Univerza v Mariboru

Fakulteta za elektrotehniko, računalništvo in informatiko

Spletne tehnologije

Splošno o predmetu

Niko Lukač

Kontakt

- Predavanje + vaje:

  izr. prof. dr. Niko Lukač

  niko.lukac@um.si

- Vso učno gradivo, obvestila in vaje so na Eštudiju (https://estudij.um.si)
- Govorilne ure: https://aips.um.si/GovorilneUre.aspx
- Najavo na govorilnih urah predhodno sporočite po e-mailu.

VAJE

- **Skupine vaj** (možne spremembe na urniku)

RV 1 - četrtek ob 11:00 do 12:30 - F-104

RV 2 - četrtek ob 17:00 do 18:30 - F-102

RV 3 - četrtek ob 18:30 do 20:00 - F-102

RV 4 - torek ob 17:00 do 18:30 - G3-Lumiere (klet)

- Izberite skupino na estudij.um.si (praviloma 20 mest na skupino)

- Beleženje prisotnosti, ocenjevanje/zagovor vaj, konzultacije

VAJE

- **Termini vaj, ocenjevanje vaj in kolokvijev** (možne spremembe)

07.10-13.10 - Razlaga 1. vaje
14.10-20.10 - Konzultacije
21.10-03.11 - Konzultacije
04.11-10.11 - Razlaga 2. vaje
11.11-17.11 - Konzultacije + ocenjevanje oddanih vaj
18.11-24.11 - Konzultacije
25.11-01.12 - 1. kolokvij + Konzultacije
02.12-08.12 - Razlaga 3. vaje
09.12-15.12 - Konzultacije + ocenjevanje oddanih vaj
16.12-22.12 - Razlaga 4. vaje
06.01-12.01 - Konzultacije
14.01-19.01 - 2. kolokvij + ocenjevanje oddanih vaj
18.01-26.01 - Konzultacije + ocenjevanje oddanih vaj

x.x.2025 (2 tedna pred 1. izpitnim rokom) - ocenjevanje oddanih vaj (ni možnih daljših konzultacij/razlag po koncu semestra!)
x.x.2025 (2 tedna pred 2. izpitnim rokom) - ocenjevanje oddanih vaj
x.x.2025 (2 tedna pred 3. izpitnim rokom) - ocenjevanje oddanih vaj

Končna ocena

- Študent mora opraviti vsaj 50 % obveznosti pri laboratorijskih vajah, če želi pristopiti k pisnemu izpitu oz. uveljaviti oceno iz kolokvijev.
- Pisni del izpita se opravi, če iz obeh kolokvijev dosežete 50% teoretičnega dela. Pri tem pa mora biti pri vsakem kolokviju doseženih vsaj 35%.
- Za pozitivno oceno je treba zbrati skupaj vsaj 50 %.
- Pravilnik Inštituta za računalništvo: https://cs.feri.um.si/site/assets/files/1037/ocenjevanje_pri_predmetih_in_projektih-2018-2019.pdf

1. kolokvij: 25%

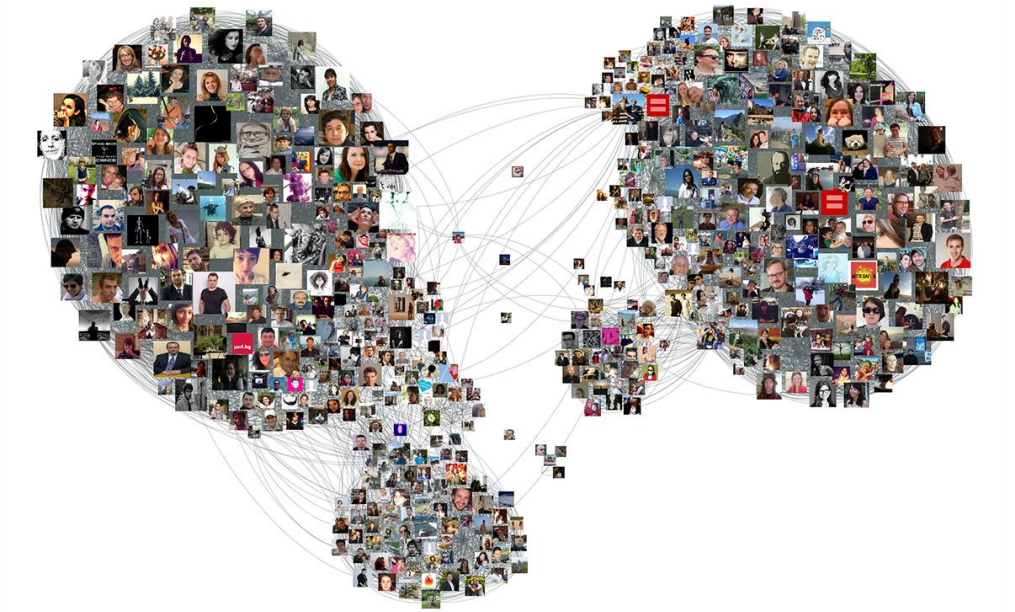
2. kolokvij : 25%

Vaje: 50%

Vsebina predavanj

- Pregled front-end spletnih rešitev
- Uvod v delovanje JS pogonov, optimizacije JS, WebAssembly
- Pregled back-end spletnih rešitev
- Spletni sledilniki, zasebnost in varnost

- Algoritmi spletnih iskalnikov
- Algoritmi za uvrstitev in indeksiranje spletnih virov
- Spletni priporočilni sistemi
- Spletna omrežja in algoritmi gručenja spletnih omrežij

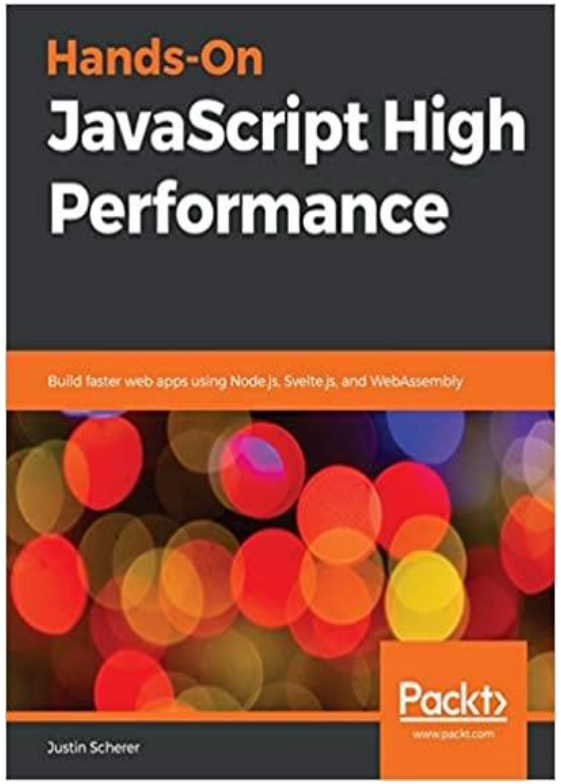


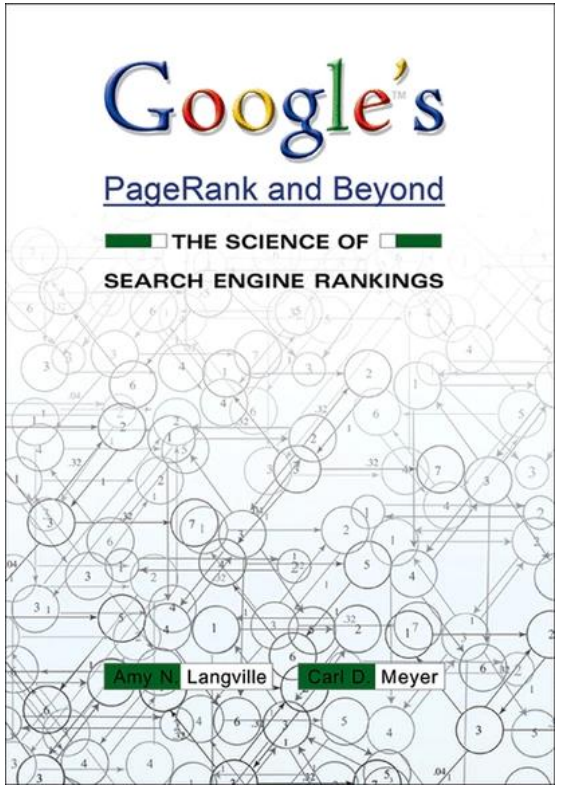
Vir slike: https://kateto.net/2014/04/facebook-data-collection-and-photo-network-visualization-with-gephi-and-r/

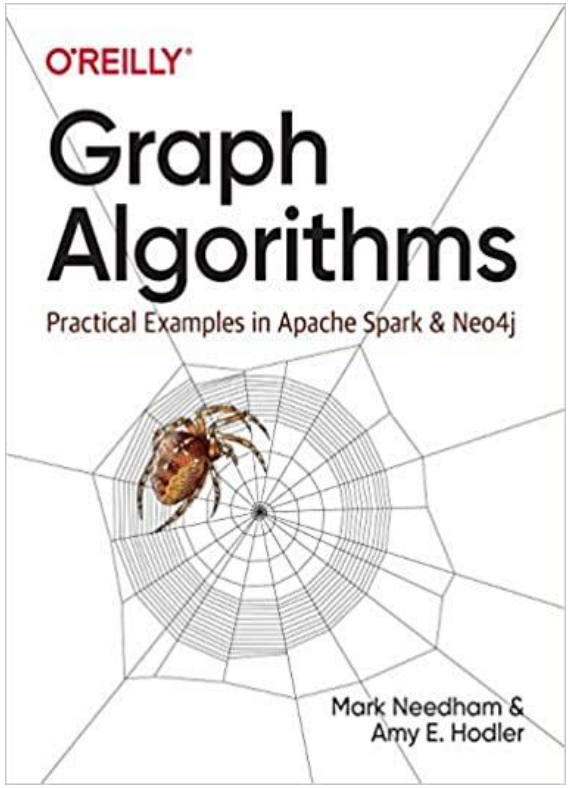
Učno gradivo

- Prosojnice iz predavanj + dve zbirki nalog (e-gradivi)

