زباله روبی (Garbage Collection)

مدیریت حافظه (Memory) در جاوااسکریپت به صورت خودکار صورت پذیرفته و ما آن را نمی‌بینیم. ما primitiveها ، objectها ، functionها و ... را استفاده می‌کنیم که همه اینها از حافظه استفاده می‌کنند.

وقتی دیگر به چیزی نیاز نباشد ، چه اتفاقی برای آن می‌افتد؟ جاوااسکریپت چطور این موضوع را تشخیص داده و حافظه را پاک می‌کند؟

قابلیت دسترسی (Reachability)

اصلی‌ترین مفهموم مدیریت حافظه در جاوا اسکریپت ، reachability است.

در بیان ساده ، مقادیر reachable به مقادیری می‌گویند که به طریقی قابل دسترسی یا قابل استفاده کردن بوده و در حافطه وجود دارند. که شامل موارد زیر می‌شود:

۱. مجموعه‌ای پایه از مقادیر ذاتا قابل دسترسی وجود دارد که به دلایل مشخص نمی‌توانند حذف شوند.

برای مثال :

- فانکشن جاری در حال اجرا ، متغیرهای محلی آن و پارامترهایش
- باقی فانکشن‌ها در زنجیره فانکشن‌های فراخوانی شده فعلی ، متغیرهای محلی آنها و پارامترهایشان.
- متغیرهای Global
- (تعداد دیگری نیز وجود دارند ازجمله متغیرهای و فانکشن‌های داخلی مورد نیاز خود جاوا اسکریپت)

به این مقادیر roots می‌گویند.

۲. هر مقداری که از طریق یک root یا زنجیره‌ای از referenceها قابل دسترسی باشد.

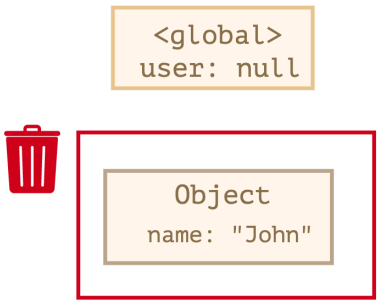
برای مثال ، اگر درون یک متغیر global ، objectای ذخیره شود و آن object نیز به object دیگری اشاره کند ، object دوم reachable در نظر گرفته می‌شود و همین‌طور الی آخر. در ادامه این موضوع با مثالی شرح داده شده است.

در engine جاوا اسکریپت پروسس پس‌زمینه‌ای وجود دارد که به آن garbage collector می‌گویند و وظیفه آن این است که همه objectها را بررسی کرده و آنهایی که غیر قابل دسترسی شده‌اند را از حافظه پاک می‌کند.

یک مثال ساده

```
// user has a reference to the object
let user = {
  name: "John"
};
```

در اینجا علامت پیکان (جهت) ، یک ارجاع به object را نمایش می‌دهد . متغیر global با نام user به {name: “john”} اشاره می‌کند (برای خلاصه فقط به آن john می‌گوییم) . مشخصه name از john یک مقدار primitive ذخیره می‌کند پس درون object ترسیم شده است . اگر مقدار user بازنویسی شود ، ارجاع گم می‌شود :

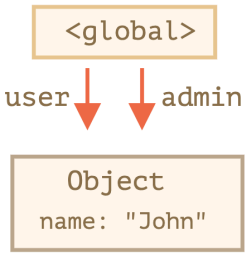


حال ، john غیر قابل دسترس شده است . هیچ راهی برای دسترسی به آن وجود نداشته و هیچ ارجاعی نیز به آن وجود ندارد بنابراین garbage collector دیتا را پاک کرده و حافظه را آزاد می‌کند .

دو ارجاع

حال تصور کنید که ارجاعی از user به admin ایجاد کرده‌ایم .

```
// user has a reference to the object
let user = {
  name: "John"
};
```




```
let admin = user;
```

حال اگر کار مشابه مثال قبل را انجام دهیم :

```
user = null;
```

می‌بینیم که object مد نظر همچنان از طریق متغیر global با نام admin قابل دسترس است پس همچنان در حافظه وجود دارد.
اگر admin را نیز بازنویسی کنیم ، آن موقع می‌توان آن را از حافظه پاک نمود.

