Частное учреждение образования

«Колледж бизнеса и права»

|  |
| --- |
| УТВЕРЖДАЮ  Заведующий  методическим кабинетом  \_\_\_\_\_\_\_\_\_\_Е.В.Паскал  «\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_ |

|  |  |
| --- | --- |
| Специальность:  2-40 01 01 «Программное обеспечение информационных технологий» | Дисциплина: «Скриптовые языки программирования» |

**Лабораторная работа № 12**

Инструкционно-технологическая карта

Тема: Сборщик мусора. Освобождение памяти.

Цель: Научиться использовать сборщик мусора для освобождения памяти.

Время выполнения: 2 часа

# Порядок выполнения работы

1. Изучить теоретические сведения к лабораторной работе.

2. Реализовать алгоритм решения задачи.

3. Отлаженную, работающую программу сдать преподавателю. Работу программы показать с помощью самостоятельно разработанных тестов.

4. Ответить на контрольные вопросы.

# Теоретические сведения

Управление памятью в JavaScript выполняется автоматически и незаметно создаём примитивы, объекты, функции… Всё это занимает память.

Но что происходит, когда что-то больше не нужно? Как JavaScript понимает, что пора очищать память?

## [Достижимость](https://learn.javascript.ru/garbage-collection" \l "dostizhimost)

Основной концепцией управления памятью в JavaScript является принцип достижимости.

Если упростить, то «достижимые» значения – это те, которые доступны или используются. Они гарантированно находятся в памяти.

1. Существует базовое множество достижимых значений, которые не могут быть удалены.

Например:

* 1. Локальные переменные и параметры текущей функции.
  2. Переменные и параметры других функций в текущей цепочке вложенных вызовов.
  3. Глобальные переменные.
  4. (некоторые другие внутренние значения)

Эти значения можно называть **корнями**.

1. Любое другое значение считается достижимым, если оно доступно из корня по ссылке или по цепочке ссылок.

Например, если в локальной переменной есть объект, и он имеет свойство, в котором хранится ссылка на другой объект, то этот объект считается достижимым. И те, на которые он ссылается, тоже достижимы. Далее будет знакомство с подробными примерами на эту тему.

В интерпретаторе JavaScript есть фоновый процесс, который называется «сборщик мусора». Он следит за всеми объектами и удаляет те, которые стали недостижимы.

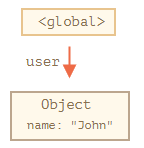
Вот самый простой пример:

// в user находится ссылка на объект

let user = {

    name: "John"

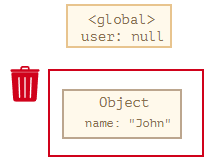
  };



Здесь стрелка обозначает ссылку на объект. Глобальная переменная user ссылается на объект {name: "John"} (можно называть его просто «John»). В свойстве "name" объекта John хранится примитив, поэтому оно нарисовано внутри объекта.

Если перезаписать значение user, то ссылка потеряется:

  user = null;



Теперь объект John становится недостижимым. К нему нет доступа, на него нет ссылок. Сборщик мусора удалит эти данные и освободит память.

## Две ссылки

Что происходит при копировании ссылки из user в admin:

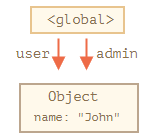
// в user находится ссылка на объект

let user = {

    name: "John"

  };

  let admin = user;



Теперь, если сделать то же самое:

user = null;

то объект John всё ещё достижим через глобальную переменную admin, поэтому он находится в памяти. Если перезаписать admin, то John был бы удалён.

## Взаимосвязанные объекты

Теперь более сложный пример. Семья:

function marry(man, woman) {

    woman.husband = man;

    man.wife = woman;

    return {

      father: man,

      mother: woman

    }

  }

  let family = marry({

    name: "John"

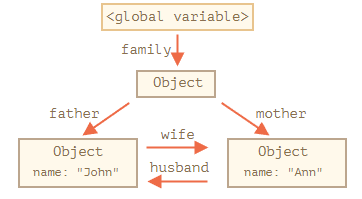
  }, {

    name: "Ann"

  });

Функция **marry** «женит» два объекта, давая им ссылки друг на друга, и возвращает новый объект, содержащий ссылки на два предыдущих.

В результате получается такая структуру памяти:



На данный момент все объекты достижимы.

