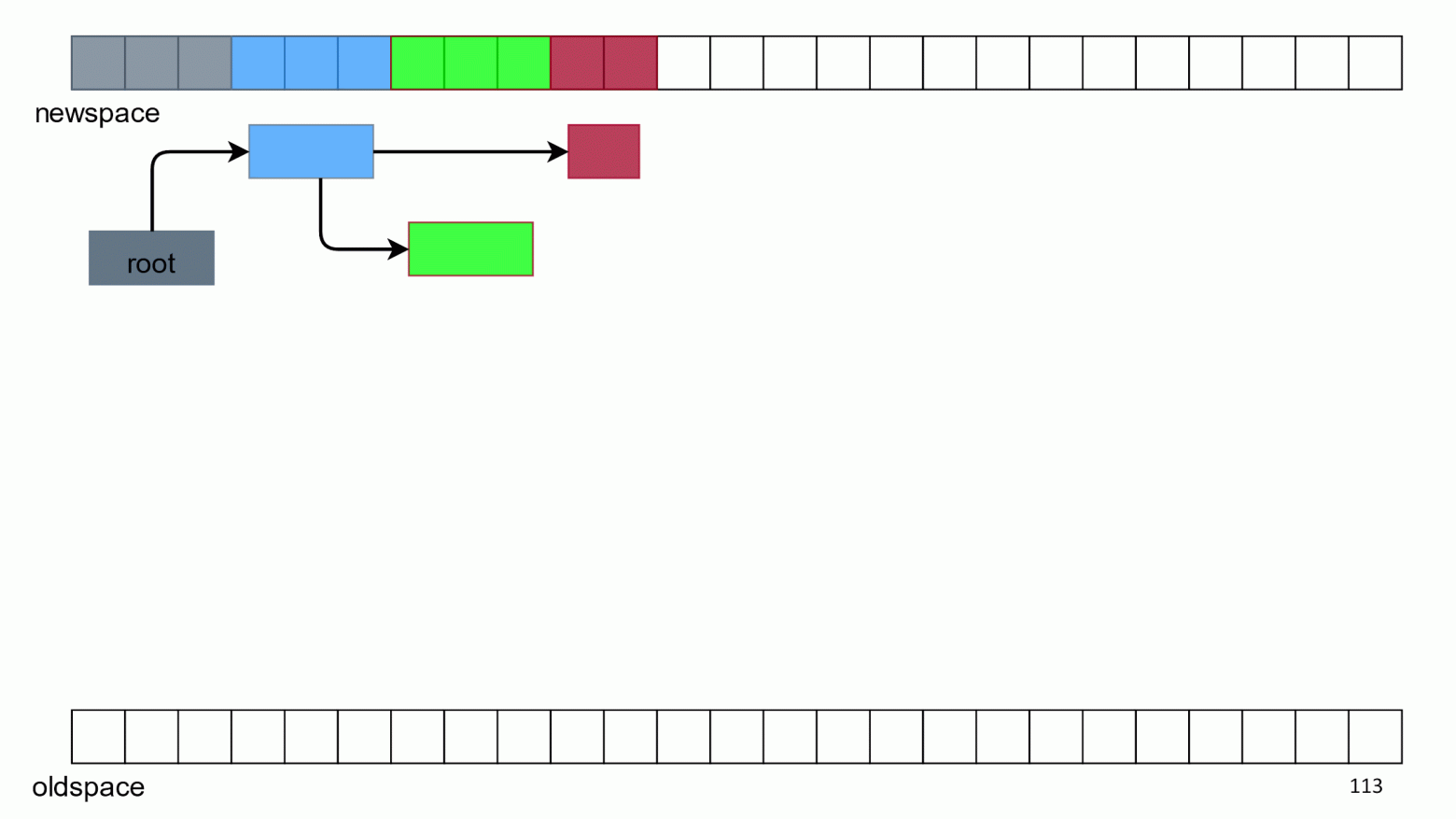
### **Сборка мусора не бесплатна**

В High performance API типа WebGL, WebAudio и WebGPU, который пока еще в разработке, объекты создаются и удаляются в отдельных фазах. Эти спецификации написаны так, чтобы сборки мусора не было в процессе. Более того, там нет даже Promise, а есть pull() — вы просто опрашиваете каждый кадр: «Произошло что-нибудь или нет?».

### **Semispace aka Lisp 2**

Есть еще один сборщик, про который хочется поговорить. Что, если не освобождать память, а копировать все живые объекты куда-нибудь в другое место.  
  
Давайте попробуем скопировать корневой объект «как есть», который куда-то ссылается.  
  
  
  
А потом все остальные.  
  
  
  
Сверху нет мусора и дырок в памяти. Кажется, что все хорошо, но возникают две проблемы:

* Дублирование объектов — у нас есть два зеленых объекта и два синих. Какой из них использовать?
* Ссылки у новых объектов ведут на старые объекты, а не друг на друга.

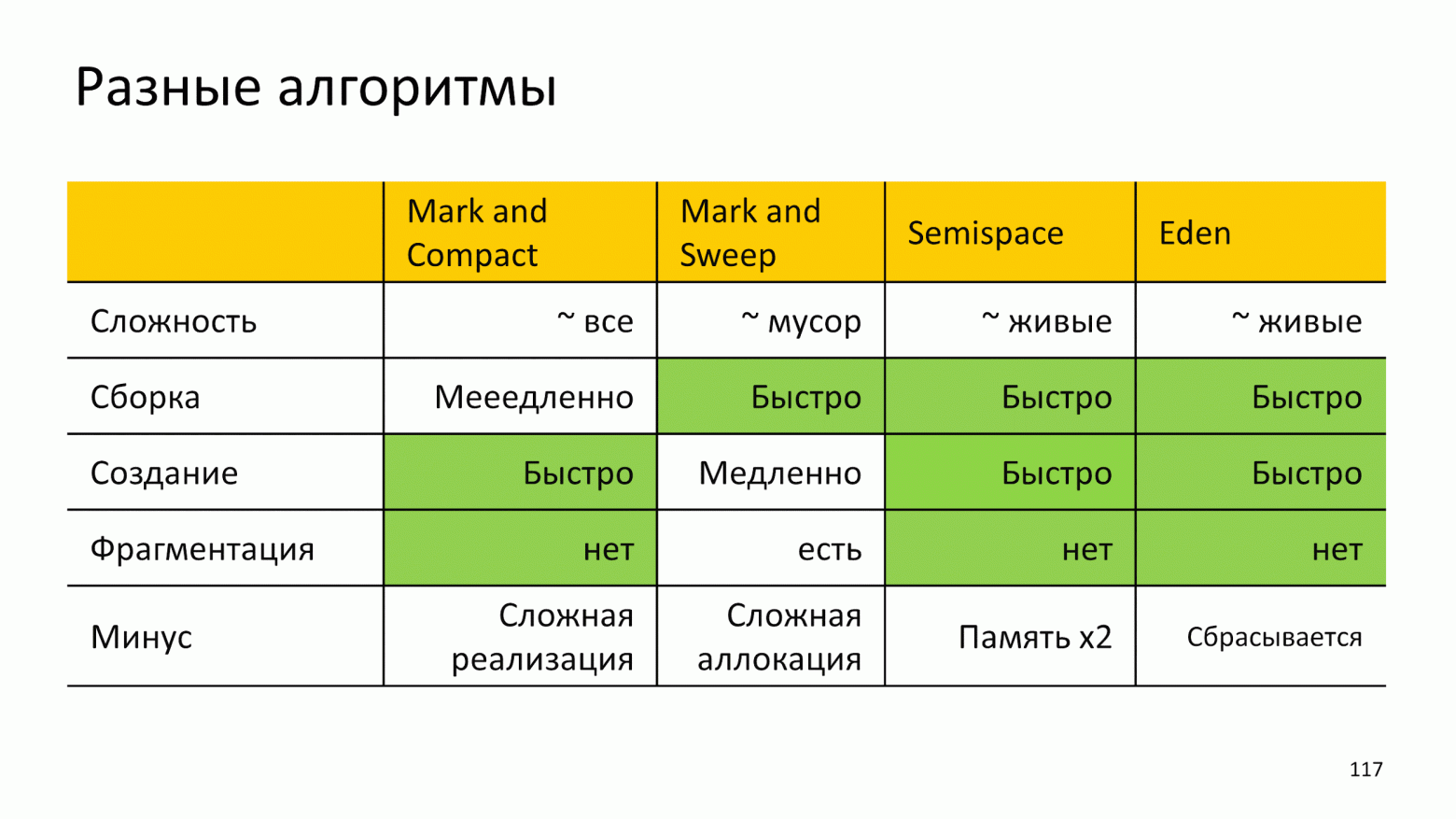
Со ссылками все решается с помощью особой алгоритмической «магии», а с дублированием объектов справляемся удалением всего, что снизу.  
  
  
В итоге у нас остается свободное место, а сверху только живые объекты в нормальном порядке. Этот алгоритм называется **Semispace**, **Lisp 2** или просто «копирующим сборщиком».  
  
Плюсы:

* Дефрагментирует память.
* Простой.
* Можно совместить с фазой обхода.
* Работает пропорционально количеству живых объектов по времени.
* Хорошо работает, когда много мусора. Если у вас есть 2 Гб памяти и в ней 3 объекта, то вы обойдете только 3 объекта, а остальных 2 Гб будто и не было.

Минусы:

* Двойной расход памяти. Вы используете памяти в 2 раза больше, чем надо.
* Перемещает объекты — это тоже не очень дешевая операция.