  +







Univerza v Mariboru

Fakulteta za elektrotehniko, računalništvo in informatiko

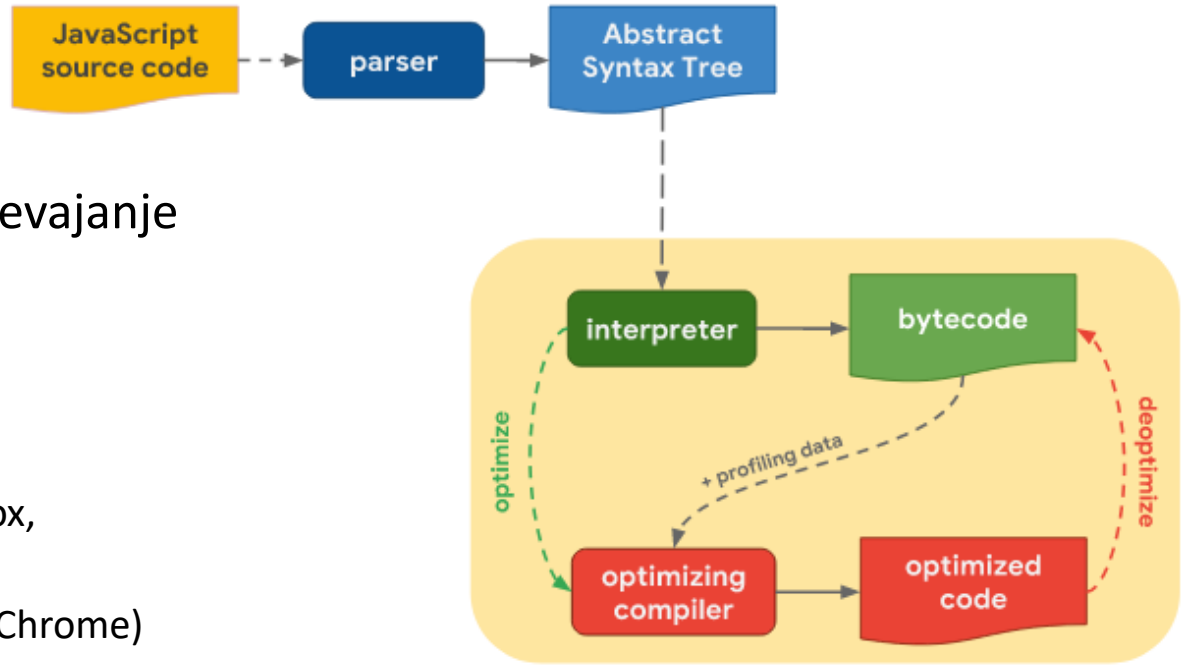
Spletne tehnologije

Uvod v JavaScript pogone

Niko Lukač

JavaScript pogoni

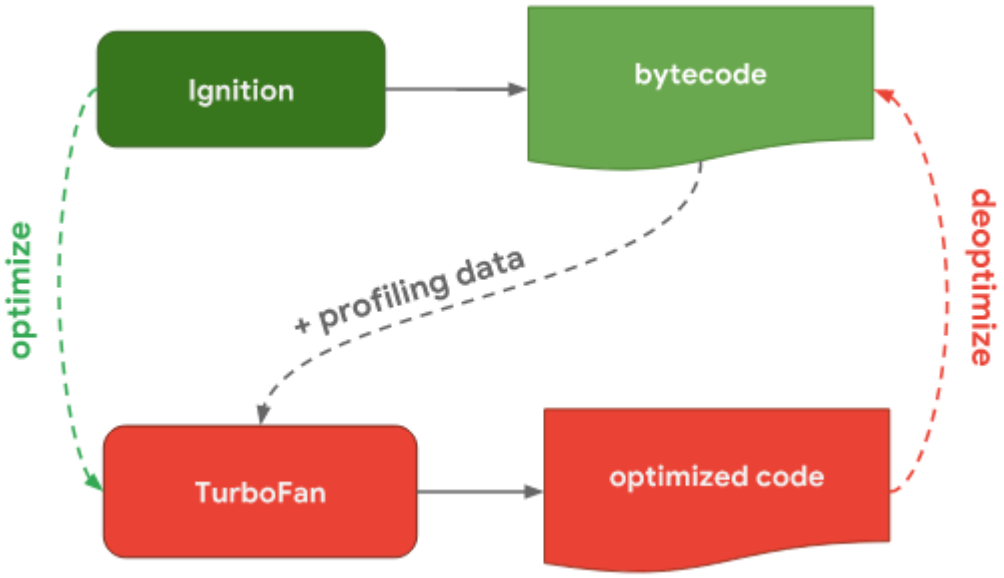
- Delovanje JS pogonov (engines) je običajno zelo podobno, kjer se znotraj virtualnega stroja izvedejo naslednji koraki:
  1. Grajenje **AST** (leksikalna,sintaktična,semantična analiza)
  2. Zagon **interpreterja** -> dobimo zložno kodo (**bytecode**)
  3. Profiliranje zložne kode in **JIT** (just-in-time) prevajanje v strojno optimizirano kodo
- Top pogoni:
  - Google **V8 (**Chrome, Node.JS, Electron, Opera, Edge)
    - V8 ni striktno del Blink-a (rendering/browser engine)
  - Mozilla **SpiderMonkey** del Gecko-ja ali Quantum-a (Firefox, SpiderNode)
  - Apple **JavaScriptCore**/Nitro del WebKit-a (Safari, starejši Chrome)
  - Microsoft **Chakra** (IE9, starejši Edge)



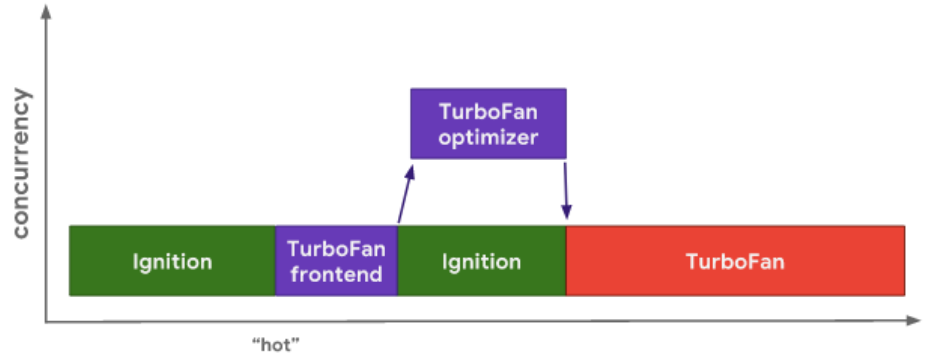
Vir slike: Mathias Bynens (https://mathiasbynens.be)

Cikel interpreter–prevajalnik (1 fazni)

- Primer cikla interpreterja (**Ignition**) in JIT prevajalnika (**TurboFan**) na V8 pogonu
- TurboFan **optimizira** in prevede "**vroče**" sekcije kode v strojno kodo
- V primeru **neučinkovite optimizacije** se lahko odseki kode **de-optimizirajo**



V8

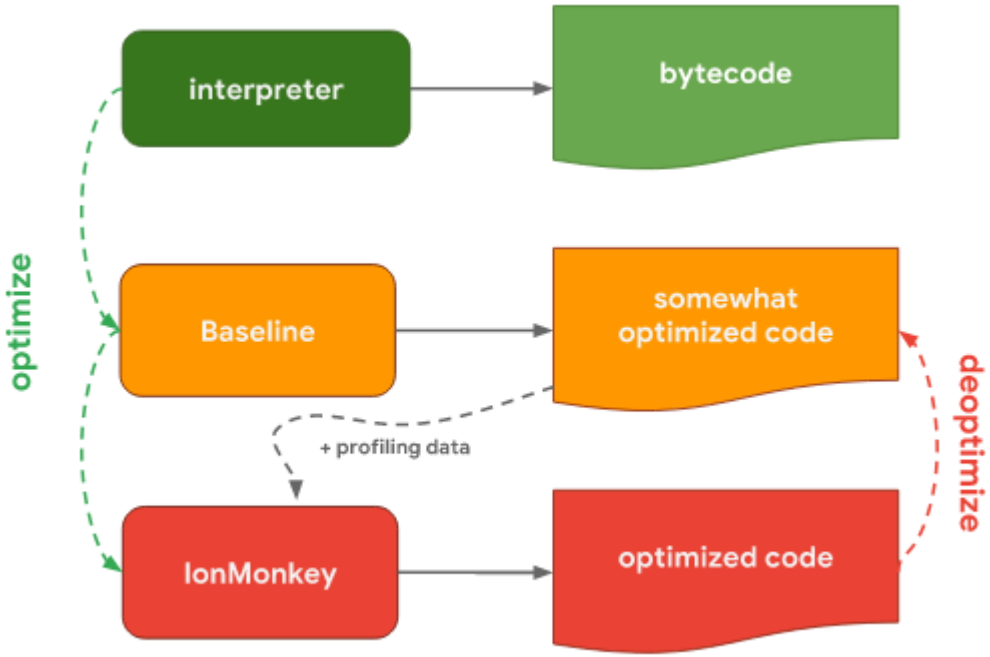


Večnitna implementacija pogona
(optimizacija deluje v ozadju!)

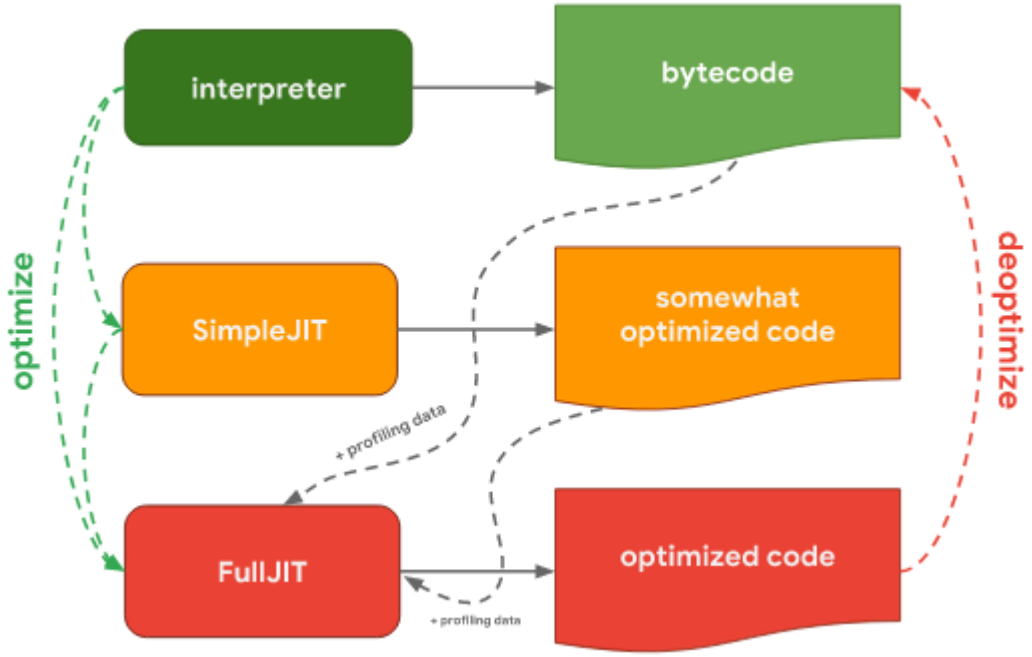
Vir slike: Mathias Bynens (https://mathiasbynens.be)

Cikel interpreter–prevajalnik (2 fazni)

- Primer cikla interpreterja in JIT prevajalnika pri SpiderMonkey in Chakra
- Vmes poteka **delno prevajanje in optimizacija** odseke kode (“**delno vroča koda**”)



SpiderMonkey

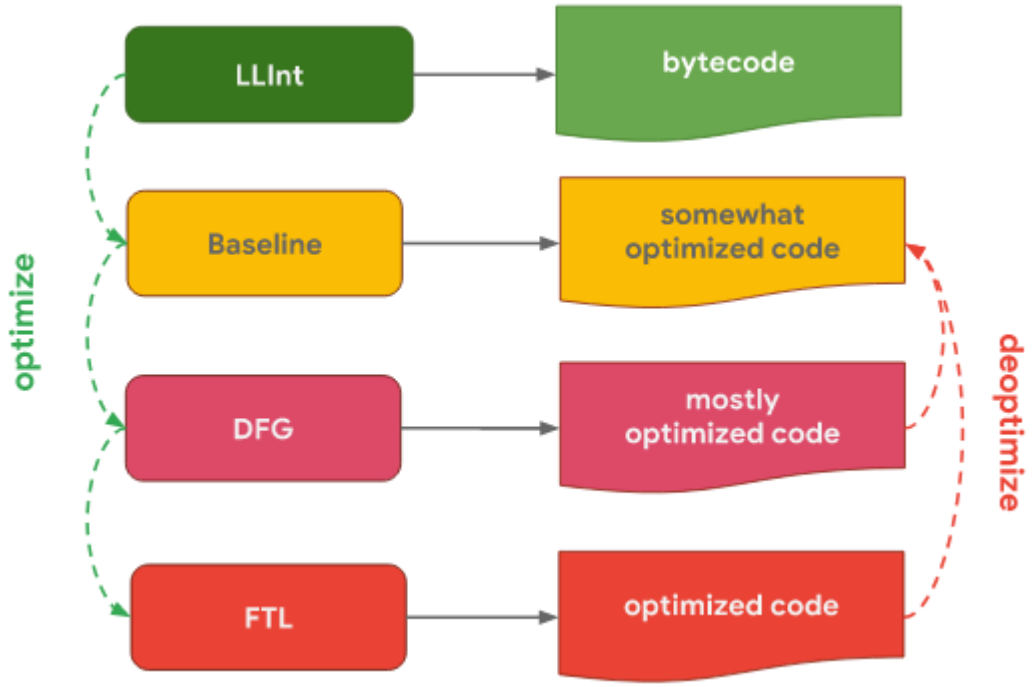


Chakra

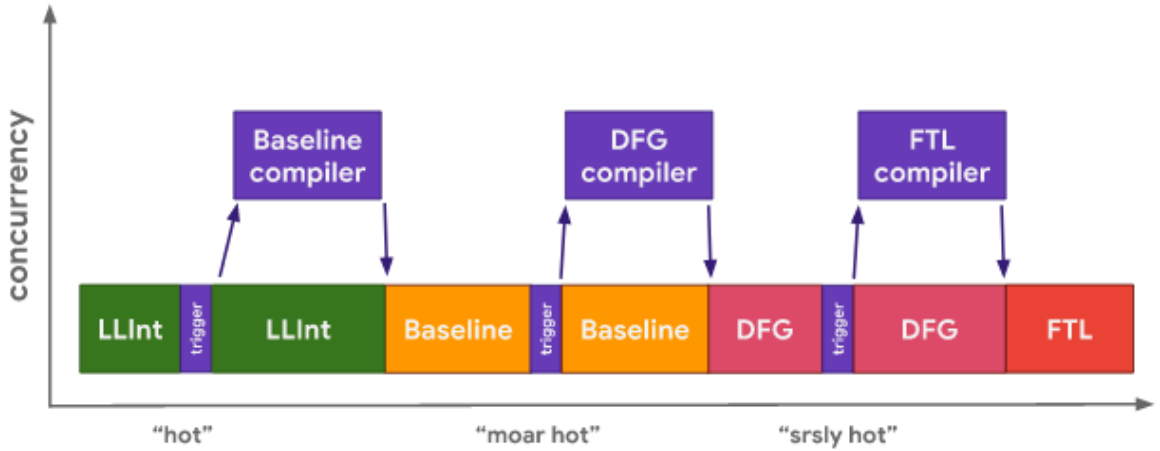
Vir slike: Mathias Bynens (https://mathiasbynens.be)

Cikel interpreter–prevajalnik (3 fazni)

- Primer pri Apple JavaScriptCore



JavaScriptCore

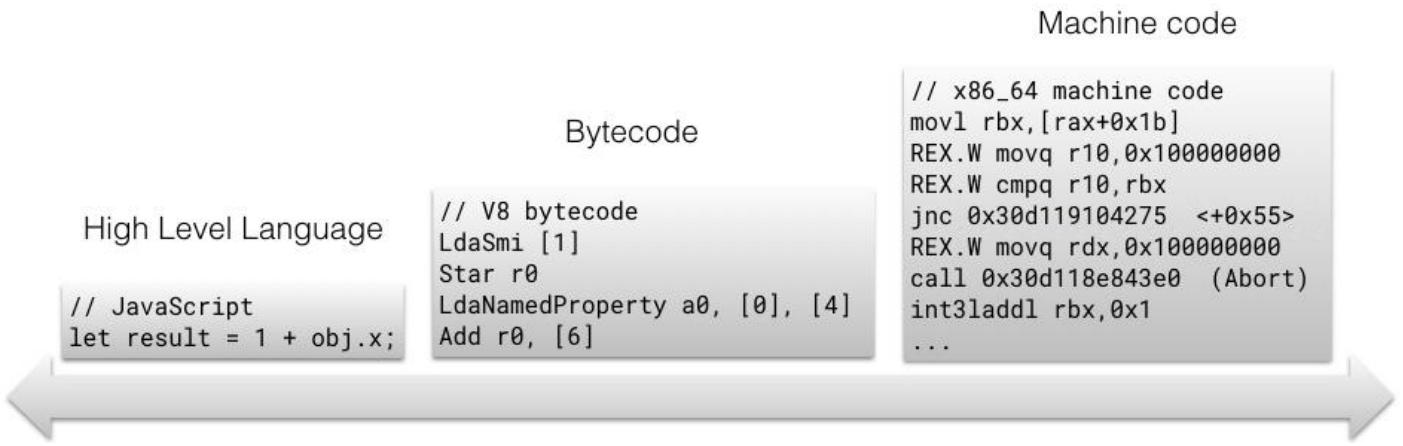


Večnitna implementacija pogona
(optimizacija deluje v ozadju!)