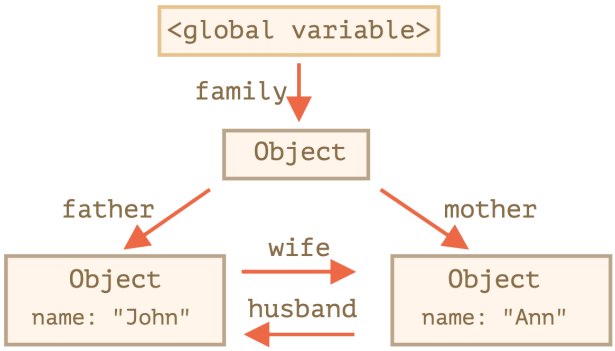
Objectهای به هم پیوسته

به مثال پیچیده‌تر زیر توجه نمایید :

```
function marry(man, woman) {
  woman.husband = man;
  man.wife = woman;
  return {
    father: man,
    mother: woman
  }
}

let family = marry({
  name: "John"
}, {
  name: "Ann"
});
```

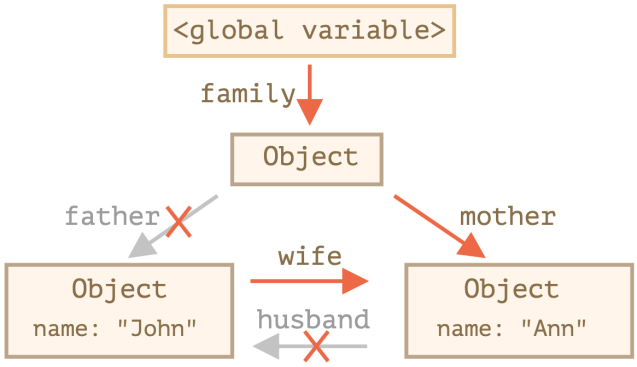

فانکشن marry دو objectای که از طریق reference به آن داده شده است را به عقد یکدیگر درآورده و object جدیدی شامل هر دوی آنها باز می‌گرداند.
ساختار حافظه آن به شکل زیر خواهد شد :



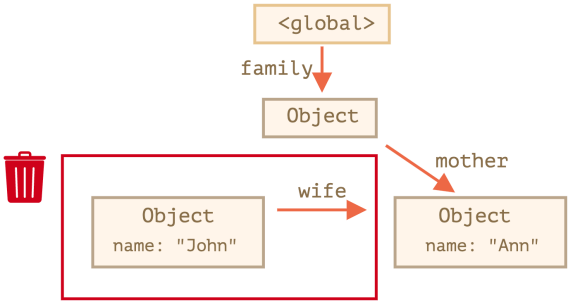
در حا حاضر ، تمام objectها reachable هستند .

حال دو ارجاع را پاک می‌کنیم :

```
delete family.father;
delete family.mother.husband;
```

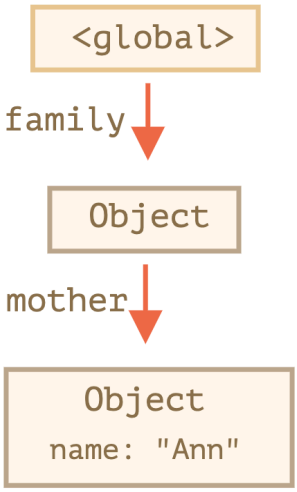



پاک کردن تنها یکی از این دو ارجاع کافی نمی‌باشد چرا که تمامی objectها همچنان قابل دسترس‌اند . ولی اگر هر دو را پاک کنیم میبینیم که ارجاع دیگری به john وجود ندارد .



ارجاع خروجی مهم نیست . تنها ارجاع‌های ورودی باعث می‌شوند که یک object ، قابل دسترس شود . پس ، john و همه دیتاهای آن اکنون غیر قابل دسترس بوده و از حافظه پاک خواهند شد .

بعد از garbage collection (زباله روبی) ، حافظه بدین شکل خواهد بود :



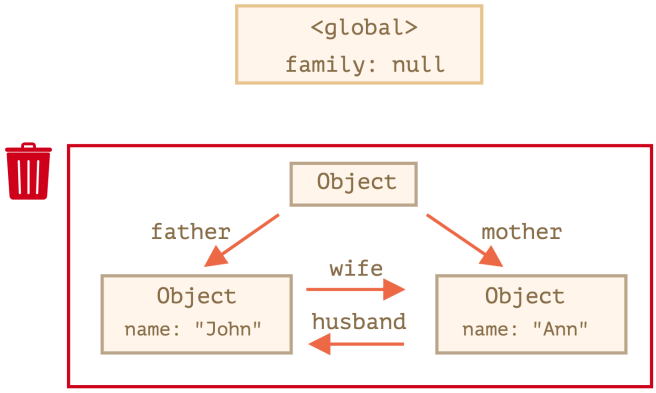
جزیره غیر قابل دستیابی

این امکان وجود دارد که جزیره‌ای از objectهای به هم پیوسته ، غیر قابل دستیابی شده و از حافظه پاک شوند .