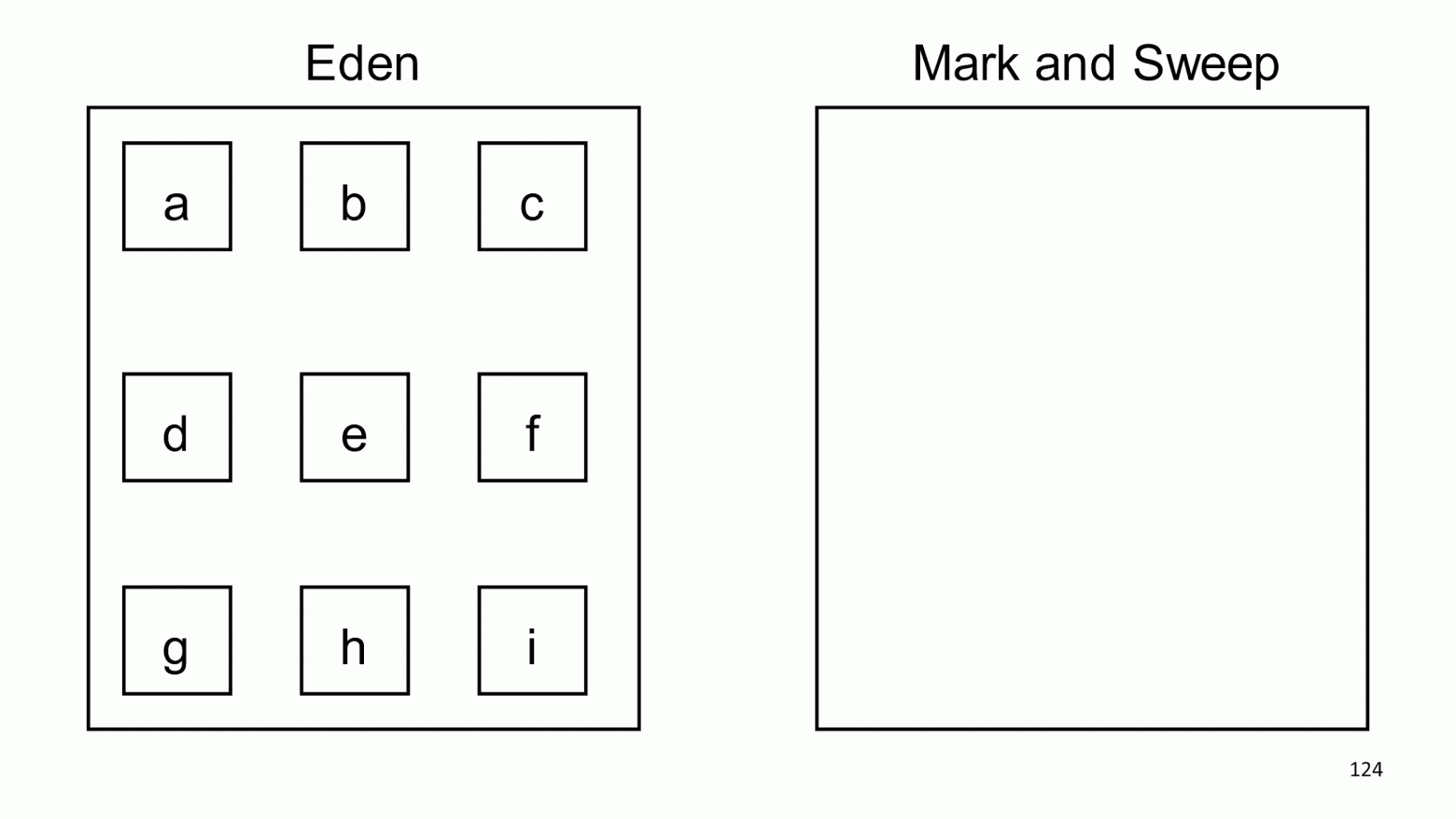
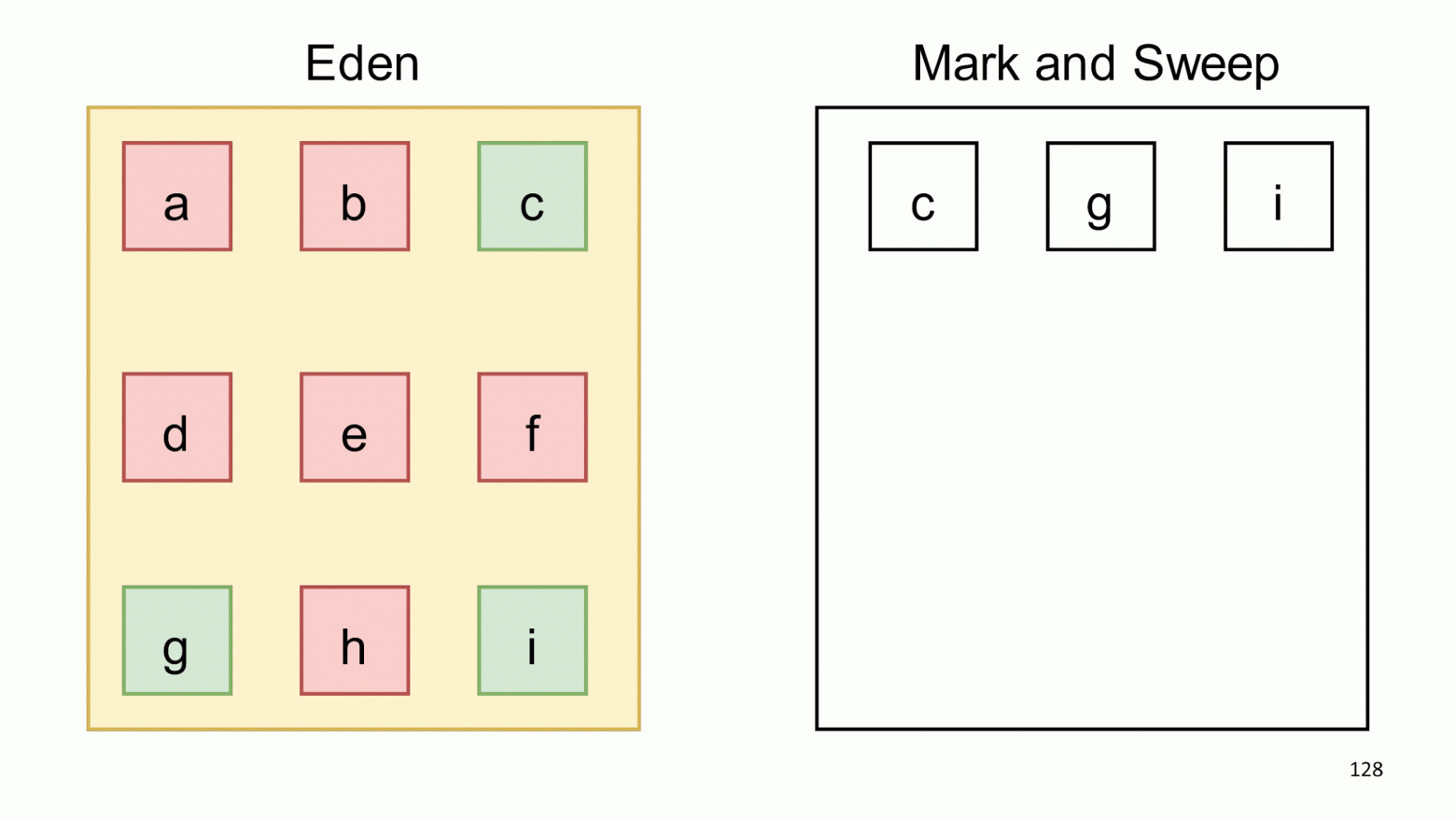
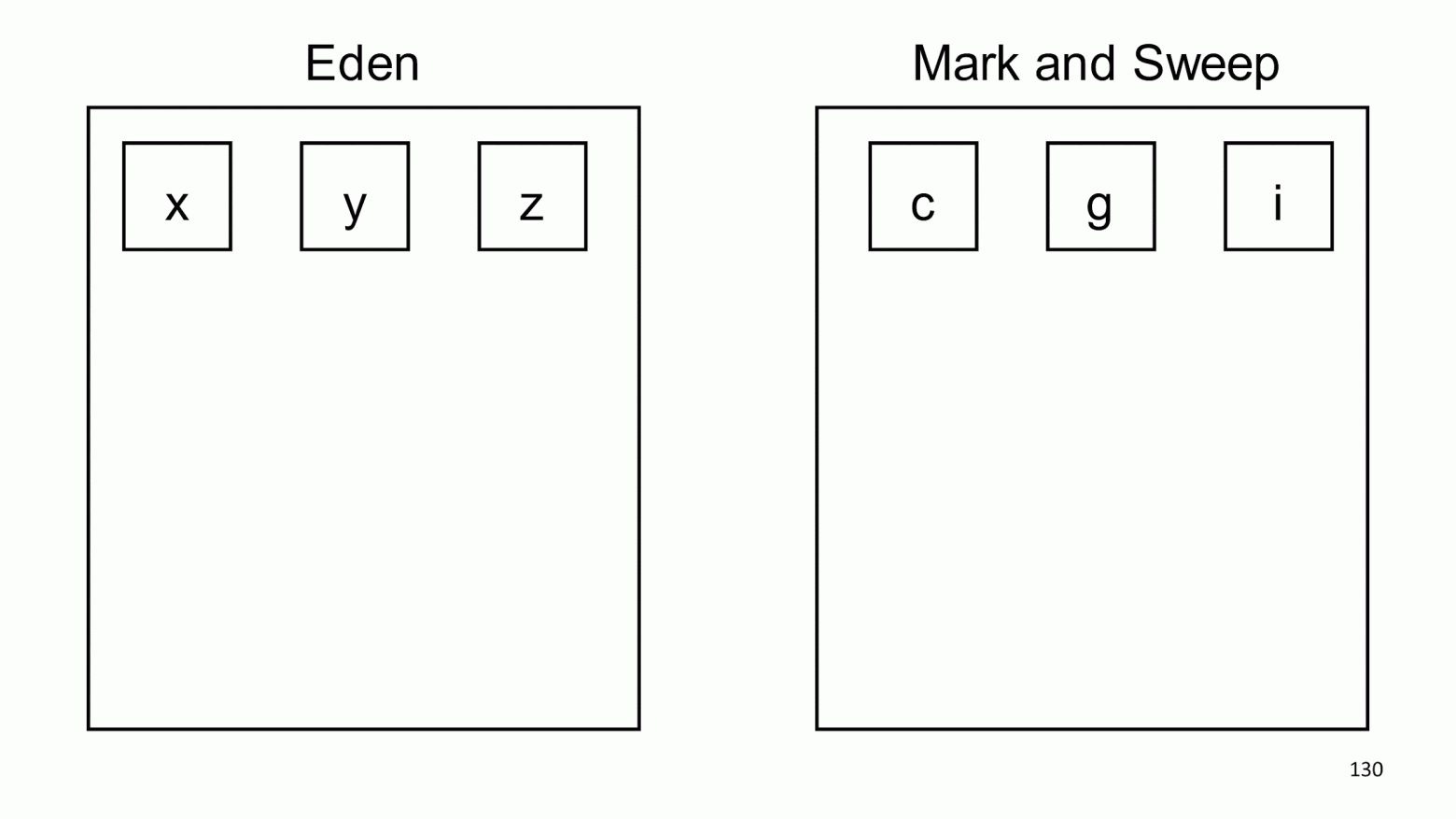
На заметку: сборщики мусора могут перемещать объекты.