Vir slike: Mathias Bynens (https://mathiasbynens.be)

Optimizacija zložne kode JS

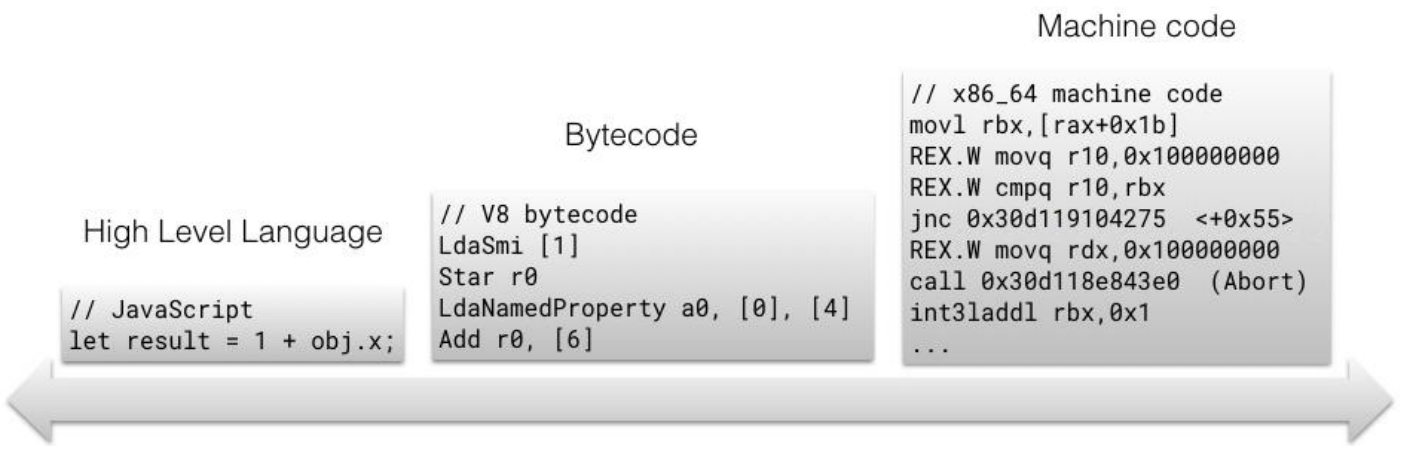
- Strojna koda vzame **več pomnilniškega prostora**, vendar omogoča hitrejše in direktno izvajanje na CPU
- Optimizacija kode tudi **traja dlje časa**
  - Kako dolgo pri intepreterju in prevajalniku, kakšen je nivo optimizacije?
- Prekomerna optimizacija ?

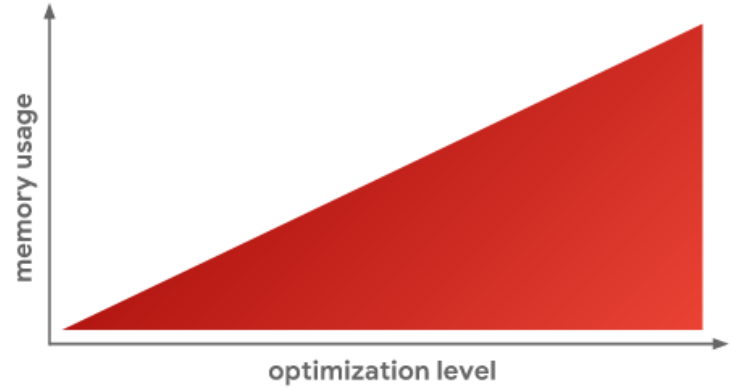


Vir slike: F. Hinkelmann, Google

Optimizacija zložne kode JS

- Strojna koda vzame več pomnilniškega prostora, vendar omogoča hitrejše in direktno izvajanje na CPU
- Optimizacija kode tudi traja dlje časa
  - Interpter: **hitro generiranje** (zložne) kode
  - Prevajalnik in optimizator: **generiranje hitre** (strojne) kode
- Prekomerna optimizacija zložne kode v strojno kodo (po JIT) lahko pripelje do prekomerne porabe pomnilnika





Vir slike: F. Hinkelmann, Google

Posledica internih implementacij JS

- Enostavni primer kode:

```
const X1 = { a: "A", b: "A"};
const X2 = { a: "B", b: "B"};
const X3 = { a: "C", b: "C"};
const X4 = { a: "D", b: "D"};
const X5 = { a: "E", b: ""};

const objekti = [ X1, X2, X3, X4, X5, X1, X2, X3];

const get_a = (bla) => bla.a;

for(var i = 0; i < 1000000000; i++)
        get_a(objekti[i & 7]);
```

Manjše spremembe:

```
const X1 = { a: "A", b: "A", c: "A"};
const X2 = { a: "B", b: "B", d: "B"};
const X3 = { a: "C", b: "C", e: "C"};
const X4 = { a: "D", b: "D", f: true};
const X5 = { a: "E"};

const objekti = [ X1, X2, X3, X4, X5, X1, X2, X3];

const get_a = (bla) => bla.a;

for(var i = 0; i < 1000000000; i++)
        get_a(objekti[i & 7]);
```

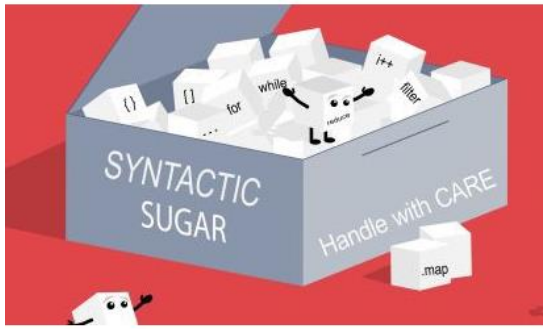
- Katera koda je hitrejša ? Druga koda bo v povprečju 2x počasnejša!
- Zakaj? Bomo pogledali interno delovanje JS pogonov

JS definiranje spremenljivk objektov

- **Spomnimo**: “razredi” objektov se simulirajo s ti. prototipi in funkcijami
- Tako lahko v JS smatramo skoraj vse kot **objekt**
- ECMAScript 6 (EC6) omogoča sintaktično “olepšavo” dane funkcionalnosti s class rezervirano besedo (**syntactic sugar**)

```
function Rectangle(height, width) {
 this.height = height
 this.width = width
}

let bla = new Rectangle(1, 2)
```