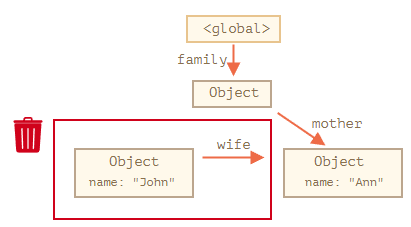
Теперь нужно удалить две ссылки:

delete family.father;

delete family.mother.husband;

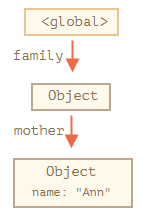
Недостаточно удалить только одну из этих ссылок, потому что все объекты останутся достижимыми.

Если удалить обе, то можно увидеть, что у объекта John больше нет входящих ссылок:



Исходящие ссылки не имеют значения. Только входящие ссылки могут сделать объект достижимым. Объект John теперь недостижим и будет удалён из памяти со всеми своими данными, которые также стали недоступны.

После сборки мусора:



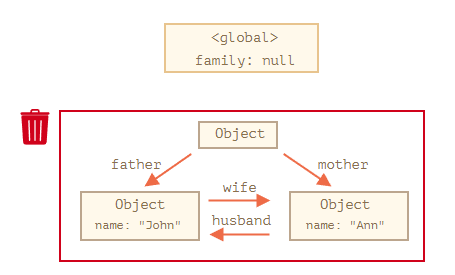
## Недостижимый «остров»

Вполне возможна ситуация, при которой целый «остров» связанных объектов может стать недостижимым и удалиться из памяти.

Нужно взять объект family из примера выше. А затем:

family = null;

Структура в памяти теперь станет такой:



Этот пример демонстрирует, насколько важна концепция достижимости.

Объекты John и Ann всё ещё связаны, оба имеют входящие ссылки, но этого недостаточно.

У объекта family больше нет ссылки от корня, поэтому весь «остров» становится недостижимым и будет удалён.

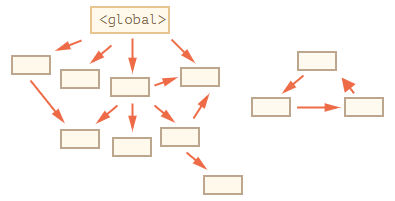
## Внутренние алгоритмы

Основной алгоритм сборки мусора – «алгоритм пометок» (англ. «mark-and-sweep»).

Согласно этому алгоритму, сборщик мусора регулярно выполняет следующие шаги:

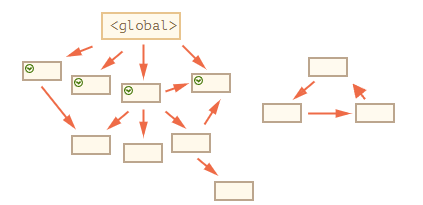
1. Сборщик мусора «помечает» (запоминает) все корневые объекты.
2. Затем он идёт по их ссылкам и помечает все найденные объекты.
3. Затем он идёт по ссылкам помеченных объектов и помечает объекты, на которые есть ссылка от них. Все объекты запоминаются, чтобы в будущем не посещать один и тот же объект дважды.
4. …И так далее, пока не будут посещены все ссылки (достижимые от корней).
5. Все непомеченные объекты удаляются.

Например, пусть наша структура объектов выглядит так:

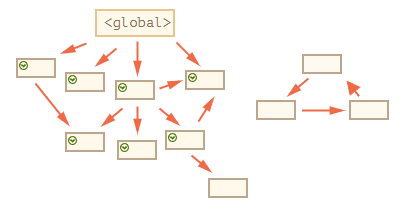


Явно виден «недостижимый остров» справа. Теперь посмотрим, как будет работать «алгоритм пометок» сборщика мусора.

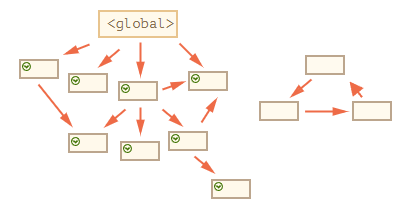
На первом шаге помечаются корни:



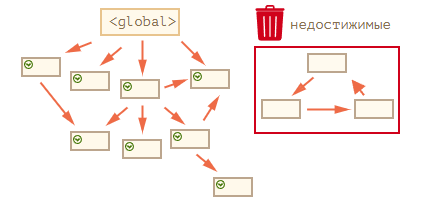
Затем помечаются объекты по их ссылкам:



А затем объекты по их ссылкам и так далее, пока это вообще возможно:



Теперь объекты, до которых не удалось дойти от корней, считаются недостижимыми и будут удалены:



Это и есть принцип работы сборки мусора.