В вебе это неактуально, а в Node.js даже очень. Если вы пишете расширение на C++, то язык про все это не знает, поэтому там есть двойные ссылки, которые называются handle и выглядят примерно так: v8::Local<v8::String>.  
  
Поэтому, если вы собираетесь писать плагины под Node.js то информация вам пригодится.  
  
Обобщим разные алгоритмы с их плюсами и минусами в таблицу. В ней есть еще алгоритм Eden, но про него позже.  
  
  
  
Очень хочется алгоритм без минусов, но такого нет. Поэтому берем лучшее из всех миров: используем несколько алгоритмов одновременно. В одном куске памяти собираем мусор одним алгоритмом, а в другом — другим алгоритмом.  
  
Как понять эффективность алгоритма в такой ситуации?  
  
Можем воспользоваться знаниями умных мужей из 60-х, которые посмотрели на все программы и поняли:

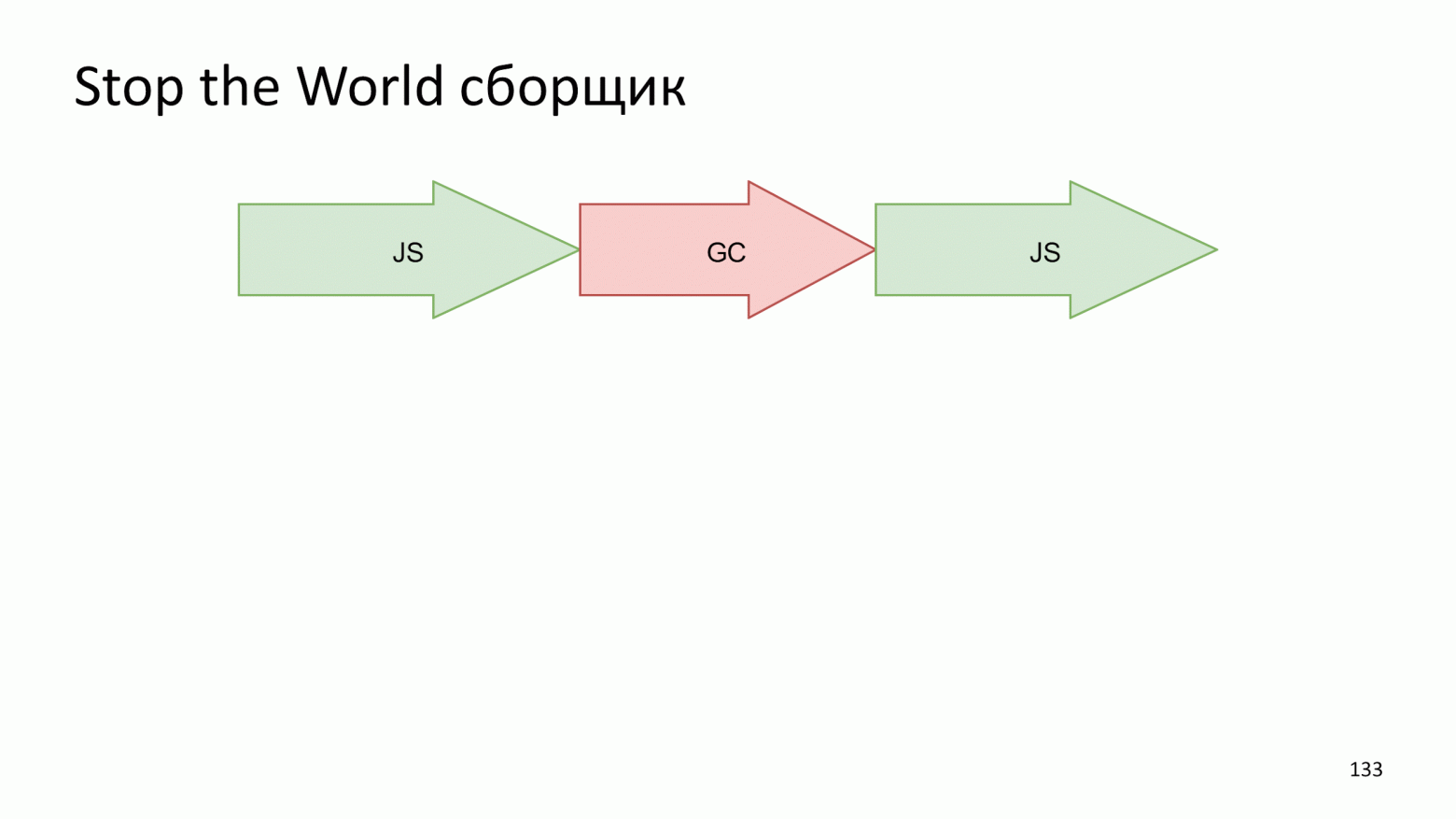
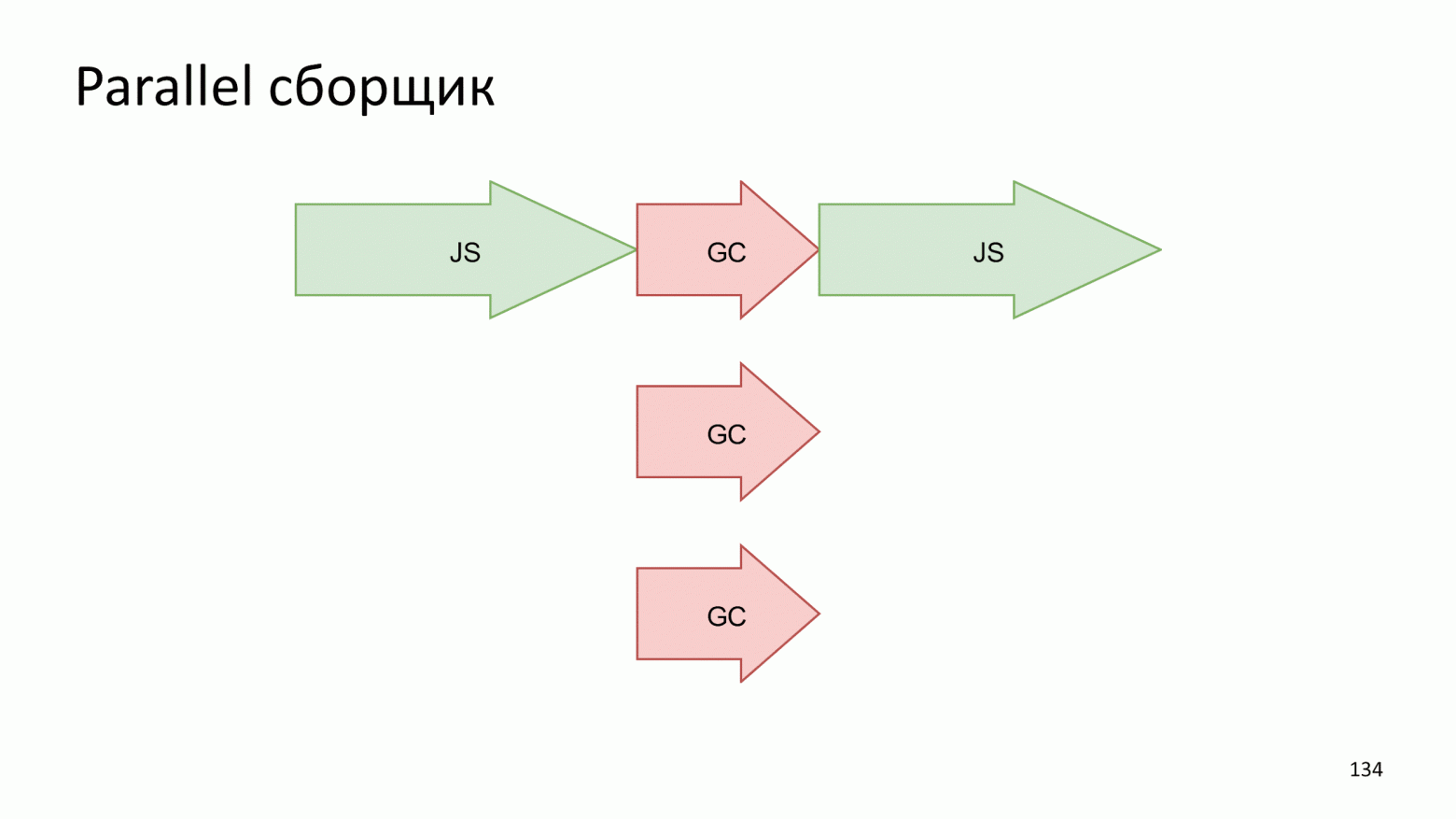
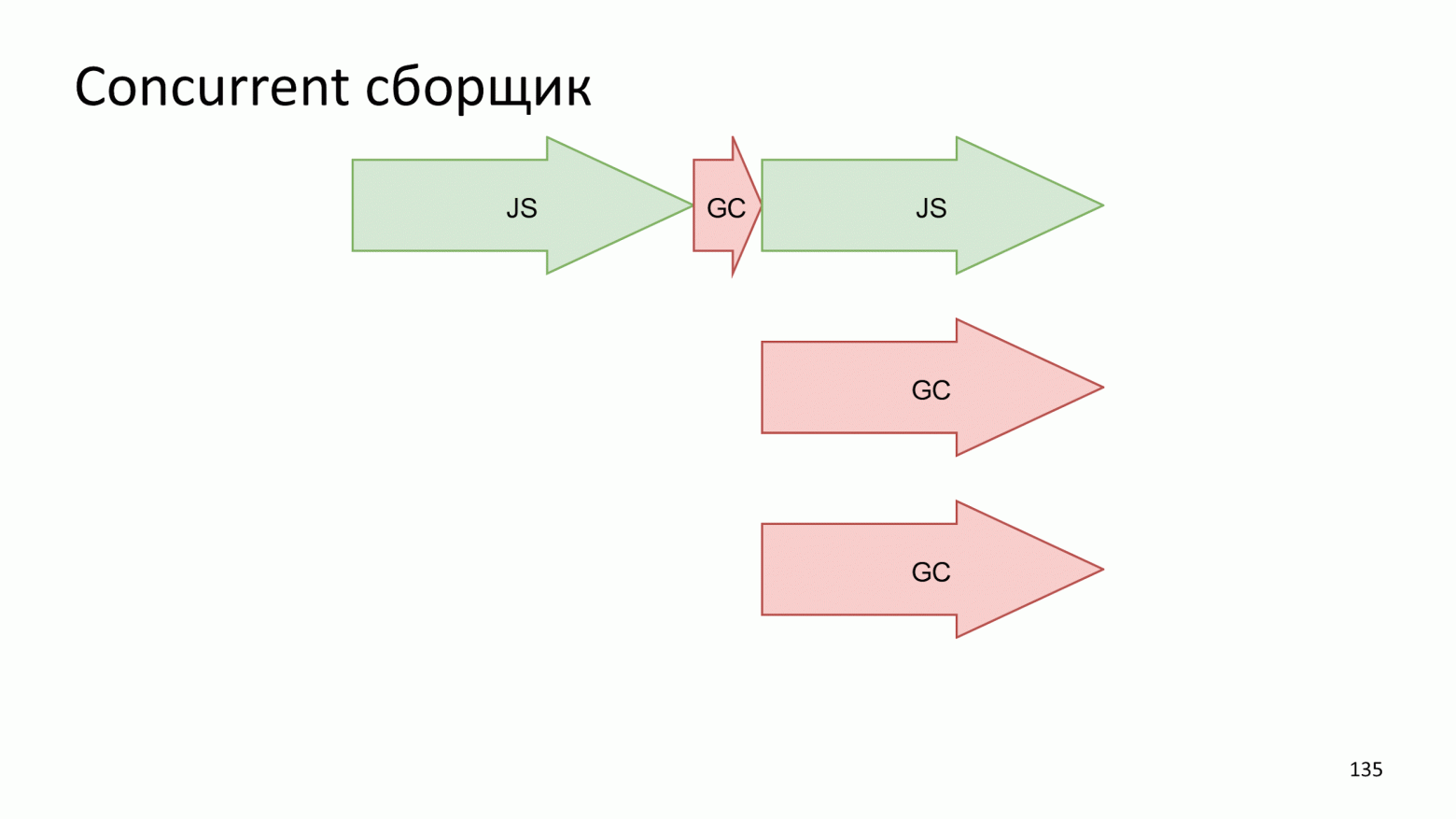
Слабая гипотеза о поколениях: большинство объектов умирают молодыми.