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}

let bla = new Rectangle(1, 2)
```

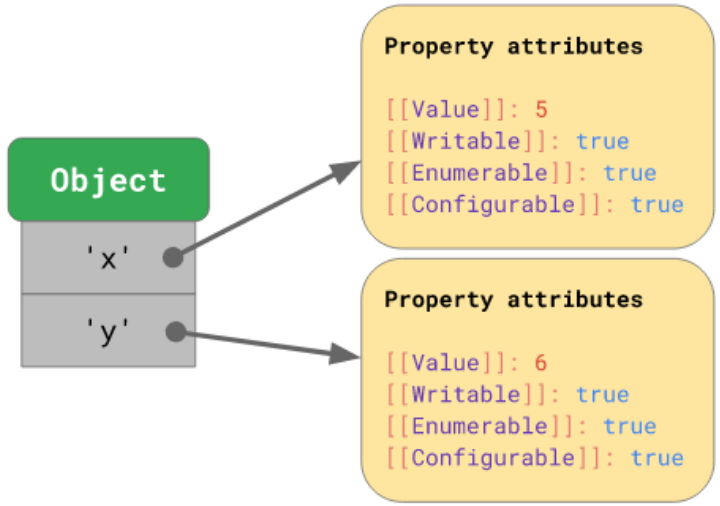


Vir slike: https://medium.com/front-end-weekly/syntactic-sugar-diabetes-alert-6329a7048cf5

JS definiranje spremenljivk objektov

- Po standardu ECMAscript so **vse spremenljivke** interno definirane kot **slovarji**
- Vsaka spremenljivka objekta (**property**) ima definirano **poimenovanje** v tekstovni (**string**) obliki
- Interno imajo spremenljivke objektov naslednje lastnosti:
  - [[Value]] vrednost,
  - [[Writable]] ali lahko spremenimo vrednost spremenljivke,
  - [[Enumerable]] ali je možno iterirat po spremenljivki (npr. v for zanki),
  - [[Configurable]] ali je možno izbrisat spremenljivko objekta
- **Debug API**: *Object.getOwnPropertyDescriptors*

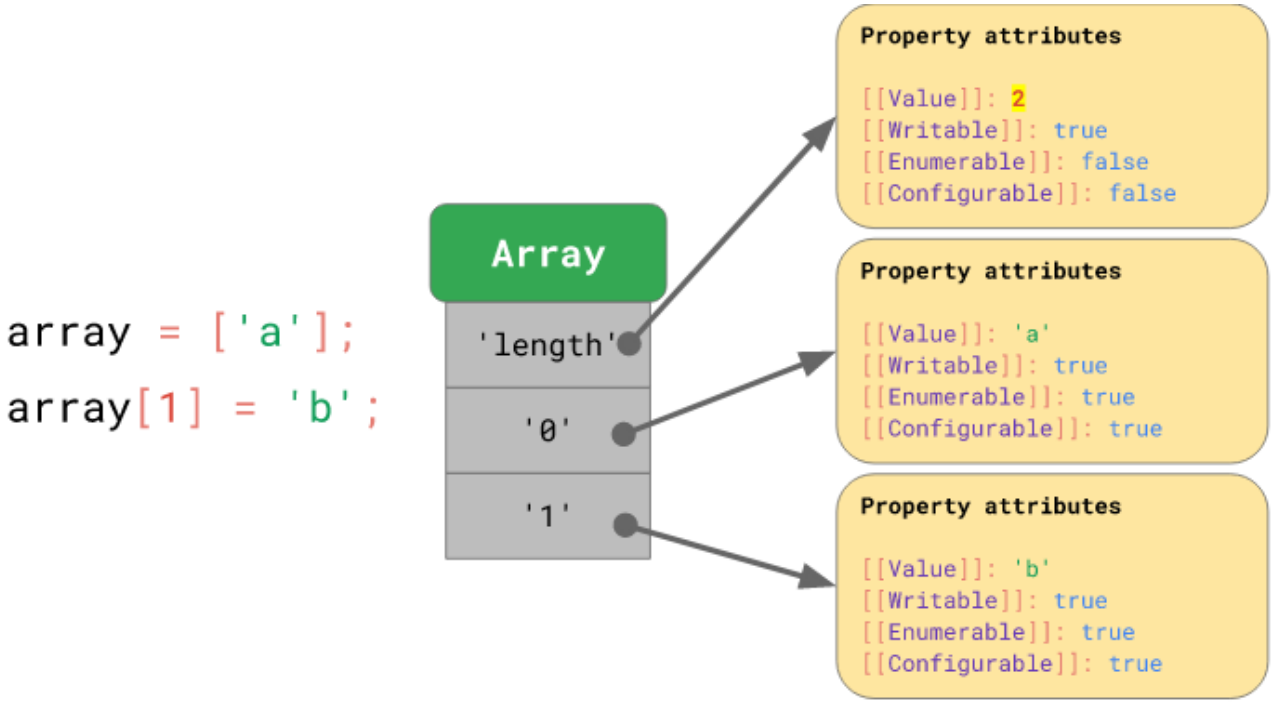
```
object = {
  x: 5,
  y: 6,
};
```



Vir slike: Mathias Bynens (https://mathiasbynens.be)

ECMAscript definiranje polj

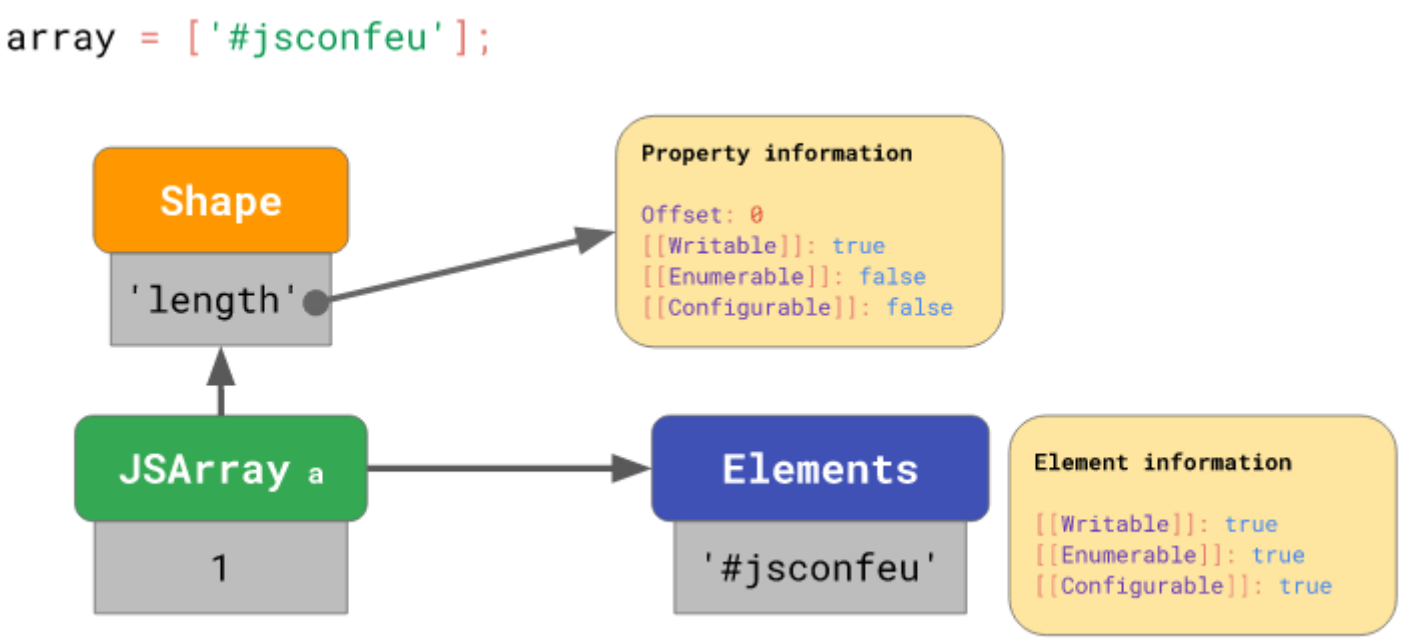
- Podobno kot spremenljivke objektov, dodatno še imamo lastnost '**length**'
- Generalno gledano se polja interno podobno obnašajo kot objekti



Vir slike: Mathias Bynens (https://mathiasbynens.be)

Optimizirano definiranje polj

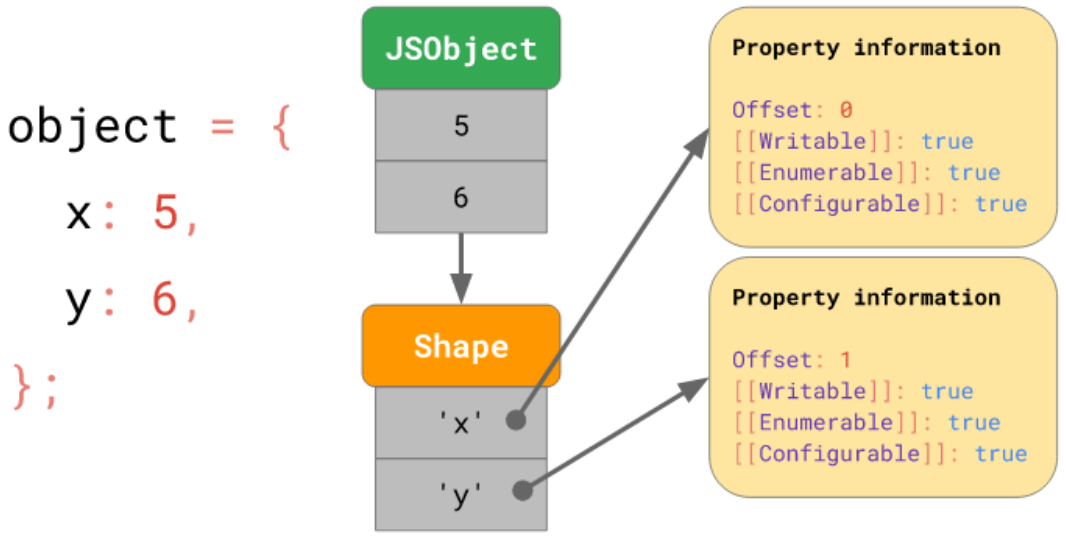
- JS pogoni nekoliko drugače hranijo polja interno (pomnilniška optimizacija)
- Vse vrednosti v polju imajo **enake interne lastnosti** (Writable,Enumerable,Configurable)



Vir slike: Mathias Bynens (https://mathiasbynens.be)

Oblike objektov

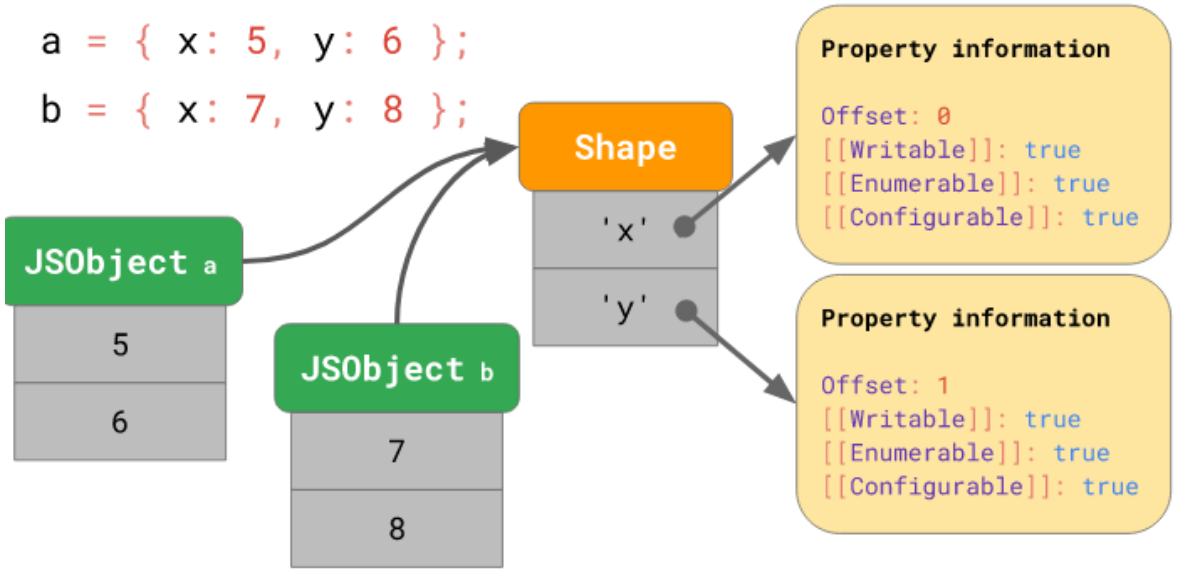
- Ob **dinamičnem instanciranju** objekta z danimi spremenljivkami se v ozadju naredi ti. oblika objekta (**shape**)
- Različna izrazoslovja:
  - Akademski svet: **Hidden classes**
  - JavaScriptCore: **Structures**
  - Chakra: **Types**
  - V8: **Maps**
  - SpiderMonkey: **Shapes**
- Potrebna interna struktura, ki jo uporablja interpreter, da ločuje dinamične objekte
- Objektimi z “enakimi” spremenljivkami lahko imajo interno enako obliko
  - Pomembno za reduciranje pomnilnika
  - Pomembno za predpomnenje in optimizacije



Vir slike: Mathias Bynens (https://mathiasbynens.be)