مثال بالا را در نظر بگیرید ، اگر بنویسیم :

```
family = null;
```

ساختار حافظه بدین شکل خواهد شد :



این مثال نشان می‌دهد که مفهوم reachability چه قدر مهم است .

واضح است که john و ann همچنان به یکدیگر لینک شده‌اند و هر دو ارجاع ورودی دارند ولی این کافی نمی‌باشد .

در این حالت family از root قطع شده و ارجاعی به آن وجود ندارد پس تمام جزیره غیر قابل دسترس شده و از حافظه پاک می شود .

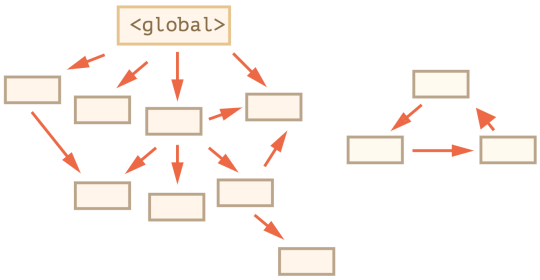
الگوریتم‌های داخلی

الگوریتم garbage collection پایه در جاوااسکریپت mark-and-sweep نام دارد .

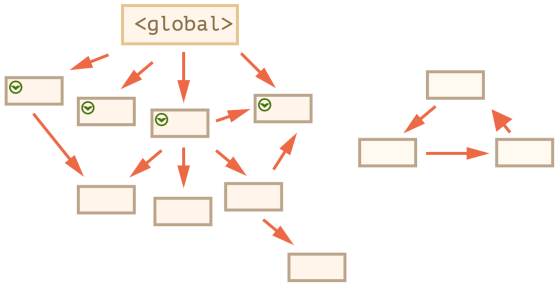
در فرآیند garbage collection ، مراحل زیر به طور منظم انجام می‌شود :

- garbage collector ، rootها را پیدا کرده و آنها را علامتگذاری (mark) می‌کند . (به خاطر می‌سپارد)
- سپس تمام ارجاع‌هایی که از آنها ایجاد شده است را یافته و mark می‌کند .
- سپس به سراغ ارجاع‌های یافته شده رفته و ارجاع‌هایی که از آنها شده است را mark می‌کند . تمام objectهای که mark شده به خاطر سپرده شده تا در آینده یک object را دوبار mark نکند .
- این فرآیند تا جایی ادامه پیدا می‌کند که تمامی ارجاع‌های قابل دسترسی یافت شوند .
- تمامی objectها به جز آنهایی که mark شده اند ، از حافظه پاک می‌شوند .

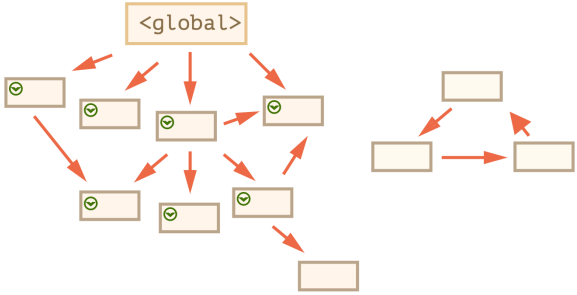
برای مثال ، فرض کنید که ساختار object ما به شکل زیر باشد :



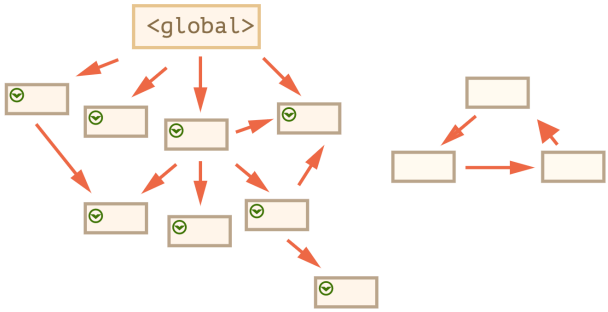
به طور واضح جزیره ای غیر قابل دسترسی را در سمت راست می‌بینیم . حال ببینیم که garbage collector چطور با آن برخورد می‌کند . در مرحله اول rootها ، mark می‌شوند :