Эти они хотели сказать, что все программы только и делают, что плодят мусор. В попытке использовать знания, мы придем к тому, что называется «сборкой поколениями».

### **Сборка поколениями**

Создаем два куска памяти, которые никак не связаны: слева Eden, а справа медленный Mark and Sweep. В Eden создаем объекты. Много объектов.  
  
  
  
Когда Eden говорит, что наполнен, мы запускаем в нем сборку мусора. Находим живые объекты и копируем их в другой сборщик.  
  
  
  
Сам Eden полностью очищаем, и можем дальше добавлять в него объекты.  
  
  
  
Полагаясь на гипотезу поколений, мы решили, что объекты с, g, i скорее всего проживут еще долго, и можно проверять их на мусор реже. Зная эту гипотезу, можно писать программы, которые обманывают сборщик. Так можно делать, но я вам не советую, потому что это приведет почти всегда к нежелательным эффектам. Если создавать долгоживущий мусор, сборщик начнет считать, что его не требуется собирать.  
  
Классический пример обмана — LRU-cache. В кэше долго лежит объект, сборщик смотрит на него и считает, что пока собирать не будет, потому что объект проживет еще очень долго. Потом в кэш попадает новый объект, а большой старый из него выталкивается и собрать этот большой объект сразу уже нельзя.  
  
Как собирать теперь мы знаем. Поговорим о том, когда собирать.

### **Когда собирать?**

Самый простой вариант, когда мы **просто все останавливаем**, запускаем сборку, а потом снова запускаем работу JS.  
  
  
  
В современных компьютерах не один поток выполнения. В вебе это знакомо по Web Workers. Почему-бы не взять и не **распараллелить процесс сборки**. Произвести несколько маленьких операций одновременно будет быстрее, чем одну большую.  
  
  
  
Еще одна идея — аккуратно сделать слепок текущего состояния, а **сборку вести параллельно выполнению JS**.  
  


Если это вас заинтересовало, то советую почитать:

* Единственную и главную книгу по сборке «Garbage Collection Handbook».
* [Википедию](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)) как универсальный ресурс.
* Сайт[memorymanagement.org.](http://www.memorymanagement.org/)
* Доклады и статьи [Александра Шепелева](https://shipilev.net/). Он рассказывает про Java, но с точки зрения мусора Java и V8 работают примерно одинаково.

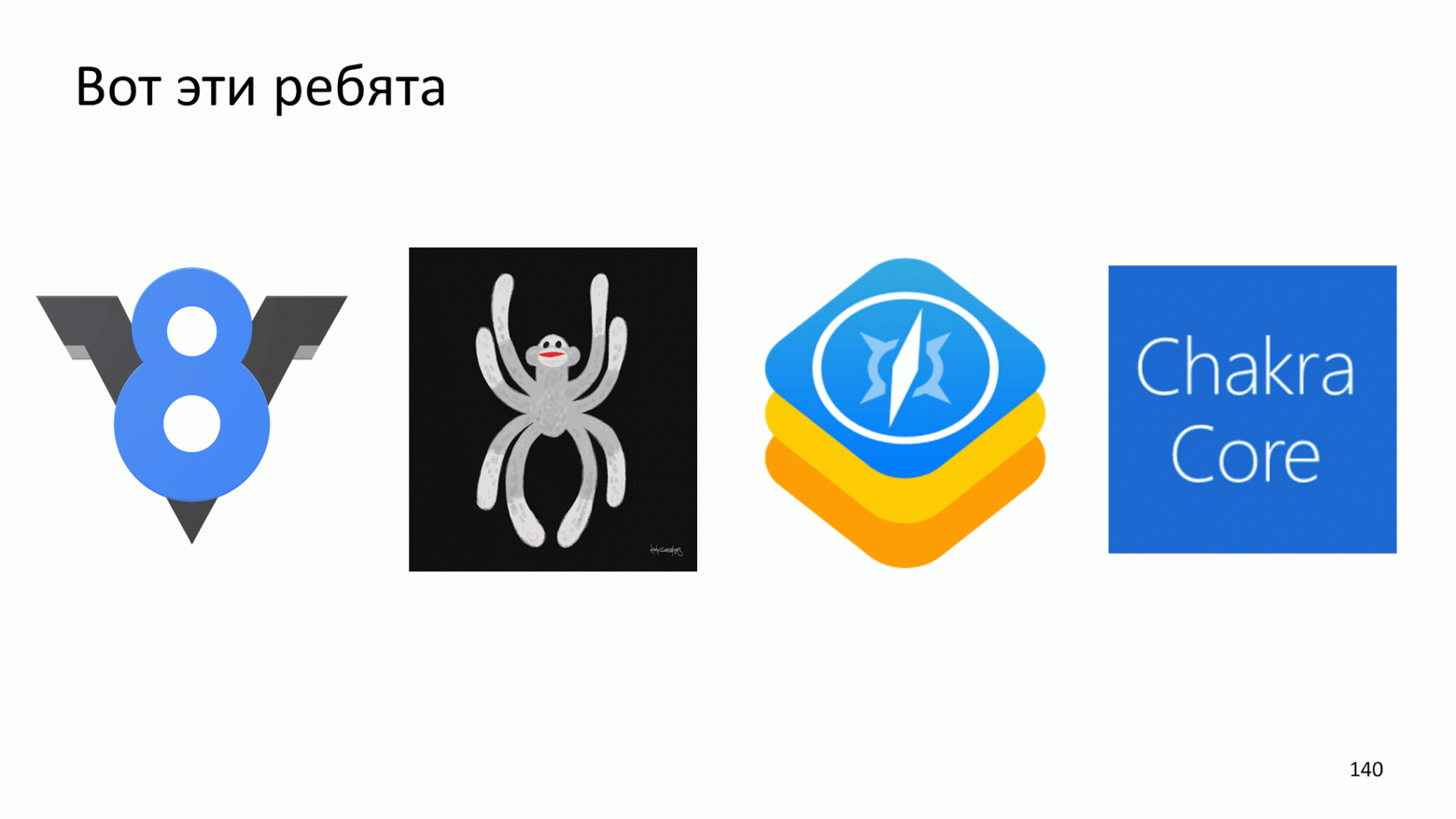
## **Браузерная реальность**

Давайте перейдем к тому, как в браузерах используется все, о чем мы уже поговорили.