Oblike objektov

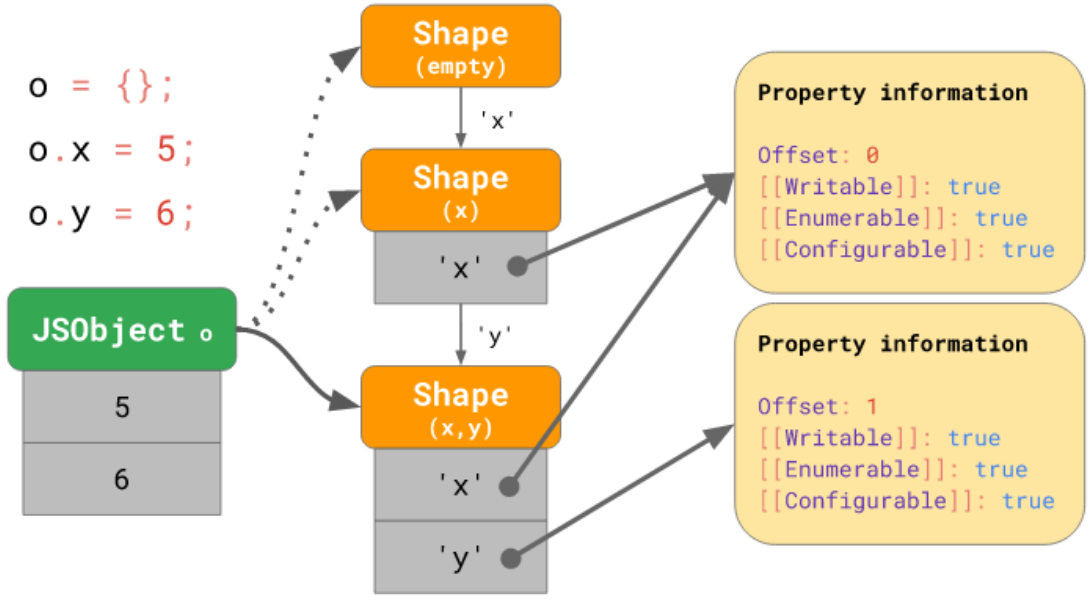
- Objektimi z “enakimi” spremenljivkami lahko imajo **enako interno obliko**
  - **Pomembni je vrstni red** definiranja spremenljivk
- Vsaka spremenljivka objekta ima kazalec na indeks oz. odmik (**offset**) pomnilniške lokacije, kjer je zapisana vrednost spremenljivke
  - Npr. Offset: 0 za ‘x’



Vir slike: Mathias Bynens (https://mathiasbynens.be)

Oblike objektov – instanciranje

- Instanciranje objekta brez spremenljivk zgradi prazno obliko
- Dodajanje spremenljivk pomeni **sprotno grajenje novih oblik**
  - Odmiki vrednosti spremenljivk se ne spremenijo
- Tranzicije oblik objektov (zaradi mutacije objekta) se ohranijo – prednost/slabost ?
- Primer veriženje oblik:



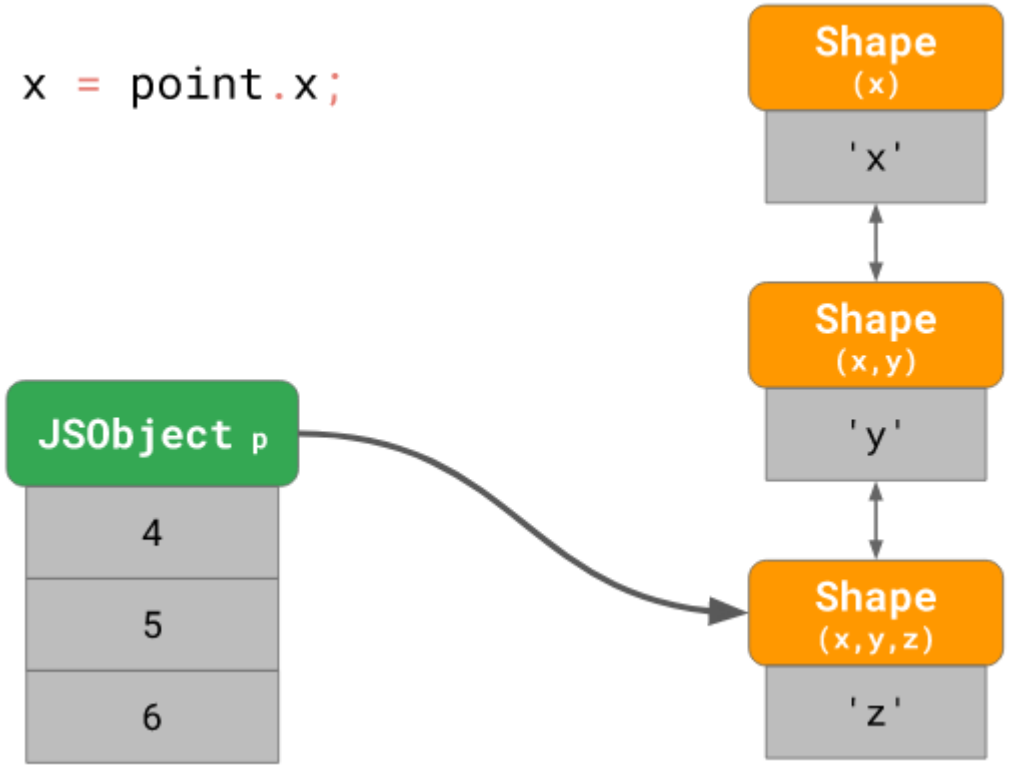
Vir slike: Mathias Bynens (https://mathiasbynens.be)

Oblike objektov – instanciranje

- Tranzicijska veriga je odvisna od zaporedja dodajanja spremenljivk.
- Časovna zahtevnost, če imamo pri objektu n oblik? O(n)

Instanciranje:

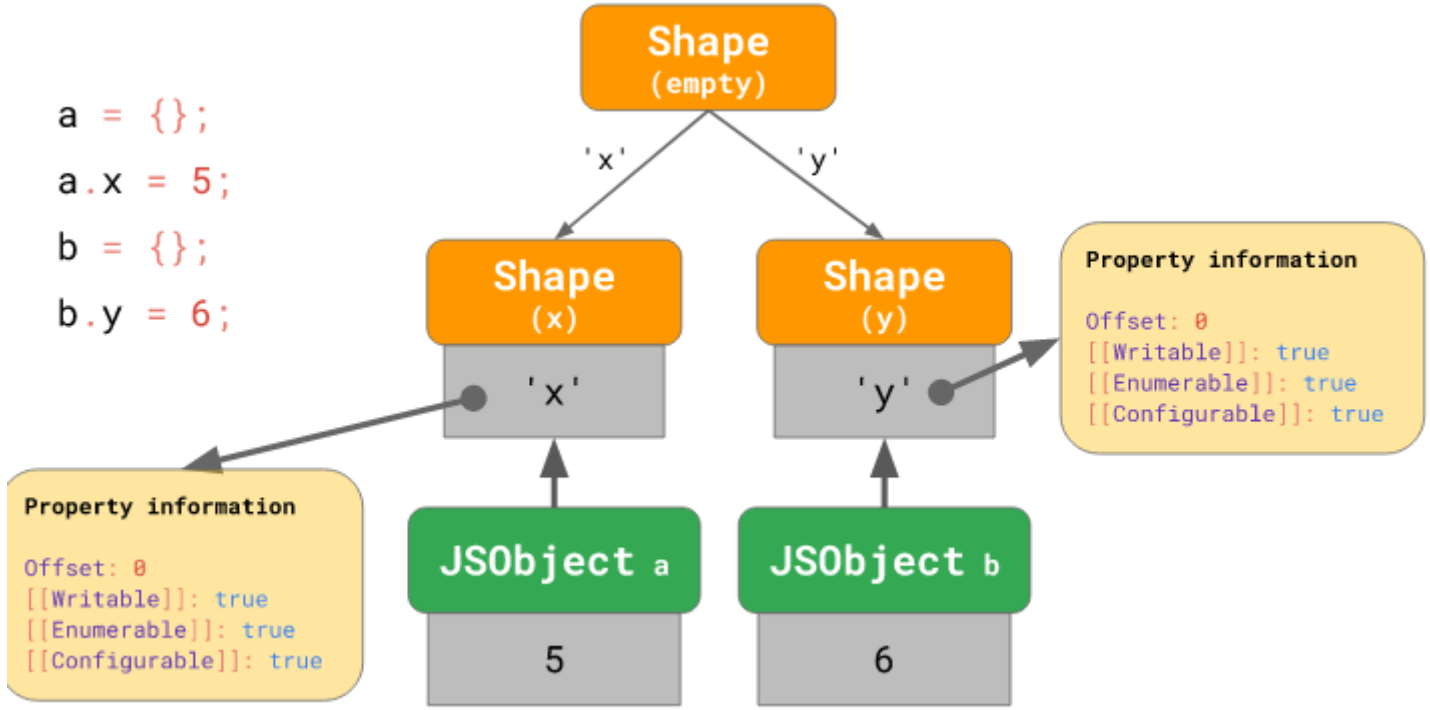
```
const point = {};
point.x = 4;
point.y = 5;
point.z = 6;
```



Vir slike: Mathias Bynens (https://mathiasbynens.be)

Oblike objektov – kolizije ?

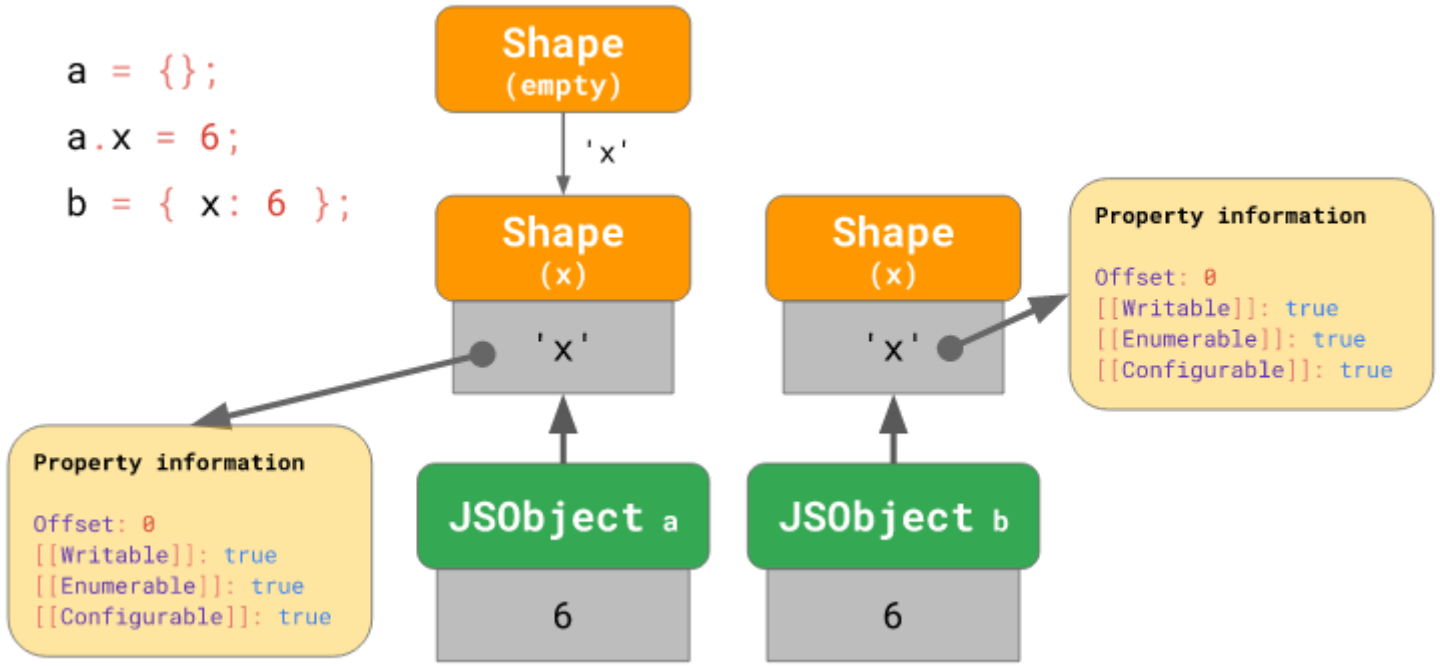
- V primeru da imamo dva enaka objekta se tranzicijska veriga spremeni v **drevo tranzicij**
- Zaradi performančnih razlogov ne želimo preveč vejitev



Vir slike: Mathias Bynens (https://mathiasbynens.be)

Oblike objektov – kolizije ?

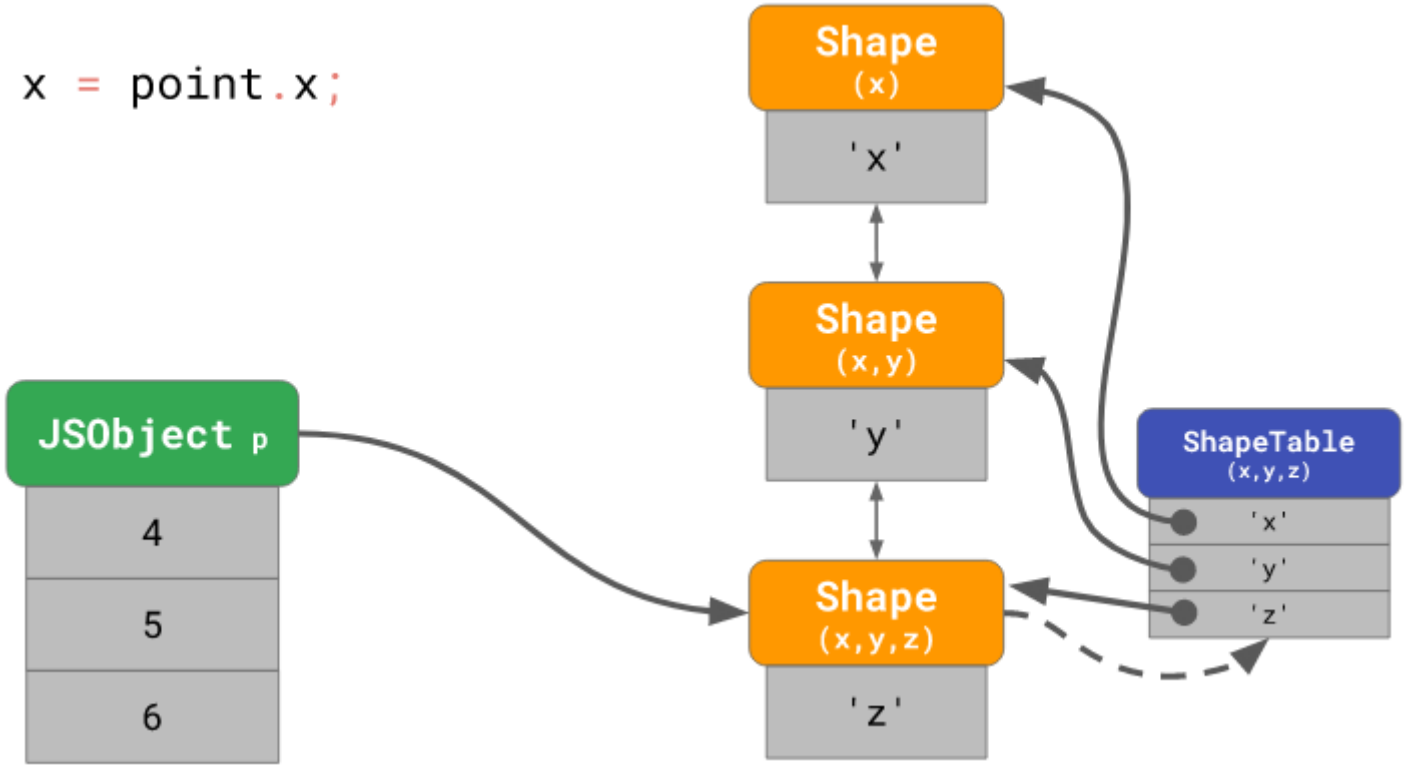
- Ne začnemo vedno v prazni obliki, če **vnaprej definiramo** spremenljivke objekta



Vir slike: Mathias Bynens (https://mathiasbynens.be)

Pohitritev internega dostopa

- Interni dostop do oblik objektov lahko pohitrimo s sekljalno tabelo (**ShapeTable**), ki poveže spremenljivko do oblike objekta, kjer je dana spremeljivka interno definirana
- Časovna zahtevnost ? Blizu O(1)



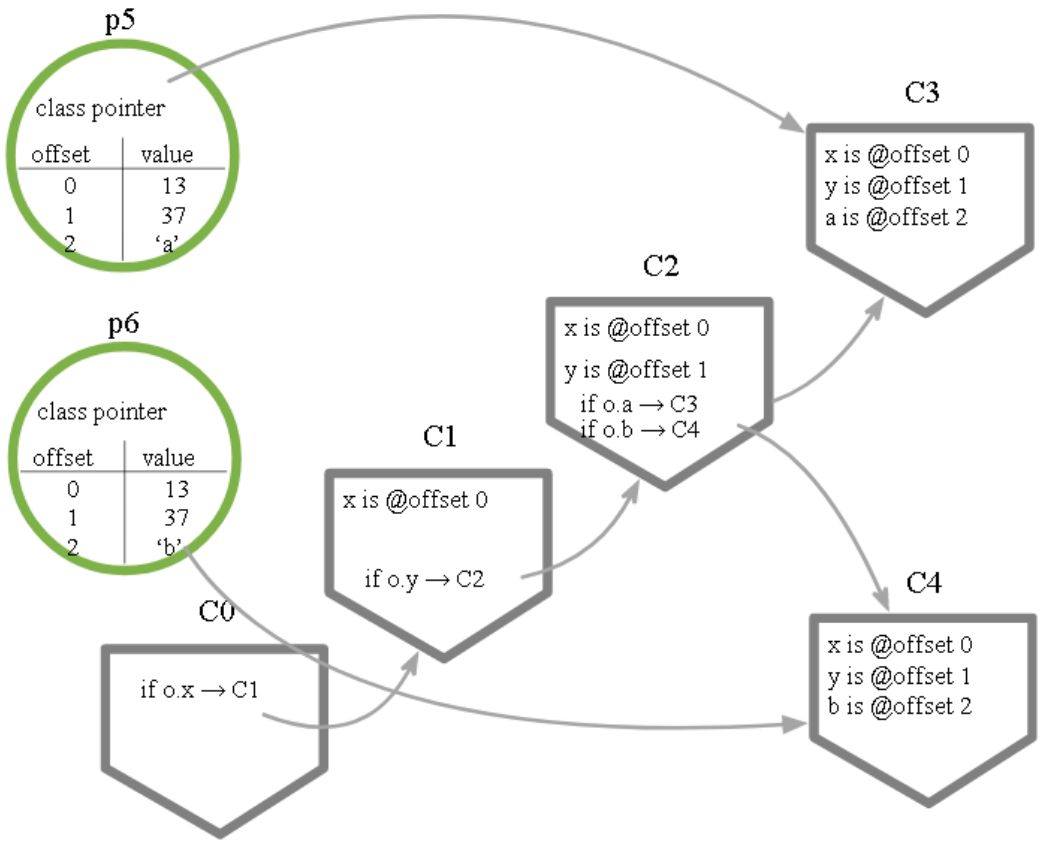
Vir slike: Mathias Bynens (https://mathiasbynens.be)

Oblike objektov predstavljajo DAG

- Direktno usmerjeni graf oz. DAG (Directed Acyclic Graph)
- Primer za dve točki:

```
const p5 = new Point(13, 37);
p5.a = 'a';

const p6 = new Point(13, 37);
p6.b = 'b';
```

- Kaj se zgodi, če zbrišemo p5.a ?
  - C3 ne sme bit enak C2, torej p5 kaže na C2
  - Ne smemo imeti ciklov

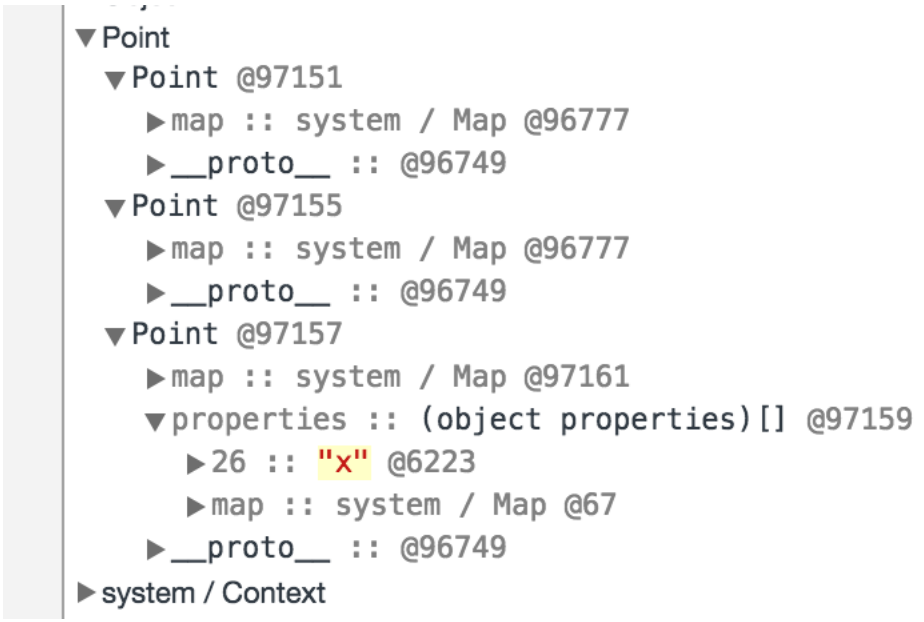


Vir slike: Victor Felder (https://draft.li/)

Brisanje spremenljivk objektov

- Nekateri pogoni **zbrišejo ShapeTable** pri **brisanju ene spremenljivke**, kar upočasni dostopne čase
- Primer JS pomnilnika pri V8 pogonu za:

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}

const p1 = new Point(1, 2);
const p2 = new Point(2, 3);
const p3 = new Point(3, 4);
delete p3.y;
```

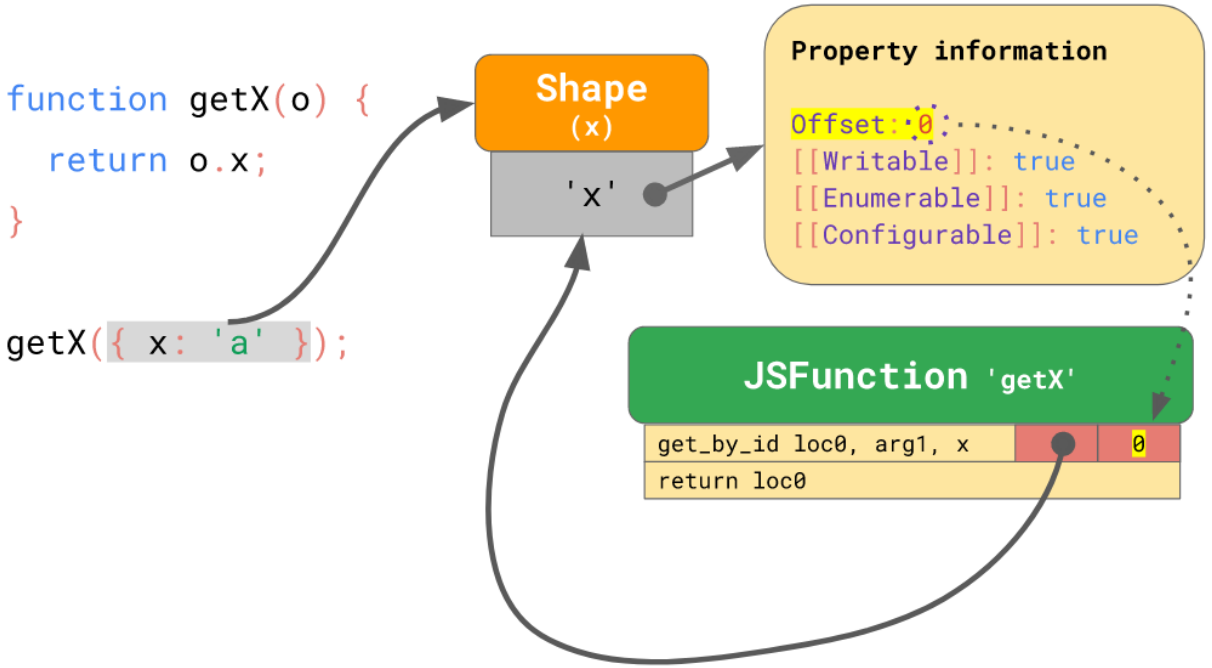


Napredno preiskovanje JS pomnilnika v V8 z Chrome brskalnikom.

- Rešitev? Uporabimo **null** ali **undefined**.
- Testirajte pogon v vašem brskalniku:
  - https://www.measurethat.net/Benchmarks/Show/604/1/delete-vs-set-undefined

Inline caching (IC)

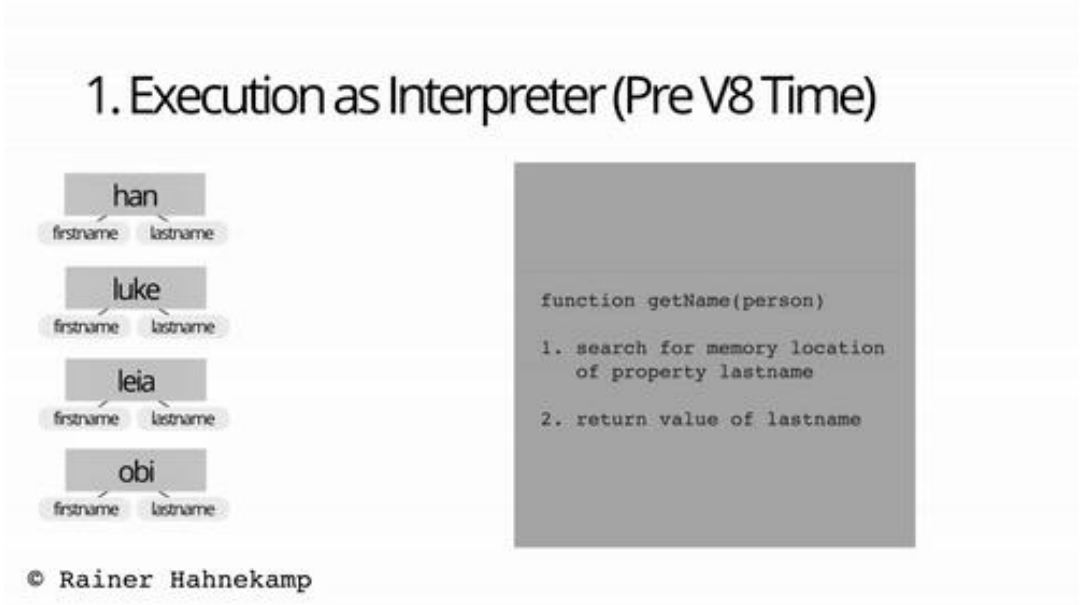
- Interpreter si pri klicu posamezne funkcije **zapomni lokacijo oblike od objekta** in **pomnilniški odmik** vrednosti spremenljivke
- Dani način predpomnenja omogoča hitrejše dostope ter tudi ima povezavo z prevajalnikom (profiliranje “vroče” kode)
- Predpomnenje ni učinkovito, **če zaporedno dostopamo do različnih objektov** (posledično različnih internih oblik)



Vir slike: Mathias Bynens (https://mathiasbynens.be)

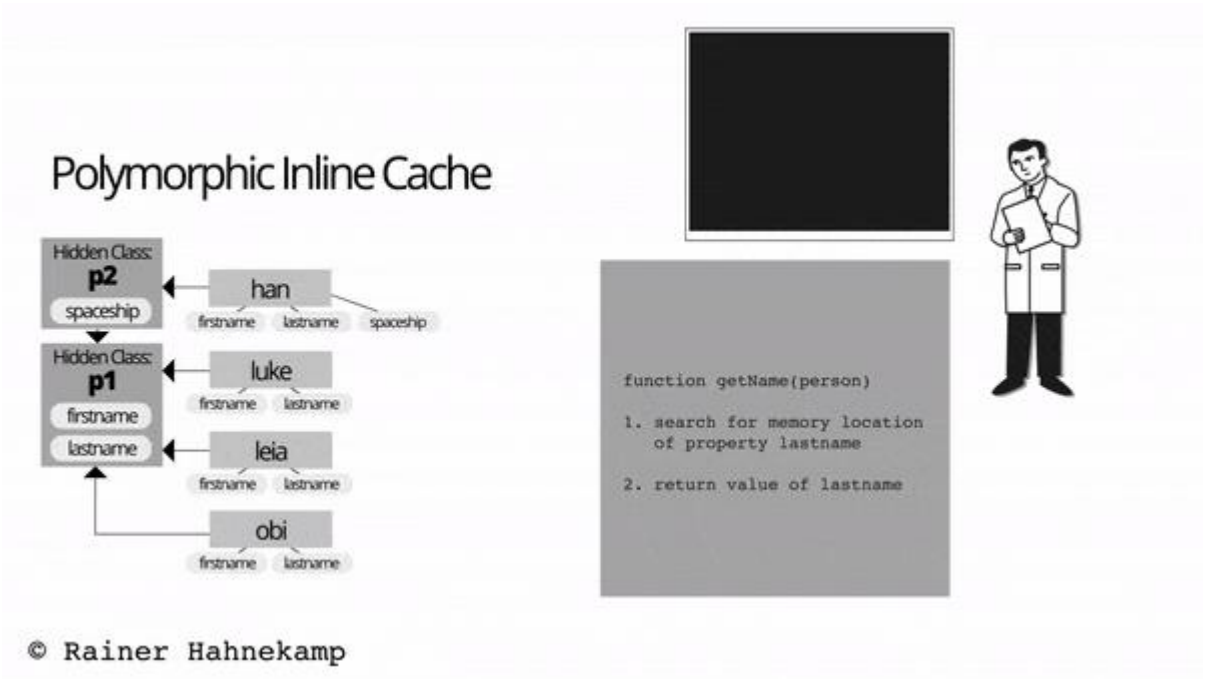
Inline caching (IC)

- **Monomorfni IC** = vsi objekti do katerih v zaporedju dostopamo **imajo enako obliko**.



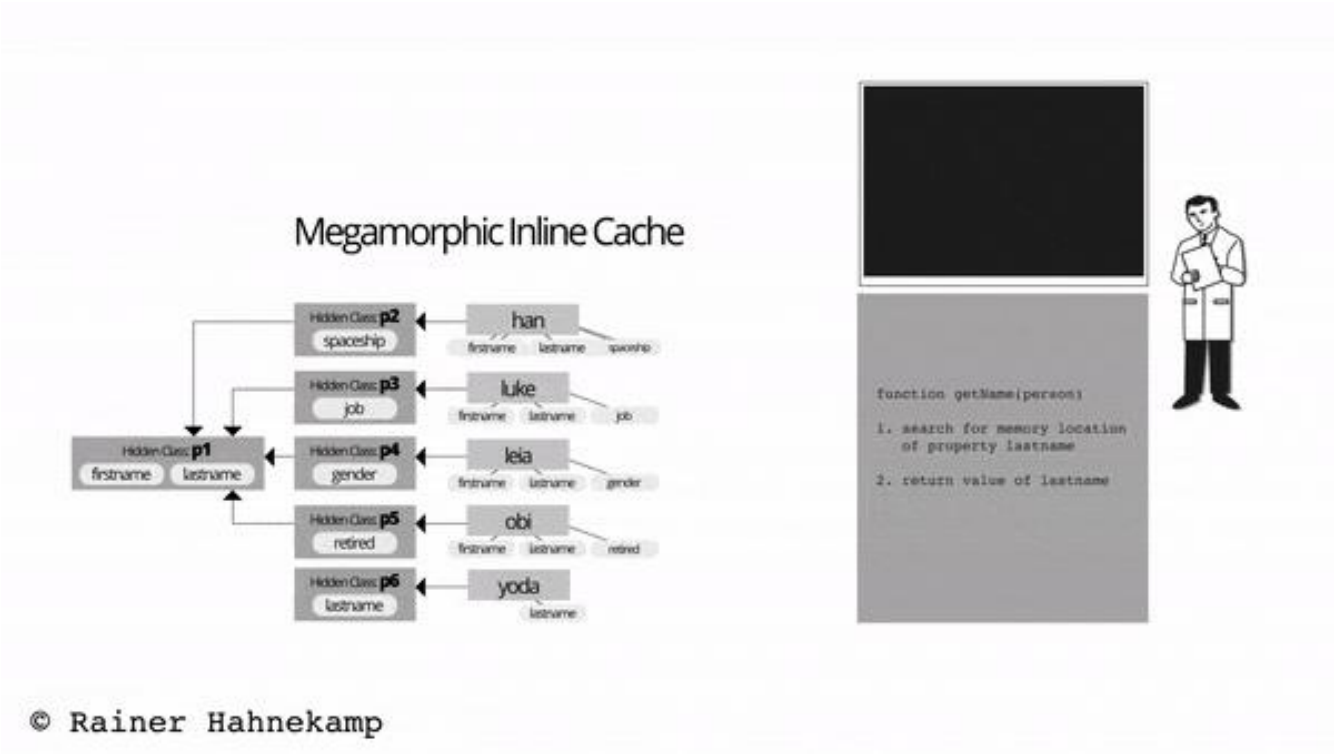
Inline caching (IC)

- **Polimorfni IC** = vsi objekti do katerih v zaporedju dostopamo **imajo različno obliko, št. oblik je 4 ali manj**.