Интерпретаторы JavaScript применяют множество оптимизаций, чтобы сборка мусора работала быстрее и не влияла на производительность.

Вот некоторые из оптимизаций:

**Сборка по поколениям (Generational collection)** – объекты делятся на «новые» и «старые». Многие объекты появляются, выполняют свою задачу и быстро умирают, их можно удалять более агрессивно. Те, которые живут достаточно долго, становятся «старыми» и проверяются реже.

**Инкрементальная сборка (Incremental collection)** – если объектов много, то обход всех ссылок и пометка достижимых объектов может занять значительное время и привести к видимым задержкам выполнения скрипта. Поэтому интерпретатор пытается организовать сборку мусора поэтапно. Этапы выполняются по отдельности один за другим. Это требует дополнительного учёта для отслеживания изменений между этапами, но зато теперь есть много крошечных задержек вместо одной большой.

**Сборка в свободное время (Idle-time collection)** – чтобы уменьшить возможное влияние на производительность, сборщик мусора старается работать только во время простоя процессора.

Существуют и другие способы оптимизации и разновидности алгоритмов сборки мусора. И, что более важно, всё меняется по мере развития интерпретаторов, поэтому углубляться в эту тему заранее, без реальной необходимости, вероятно, не стоит. Если же это не вопрос чистого интереса, тогда будут полезны некоторые ссылки ниже.

#### Как узнать потребление памяти

Чтобы узнать потребление памяти в node.js используется

process.memoryUsage().heapUsed

Пример:

let arr = Array(1e6).fill("");

let used = process.memoryUsage().heapUsed / 1024 / 1024;

console.log(`The script uses approximately ${used} MB`);

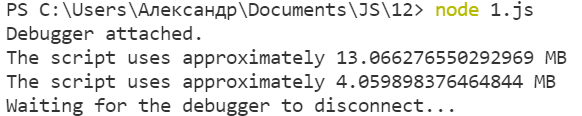
arr = null;

setTimeout(() => {

    used = process.memoryUsage().heapUsed / 1024 / 1024;

    console.log(`The script uses approximately ${used} MB`);

}, 10000);



## Итого

О главном:

* Сборка мусора выполняется автоматически. Её нельзя ускорить или предотвратить.
* Объекты сохраняются в памяти, пока они достижимы.
* Наличие ссылки не гарантирует, что объект достижим (от корня): несколько взаимосвязанных объектов могут стать недостижимыми как единое целое.

Современные интерпретаторы реализуют передовые алгоритмы сборки мусора.

Некоторые из них освещены в книге «The Garbage Collection Handbook: The Art of Automatic Memory Management» (R. Jones и др.).

Глубокое понимание работы интерпретаторов необходимо, когда нужны низкоуровневые оптимизации. Было бы разумно запланировать их изучение как следующий шаг после освоения языка.

***Варианты индивидуальных заданий:***

**Задание 1.**

Реализовать массив объектов из заданий лабораторной работы № 11, затем, спустя две секунды удалить копию.

Произвести отладку в браузере на вкладке memory, наблюдая за объёмом потребляемой памяти.

Произвести отладку в node.js.

# Контрольные вопросы

1. Что такое достижимость?
2. Что такое недостижимый остров?
3. До какого момента объекты не удаляются из памяти?
4. Почему в последнем примере перед тем, как повторно выводить объём потребляемой памяти, нужно ждать некоторое время?

**Литература**

**Диков, А.В.** Клиентские технологии веб программирования: JavaScript и DOM: учебное пособие / А.В. Диков. – СПб: Лань, 2020 – 124 с.

**Читанамбри, Кирупа.** Изучаем React / Кирупа Читанамбри – 2-е изд. – М.: Эксмо, 2019. 368 с.

**Хавербеке, Марейн.** Выразительный JavaScript. Современное веб-программирование. / Марейн Хавербеке – 3-е изд. – СПб.: Питер, 2019. – 480 с.:

Преподаватель Рогалевич А. В.

Рассмотрено на заседании цикловой

комиссии программного обеспечения

информационных технологий №10

Протокол № \_\_\_ от \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Председатель ЦК\_\_\_\_\_\_\_\_ В.Ю.Михалевич