### **IoT-движки**

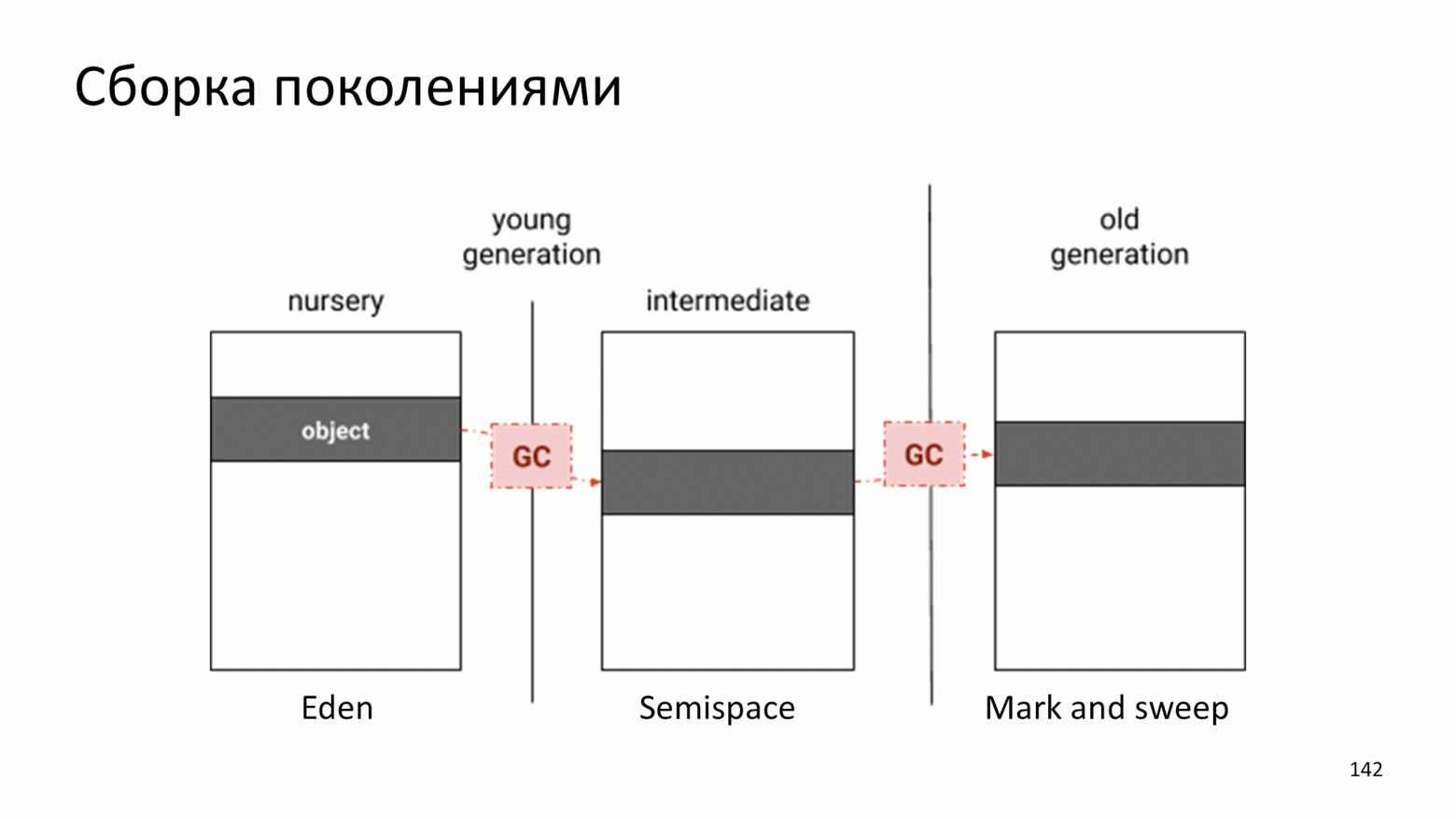
Начнем не совсем с браузеров, а с Internet of Things движков: JerryScript и Duktape. В них используются алгоритмы Mark’n’sweep и Stop the world.  
  
IoT-движки работают на микроконтроллерах, а значит: язык медленный; секундные зависания; фрагментация; и все это для чайника с подсветкой:)  
  
Если вы пишите Internet of Things на JavaScript, то расскажите в комментариях? есть ли смысл?  
  
IoT-движки оставим в покое, нас интересуют:

* V8.
* SpiderMonkey. Логотипа у него, на самом деле, нет. Логотип самодельный:)
* JavaScriptCore, который используется в WebKit.
* ChakraCore, что используется в Edge.

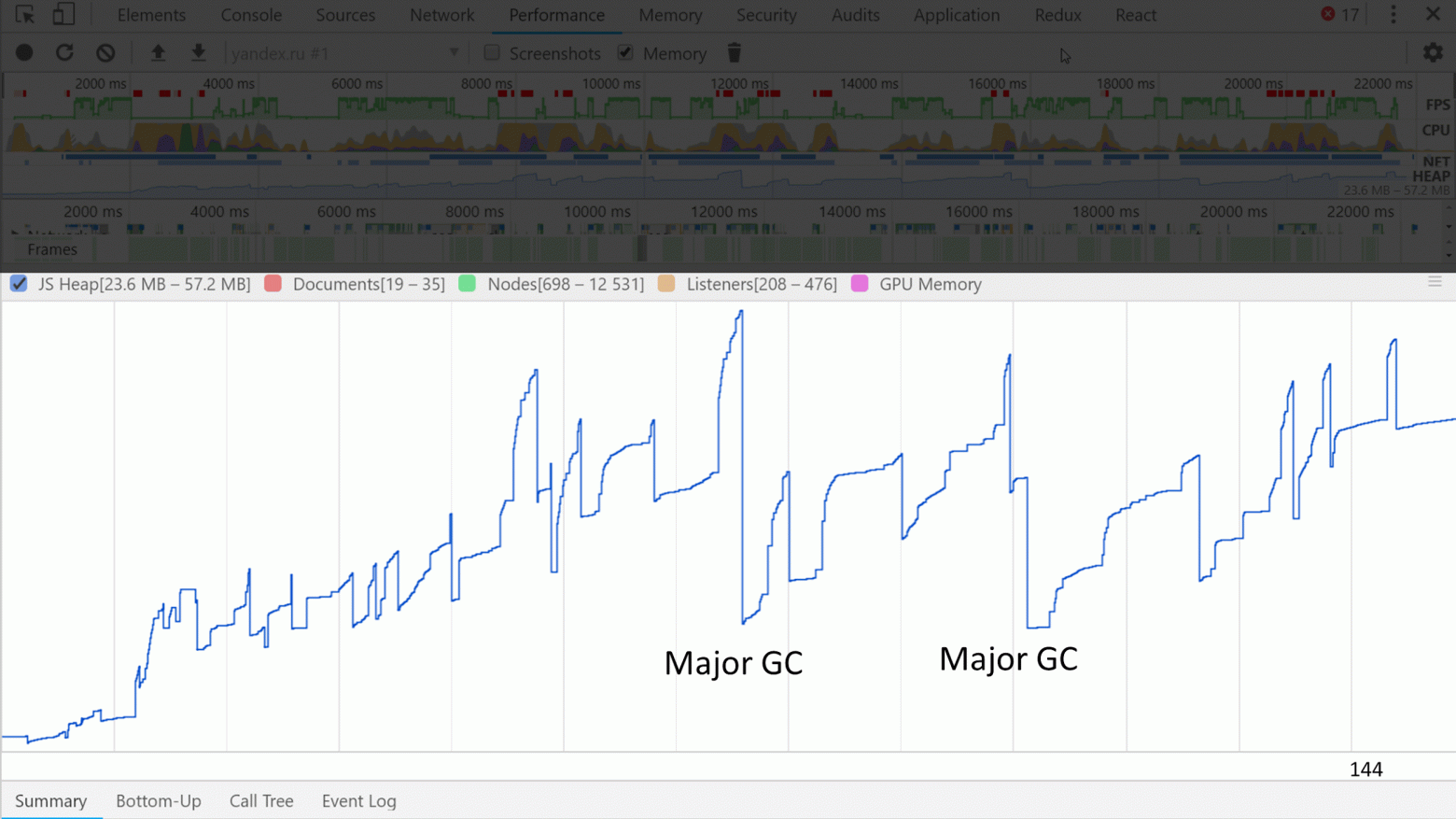
  
  
Все движки примерно одинаковы, поэтому говорить будем про V8, как самый известный.

### **V8**

* Почти весь серверный JavaScript, потому что это Node.js.
* Почти 80% клиентского JavaScript.
* Самые общительные разработчики, информации много и хорошие исходники, которые проще всего читать.