Inline caching (IC)

- **Megamorfni IC** = vsi objekti do katerih v zaporedju dostopamo **imajo različno obliko, št. oblik je več kot 4**.
- Optimizacije kode se izklopijo.



Vrnemo se nazaj na problematični primer

- IC v danem primeru deluje megamorfno
- Zložna koda od *get_a* postane “vroča” in se optimizira v strojno kodo
- Rešitev?

Problematična koda:

```
const X1 = { a: "A", b: "A", c: "A"};
const X2 = { a: "B", b: "B", d: "B"};
const X3 = { a: "C", b: "C", e: "C"};
const X4 = { a: "D", b: "D", f: true};
const X5 = { a: "E"};

const objekti = [ X1, X2, X3, X4, X5, X1, X2, X3];

const get_a = (bla) => bla.a;

for(var i = 0; i < 1000000000; i++)
        get_a(objekti[i & 7]);
```

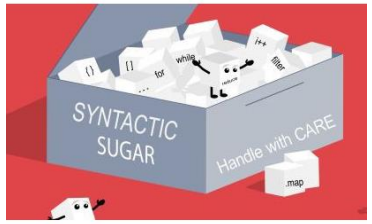
Rešitev problematičnega problema

Neproblematična koda:

```
const X1 = { a: "A", b: "A", c: "A", d: null, e: null, f: null};
const X2 = { a: "B", b: "B", c: null, d: "B", e:null, f:null};
const X3 = { a: "C", b: "C", c:null, d:null, e: "C", f:null};
const X4 = { a: "D", b: "D", c:null, d:null, e:null, f: true};
const X5 = { a: "E", b:null, c:null, d:null, e:null, f:null};

const objekti = [ X1, X2, X3, X4, X5, X1, X2, X3];

const get_a = (bla) => bla.a;

for(var i = 0; i < 1000000000; i++)
        get_a(objekti[i & 7]);
```

Polimorfizem v praksi

- Polimorfne in megamorfne anomalije so precej pogoste pri znanih JS ogrodjih, zaradi **dinamične gradnje DOM elementov**, ki so vezani na dinamične objekte
- Primer:



Vir slike: https://medium.com/front-end-weekly/syntactic-sugar-diabetes-alert-6329a7048cf5

```
class Component {
  render() {
    return "";
  }
}

class HelloComponent extends Component {
  render() {
    return "<div>Hello</div>";
  }
}

class LinkComponent extends Component {
  constructor(text) {
    this.text = text;
  }
  render() {
    return '<a href="'+this.target+“click</a>";
  }
}
```

```
class DOM {
  static renderAll(target, components) {
    let html = "";
    for (const component of components) {
      html += component.render();
    }
    target.innerHTML = html;
  }
}

const components = [
  new HelloComponent(),
  new LinkComponent("http://google.com")
];

DOM.renderAll(document.getElementById("my-app"),
              components);
```

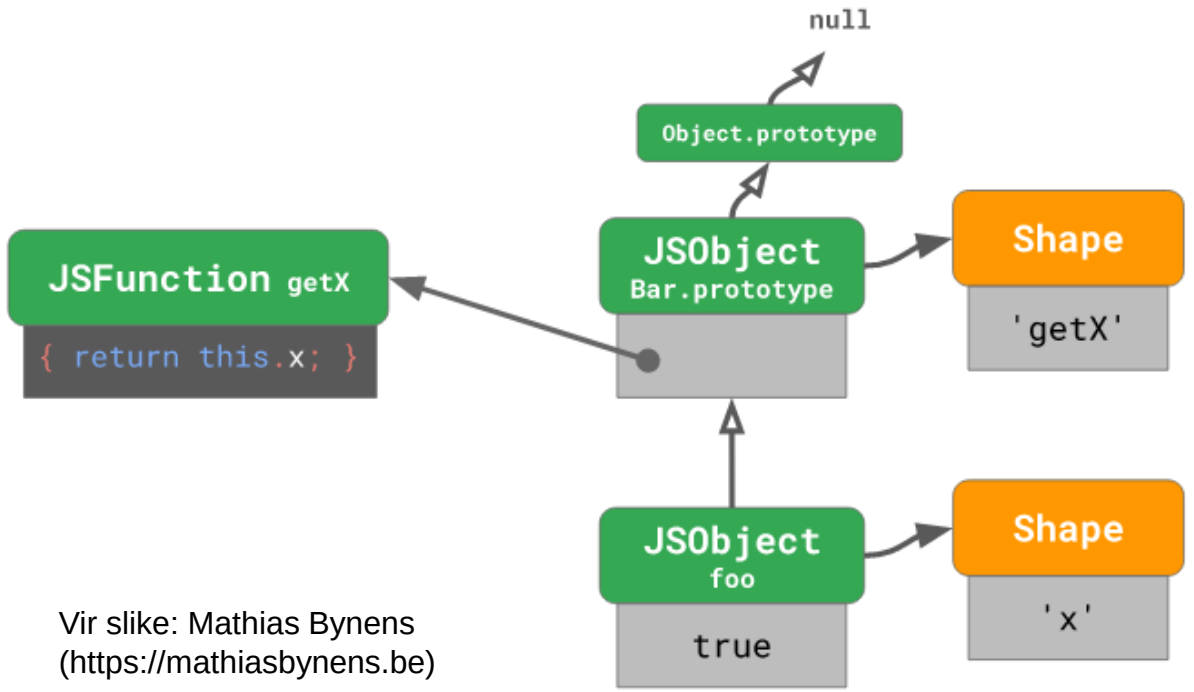
Različne interne oblike objektov

Polimorfna koda (izogibajte se večkratnem klicu)

Prototipi

- V JS lahko uporabimo prototipe objektov preko katerih omogočamo **dedovanje atributov in metod** pri ustvarjanju novih objektov
- Vsak JS objekt ima **verigo dedovanja** do splošnega **Object.prototype** prototipa
- JS pogoni v ozadju uporabljajo **oblike objektov tudi za prototipe objektov**

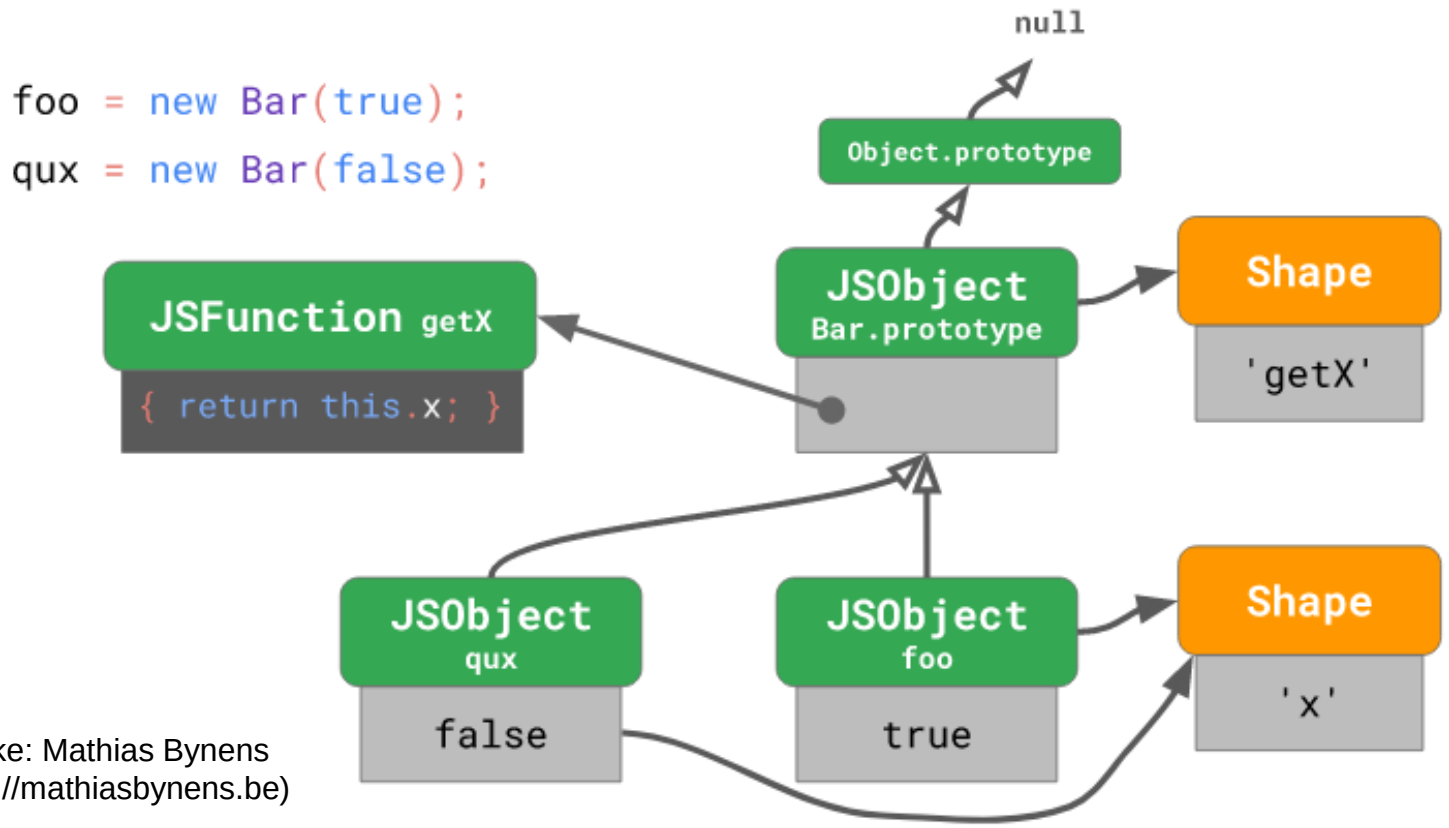
```
function Bar(x) {
        this.x = x;
}

Bar.prototype.getX = function getX() {
        return this.x;
};

foo = new Bar(true)
```



Vir slike: Mathias Bynens (https://mathiasbynens.be)

Prototipi

- Primer dveh instanc prototipa Bar.
- Interne oblike ostanejo enake.