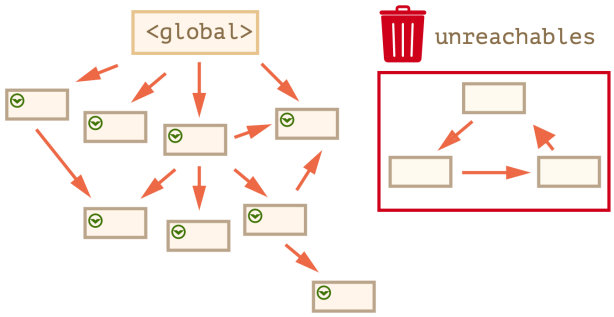
سپس ارجاع‌هایی که از آنها شده است mark می‌شوند :



و در ادامه تمامی ارجاع‌های ارجاع‌ها mark می‌شوند :



حال ، objectهایی که در طی این فرآیند mark نشده اند ، از حافظه پاک خواهند شد :



تـوضیح داده شـده فـوق ،نـحوه کار garbage collection را شـرح داد .همچنین engineهـای جـاوااسکریپت ممکن اسـت بهینه‌سازی‌هایی برای اجرای هرچه بهتر این فرآیند انجام دهند . بعضی از این بهینه سازی‌ها عبارتند از :

- Generational collection : objectها به دو دسته تقسیم می‌شوند : جدیدها و قدیمی‌ها . خیلی از objectها وظیفه خود را انجام داده و به سرعت نابود می‌شوند . آنهایی که باقی می‌مانند ، قدیمی نامیده شده و کمتر مورد بررسی قرار می گیرند .
- Incremental collection : اگر تعداد زیادی object وجود داشته باشد و ما بخواهیم همه آنها را در یک مرحله یافته و mark کنیم ، زمـان زیادی طـول کشیده و delay قـابـل تـوجهی در اجـرا ایجاد می‌شـود . پـس engineهـا سعی می‌کنند garbage collection را تقسیم بـندی کرده و آنـها را تک بـه تک و بـه صـورت جـداگـانـه اجـرا نمـایند . البته این کار مستلزم مدیریت بهتری برای نگهداری objectها و تغییرات آنها می‌باشد ولی در عوض به جای یک delay بزرگ ، چندین delay کوچک خواهیم داشت .
- Idle-time collection : garbage collector فقط زمانی فعایت خود را شروع می‌کند که CPU بیکار است و این باعث می‌شود که تاثیر جانبی آن کمتر شده و سریعتر انجام شود .

الـگوریتم‌هـای بهینه سـازی و garbage collection مختلفی وجـود دارد که تـوضیح آن خـارج از تـوانـایی این سـری آمـوزش‌هـا می‌باشد .

همچنین در فرآیند توسعه ، خیلی چیزها تغییر می‌کند پس مطالعه عمیق مطالب بدون نیاز به استفاده از آنها ، ارزش نخواهد داشت .

خلاصه :

نکات مهمی که باید بدانید :

- فرآیند garbage collection به صورت خودکار اتفاق می‌افتد . نمی‌توانید آن را اجرا کرده و یا مانع آن شوید .
- objectها تا زمانی که قابل دسترس باشند ، در حافظه باقی می‌مانند .
- ارجاع به یک object به معنای قابل دسترس بودن آن نیست (از root) . به همین خاطر گاهی اوقات مجموعه‌ای از objectهای به هم پیوسته ممکن است غیر قابل دسترس باشند .

engineهای مدرن ، الگوریتم‌های قویتری برای garbage collection پیاده سازی کرده‌اند .

در ادامه برخی از منابع مفید در این خصوص را خواهید یافت :

- "The Garbage Collection Handbook: The Art of Automatic Memory Management"
- http://jayconrod.com/posts/55/a-tour-of-v8-garbage-collection
- https://v8.dev/
- http://mrale.ph/

شناخت عمیق از engineها زمانی مفید واقع خواهد شد که بخواهید بهینه سازی‌های low-level انجام دهید . اگر بعد از یادگیری کامل زبان جاوا اسکریپت ، آن را نیز فرا بگیرید ، قطعا مفید خواهد بود .