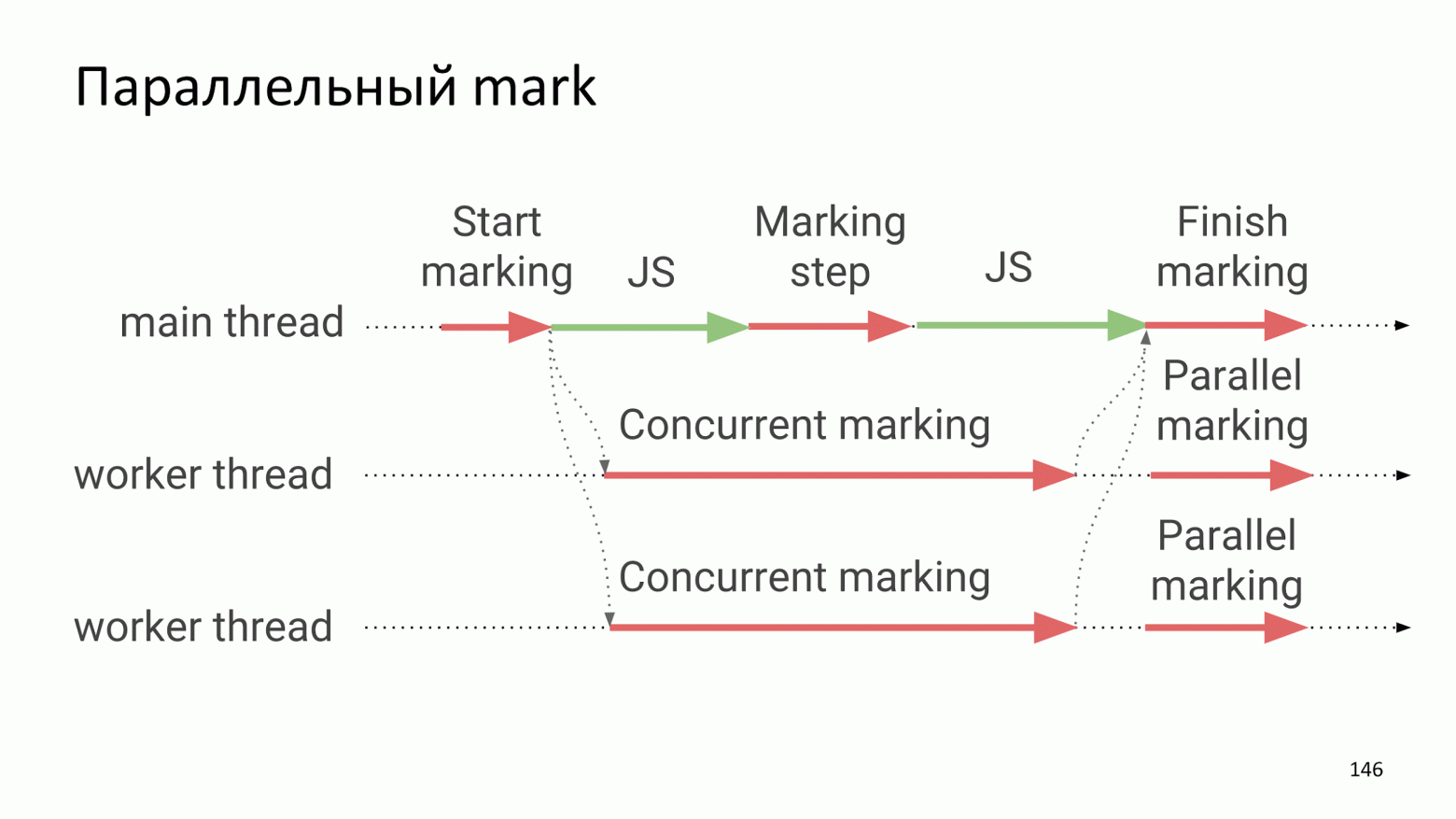
В V8 используется сборка поколениями.  
  
  
Единственное отличие — раньше у нас было два сборщика, а теперь три:

* Создается объект в Eden.
* В какой-то момент в Eden становится слишком много мусора и объект перекладывается в Semispace.
* Объект подростает и когда сборщик понимает, что он слишком старый и скучный, то перекидывает в Mark and Sweep, в котором сборка мусора производится крайне редко.

Наглядно посмотреть, как это выглядит можно на **memory trace**.  
  
  
  
Заметны несколько больших волн, на которых есть маленькие волны. Маленькие — это минорные сборки, а большие — мажорные.  
  
Смысл нашего существования, согласно гипотезе о поколениях — генерировать мусор, поэтому следующая ошибка — это боязнь создания мусора.

Мусор можно создавать, когда это действительно мусор. Если переиспользовать объект, то он будет жить гораздо дольше и сборщик решит, что его не надо собирать, поэтому не делайте так.

#### **Параллельный mark**

Относительно недавно разработчики V8 распараллелили фазу поиска живых объектов.  
  
  
Фаза сборки все еще Stop the world, но большую часть сборки составляет поиск живых объектов, что происходит абсолютно независимо от выполнения JS, то есть добавляет очень и очень маленькие зависания.

#### **Сколько стоит все это удовольствие?**

От 1 до 3%, что не очень много.  
  
Но 3% = 1/33 и это много для GameDev. В GameDev 3% это 1 кадр, что уже заметный лаг. Поэтому в GameDev стараются не использовать сборщик мусора.

**const** pool = [**new** Bullet(), **new** Bullet(), /\* ... \*/];

**function** **getFromPool**() {

**const** bullet = pool.find(x => !x.inUse);

bullet.isUse = true;

**return** bullet;

}

**function** **returnToPool**(bullet) { bullet.inUse = false; }

// Frame

**const** bullet = getFromPool();

// ...

returnToPool(bullet);

Вместо использования сборщика заранее создается, например, 10 000 пуль и эти пули выделяются и освобождаются руками.  
  
Еще один пример — система частиц в играх. Если у вас есть огонь или звездочки из волшебной палочки, то вы можете создавать тысячи и десятки тысяч объектов. Это очень большая нагрузка на сборщика, что неизбежно приведет к зависанию.

### **Статистика сборщика мусора: Chromium**

Из сборщика можно вытащить немного статистики, но, к сожалению, только в Chromium.

> performance.memory

MemoryInfo {

totalJSHeapSize: 10000000,

usedJSHeapSize: 10000000,

jsHeapSizeLimit: 2330000000

}

В Chromium есть **performance.memory** и можно узнать, сколько в данную минуту зарезервировано под страницу, сколько памяти используется и сколько Chromium готов выделить.  
  
**Спойлер:** Chromium готов выделять 2 Гб памяти для JavaScript.  
  
К сожалению, документации все еще нет и почитать можно только исходники и блогпосты.

### **Статистика сборщика мусора: Node**

У Node.js все стандартизовано в **process.memoryUsage**, который возвращает примерно то же самое.

> process.memoryUsage()

{ rss: 22839296,

heapTotal: 10207232,

heapUsed: 5967968,

external: 12829 }

Надеюсь, что когда-нибудь эту информацию стандартизуют и она попадет в другие браузеры, но пока надежды нет. Зато есть мысли дать разработчикам больший контроль над сборкой мусора. Имеются в виду слабые ссылки.

### **Будущее**

**Слабые ссылки** — это почти как обычные, только они могут быть собраны в случае нехватки памяти. На это дело есть [proposal](https://github.com/tc39/proposal-weakrefs), но пока он во второй стадии.  
  
Если у вас Node.js, то вы можете иcпользовать [node-weak](https://www.npmjs.com/package/weak) и слабые ссылки, например, для кэширования.

**let** cached = **new** WeakRef(myJson);

// 2 часа спустя

**let** json = cached.deref();

**if** (!json) {

json = **await** fetchAgain();

}

Вы можете сохранить большой объект, например, если у вас видео-процессинг на JS. Можно хранить кэши в слабых ссылках и сообщать сборщику, что если памяти будет не хватать, то эти объекты он может удалить, а после проверить удаление.  
  
Еще в будущем нас ждет [сборщик в WebAssembly](https://github.com/WebAssembly/gc), но пока все туманно. На мой взгляд, в текущем решении не решено большое количество проблем и даже когда разработчики его допишут, туман неопределенности будет еще плотнее, ведь никто не умеет этим пользоваться.

Почитать про браузеры особо нечего: есть [v8.dev](https://v8.dev/) и исходники движков JS.

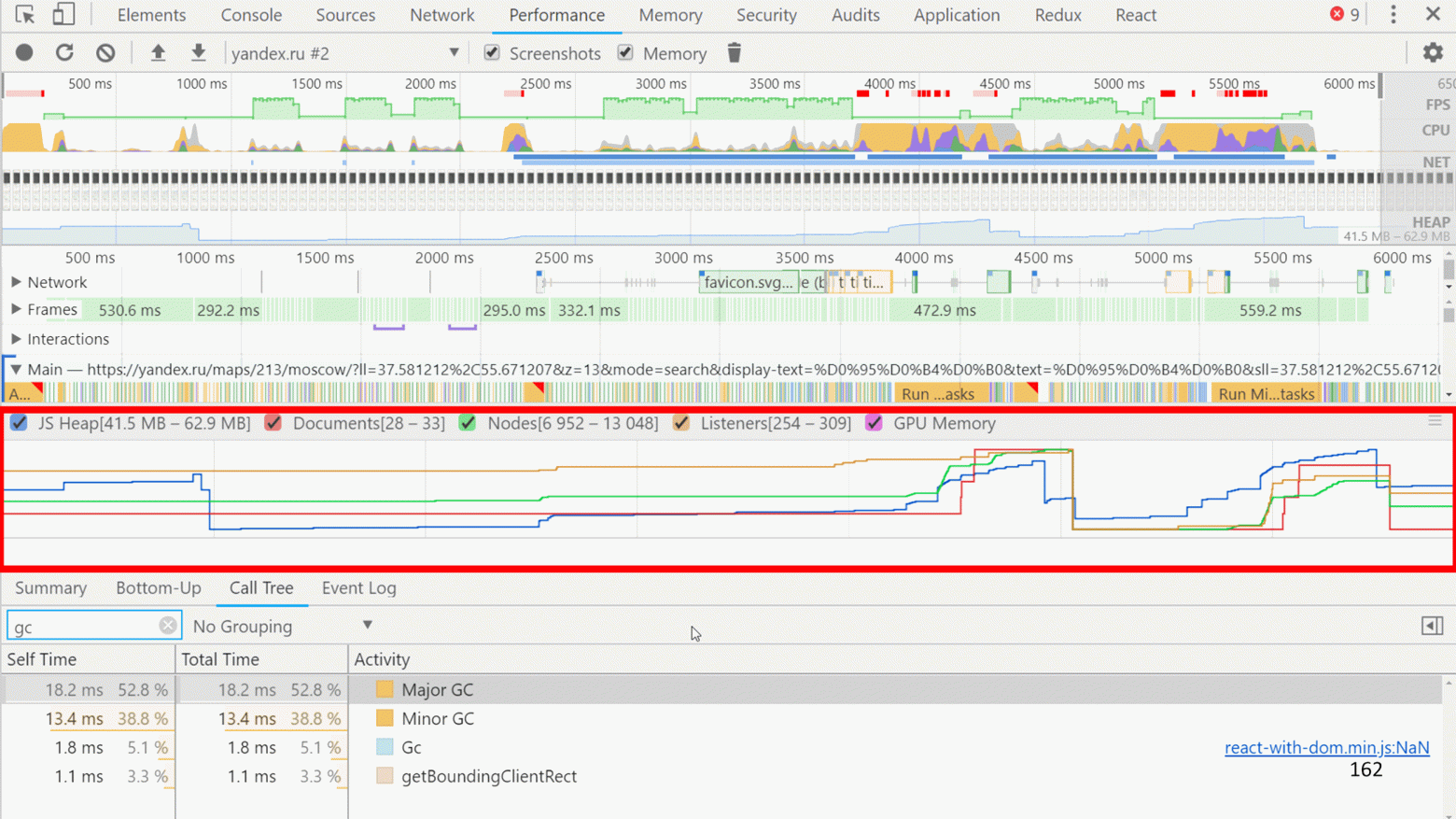
* [github.com/v8/v8/tree/7.0.237/src/heap](https://github.com/v8/v8/tree/7.0.237/src/heap)
* [github.com/servo/mozjs/blob/master/mozjs/js/src/gc/](https://github.com/servo/mozjs/blob/master/mozjs/js/src/gc/)
* [github.com/WebKit/webkit/.../JavaScriptCore/heap/MarkedSpace.cpp](https://github.com/WebKit/webkit/blob/master/Source/JavaScriptCore/heap/MarkedSpace.cpp)
* [github.com/Microsoft/ChakraCore/.../HeapAllocator.cpp](https://github.com/Microsoft/ChakraCore/blob/master/lib/Common/Memory/HeapAllocator.cpp)
* [github.com/svaarala/duktape/.../duk\_heap\_markandsweep.c](https://github.com/svaarala/duktape/blob/master/src-input/duk_heap_markandsweep.c)
* [github.com/jerryscript-project/jerryscript/.../ecma-gc.c](https://github.com/jerryscript-project/jerryscript/blob/v1.0/jerry-core/ecma/base/ecma-gc.c)

Как это все можно применять в повседневной жизни?

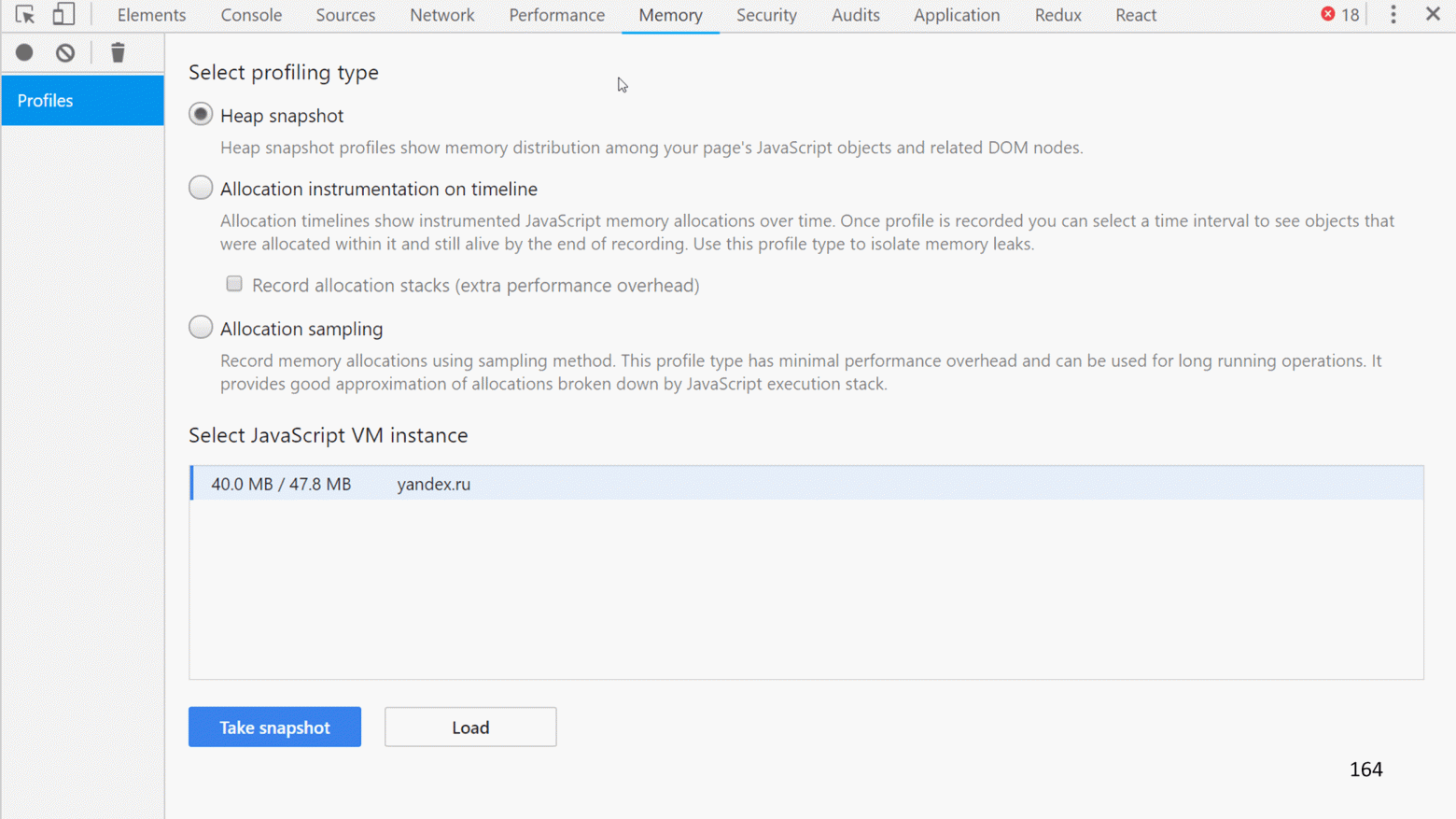
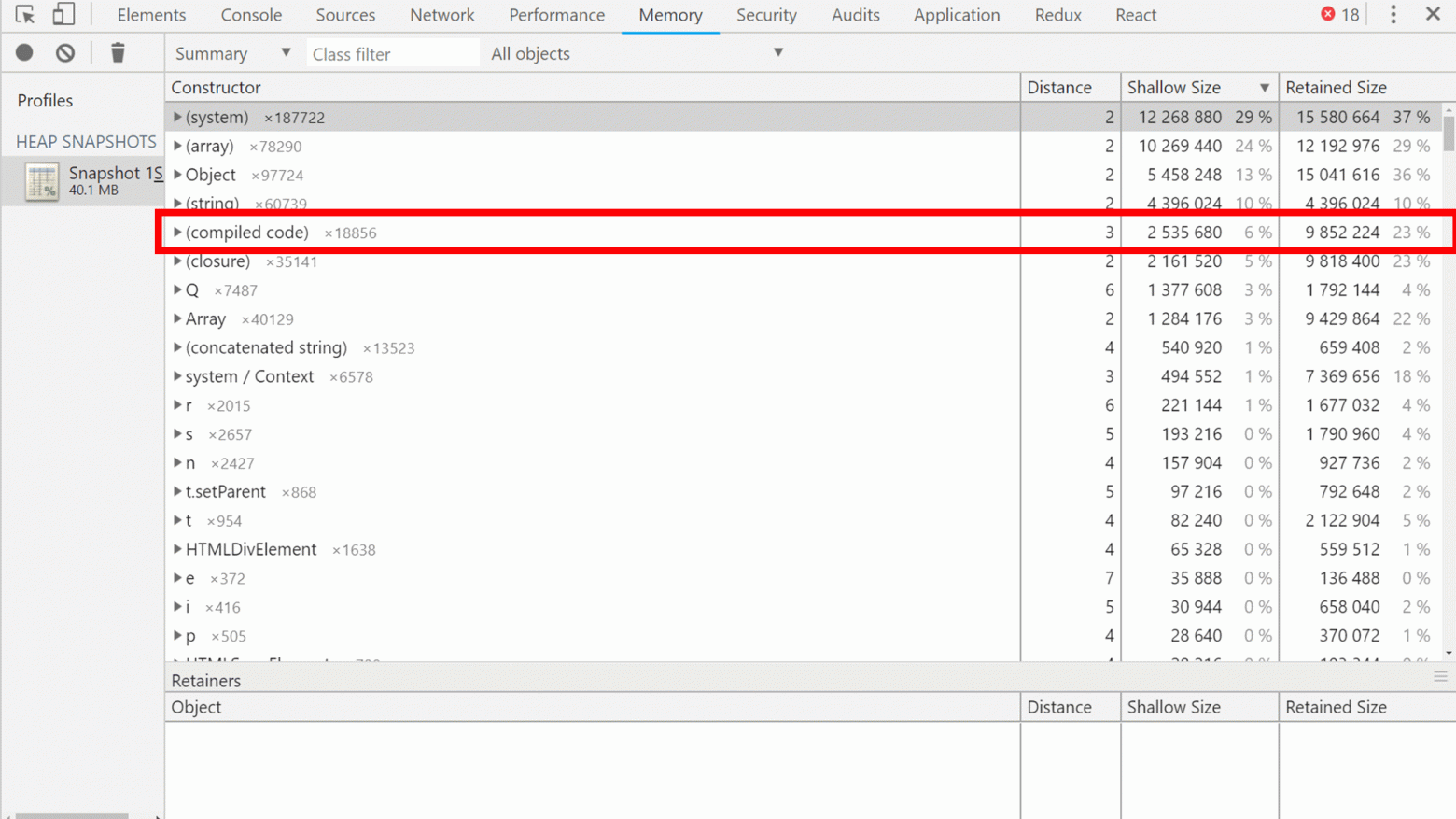
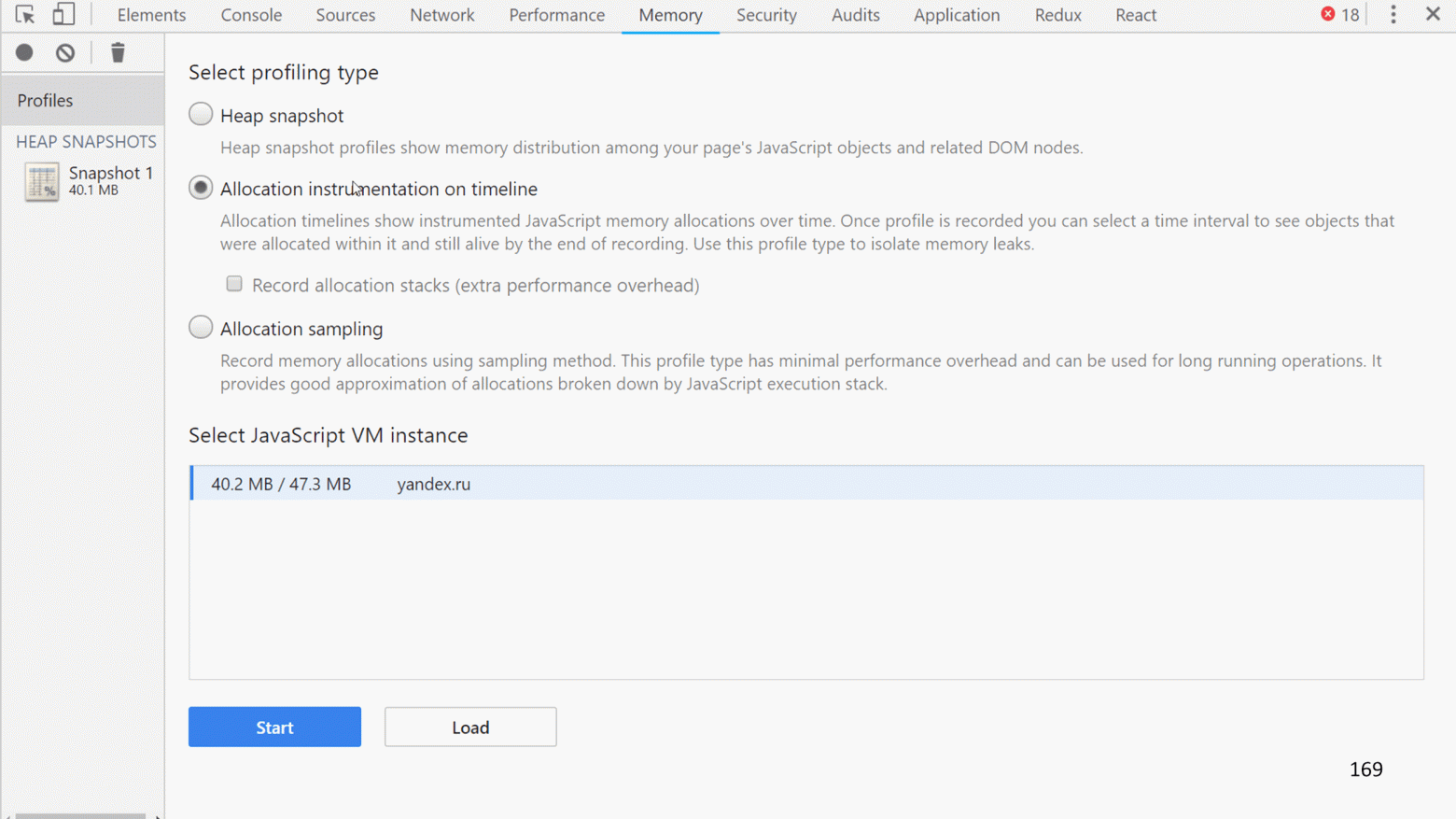
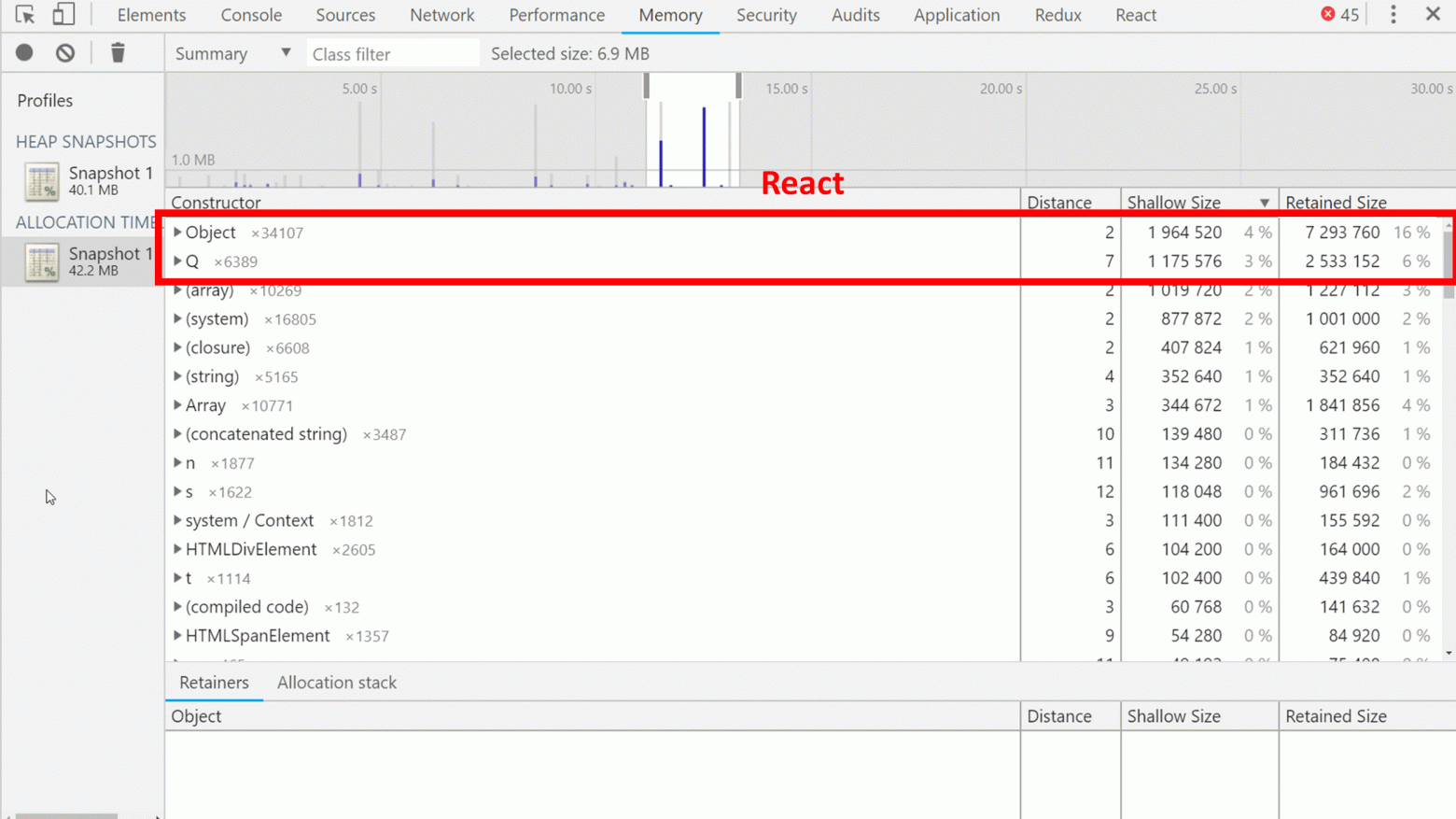
## **Повседневность**

В повседневности есть DevTools и две замечательные вкладки: **Performance** и **Memory**. Вкладки рассмотрим на примере Chromium, потому что все им пользуются, а для Firefox и Safari все аналогично.

### **Вкладка Performance**

Если снять Trace, нажав галочку «Memory» прямо под вкладкой Performance, вместе со слепком JS запишется красивый график потребления памяти.  
  
  
  
А еще вместе с JS записываются все внутренние события в V8 и вся информация о том, сколько работал сборщик. На слайде на примере Яндекс.Карт можно увидеть, что GC в сумме работал 30 мс из 1200 мс JS, то есть 1/40.

### **Вкладка Memory**

На вкладке можно снять слепок всей памяти со всеми объектами.  
  
  
  
Выглядит это примерно так.  
  
  
  
Объекты можно сортировать по размерам, по размерам объектов плюс сколько они за собой тянут других объектов. Из интересного можно увидеть, например, скомпилированный код, который V8 собирает в машинный, чтобы выполняться очень и очень быстро. Он тоже живет в сборщике, и сборщик мусора его собирает.  
  
Еще, например, объект Q (двумя строчками ниже compiled code) — это React в минифицированном коде Карт. Занимает много места, но что поделать?  
  
Если вы хотите видеть, сколько места занимают ваши объекты, то давайте им имена для удобного поиска в слепке или при минификации сохраняйте, во что они минифицировались.  
  
Еще можно снять слепок аллокации, то есть создания объектов.  
  
  
  
Это примерно то же самое, что и слепок текущего состояния, только их много, они создаются постоянно и у вас есть график процесса. На графике видно, что есть пики — около 4 Мб создается за один тик. Можно посмотреть, что там.  
  
  
  
Разные утилитарные объекты и снова React, потому что в этот момент карта что-то перерисовывала: пришел ответ от сервера и обновился почти весь интерфейс. Соответственно, создавалась куча JSX.  
  
Если Performance и Memory вдруг не хватило, то можно использовать:

* В Chromium: about:tracing.
* В Firefox: about:memory и about:performance, но их сложно читать.
* Флаги для Node — trace-gc, —expose-gc, require(’trace\_events’). Через trace\_events можно программно собирать статистику.

## **Итоги**

* Сборщик мусора умный, его делали знающие разработчики, чтобы он собирал мусор, который вы производите.
* Никто не мешает вам обмануть сборщик или усложнить ему работу и прострелить себе ногу таким способом.
* Не бойтесь создавать мусор. Сборщик собирает мусор, так зачем отбирать у него работу?
* Следите за производительностью, потому что вы можете случайно сделать что-нибудь не то и получить последствия в неожиданном месте.
* Если у вас не SPA, то можно ничего не предпринимать, потому что выигрыш 1 кадра зависания может не стоить той работы, которая будет затрачена.
* Большая часть ошибок, сомнительных мест и бесполезной работы создается из-за непонимания работы инструмента.

Контакты спикера Андрея Роенко: [flapenguin.me](https://flapenguin.me/), [Twitter](https://twitter.com/fla_penguin), [GitHub](https://github.com/flapenguin).