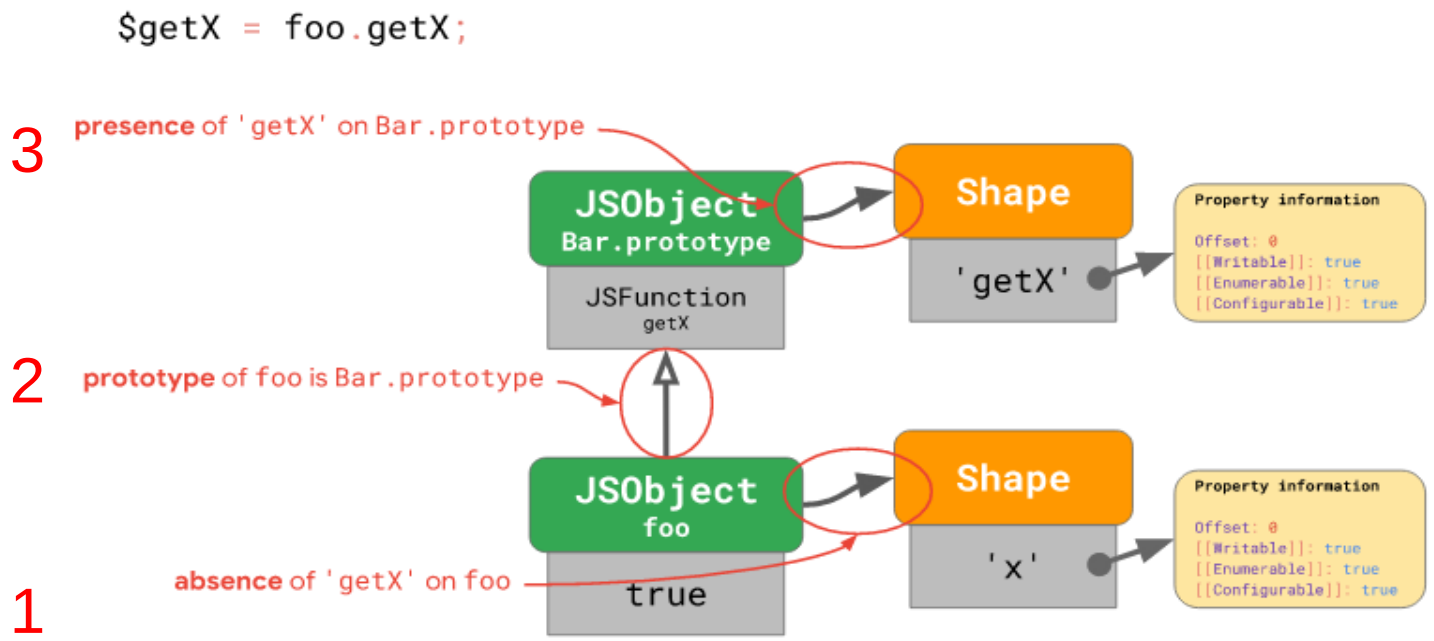


Vir slike: Mathias Bynens (https://mathiasbynens.be)

Prototipi

- Pri klicu funkcije ali dostopa atributa mora JS pogon pogledat v drevo prototipov, v primeru da danega atributa/funkcije ni možno najti v obliki objekta.
  V najslabšem primeru imamo koliko preverjanj? 1+2N (N=št. preiskanih prototipov)

- Za doseganje hitrega dostopa **z IC** ne smemo spreminjat:
  - Dedovan object (**foo**)
  - Reference na prototip foo-ja (**foo.__proto__**)
  - Obliko prototipa (**Bar.prototype**)

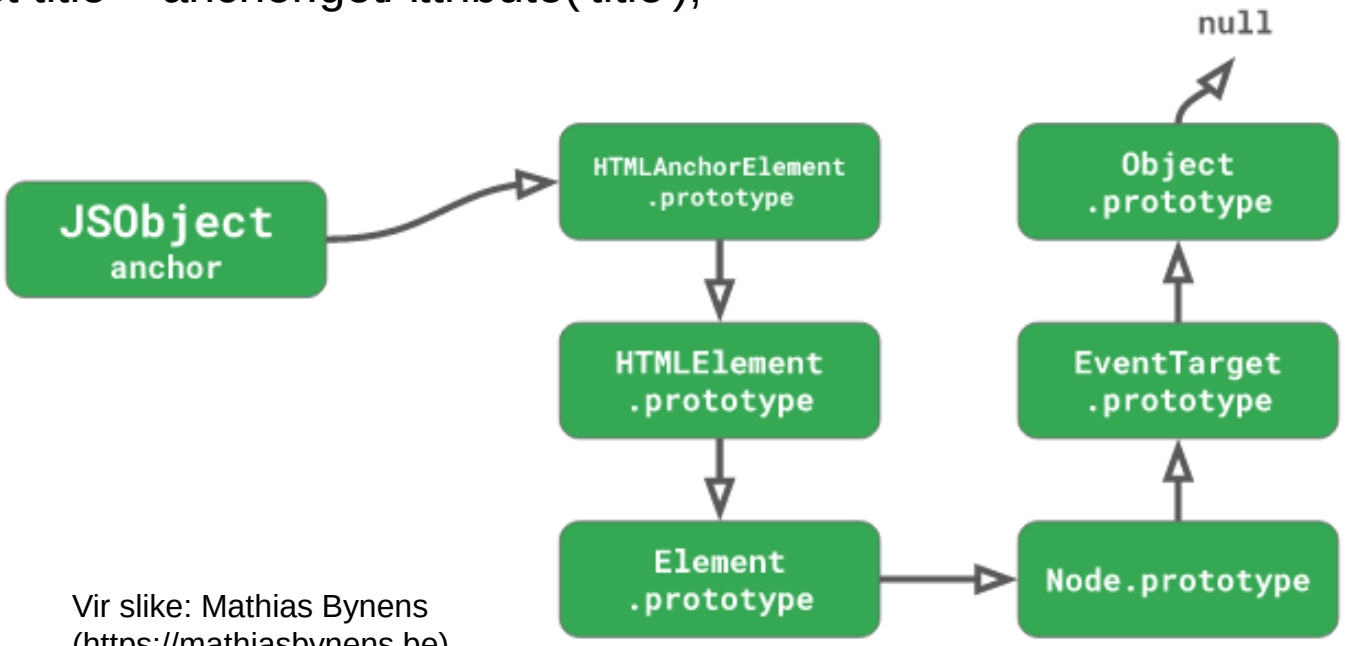


Vir slike: Mathias Bynens (https://mathiasbynens.be)

Prototipi DOM

- Velikokrat je **DOM zapakiran v JS objekte**, kjer se prototipi uporabljajo za dedovanje.
- Primer kjer preiščemo funkcijo getAttribute sproži 7 preverjanj:

```
const anchor = document.createElement('a');

const title = anchor.getAttribute('title');
```



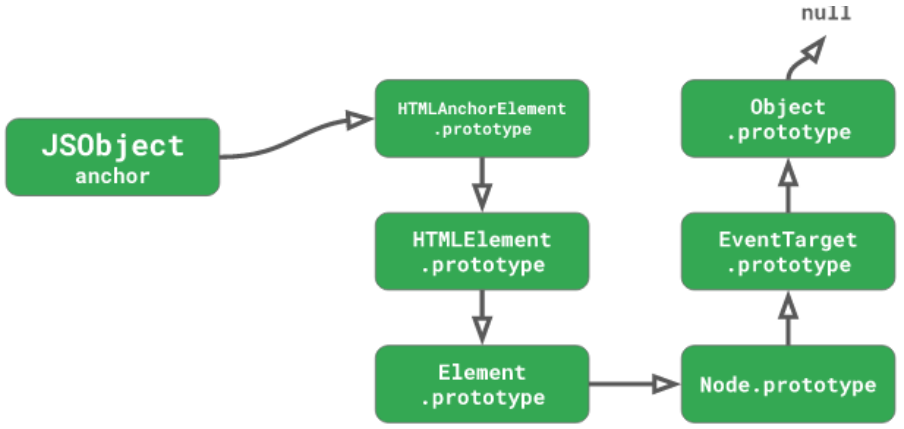
Vir slike: Mathias Bynens (https://mathiasbynens.be)

Prototipi DOM

- Velikokrat je DOM zapakiran v JS objekte, kjer se prototipi uporabljajo za dedovanje.
- Primer kjer preiščemo funkcija getAttribute sproži 7 preverjanj:
  1. Preveri getAttribute na obliki od anchor
  2. Preveri prototip __proto__
  3. Preveri getAttribute na obliki od HTMLAnchorElement.prototype
  4. Preveri prototip __proto__.__proto__
  5. Preveri getAttribute na obliki od HTMLElement.prototype
  6. Preveri prototip __proto__.__proto__.__proto__
  7. Preveri getAttribute na obliki od Element.prototype
  8. Klic:

     anchor.__proto__.__proto__.__proto__.getAttribute(…)

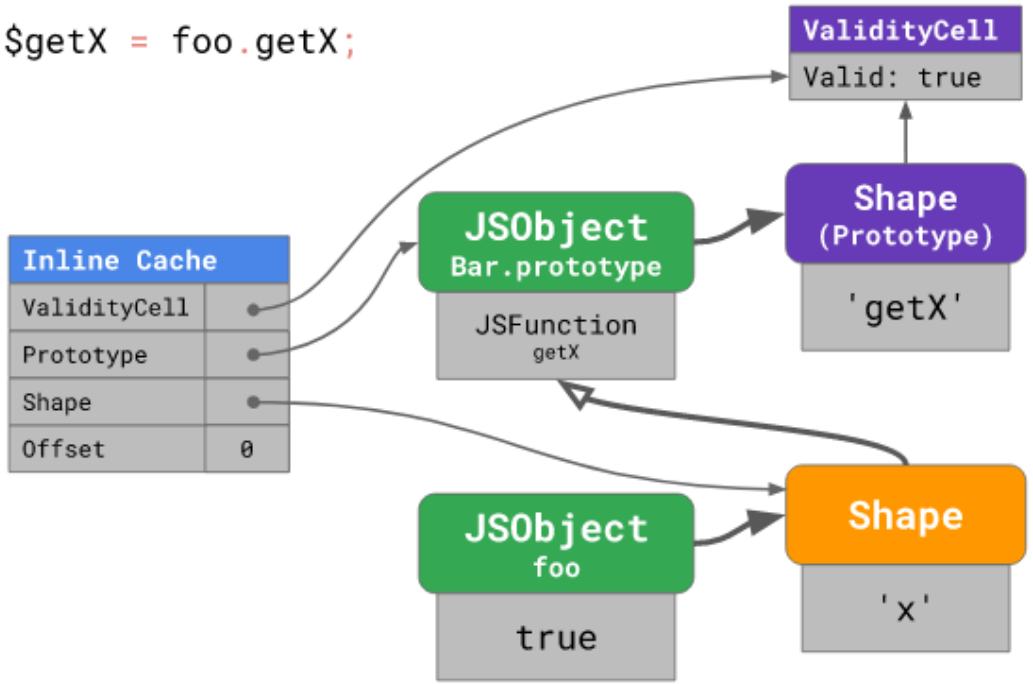
```
const anchor = document.createElement('a');

const title = anchor.getAttribute('title');
```



Vir slike: Mathias Bynens (https://mathiasbynens.be)

Prototipi

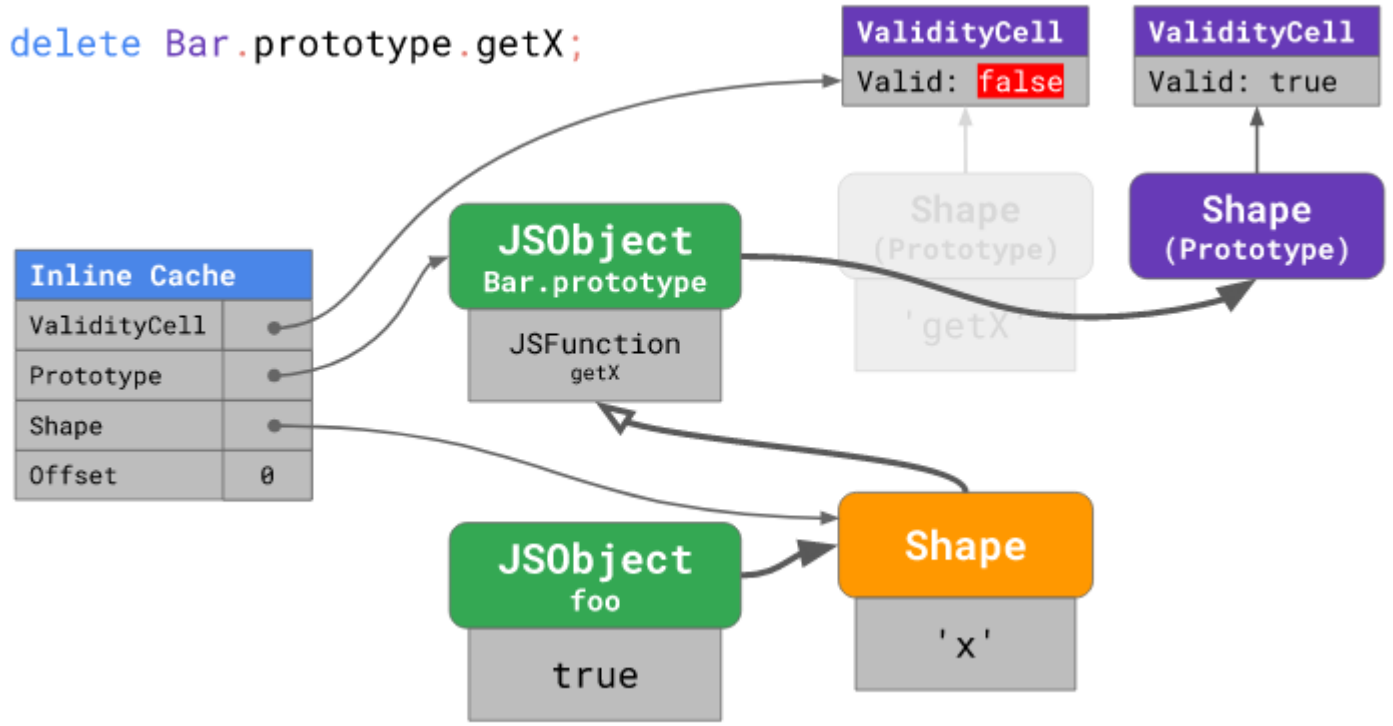
- JS pogoni optimizirajo dostope do **prototipnih atributov in funkcij** z uporabo ti. **validacijsko celico in medpomnilnikom** (inline cache, IC)
- JS pogon ob prvem dostopu prototipne funkcije/atributa hrani **direktno povezavo do oblike prototipa v IC**
- **Validacijska celica** (Boolean) se negira, če pride do spremembe prototipa v verigi prototipov.
- Validacijska celica torej kontrolira ali uporabimo IC ali ne



Vir slike: Mathias Bynens (https://mathiasbynens.be)

Prototipi

- Primer: validacijska celica se negira zaradi spremembe v Bar.prototype
- V tem primeru IC ne bo več uporaben (ti. **cache miss**) in spet iščemo počasi

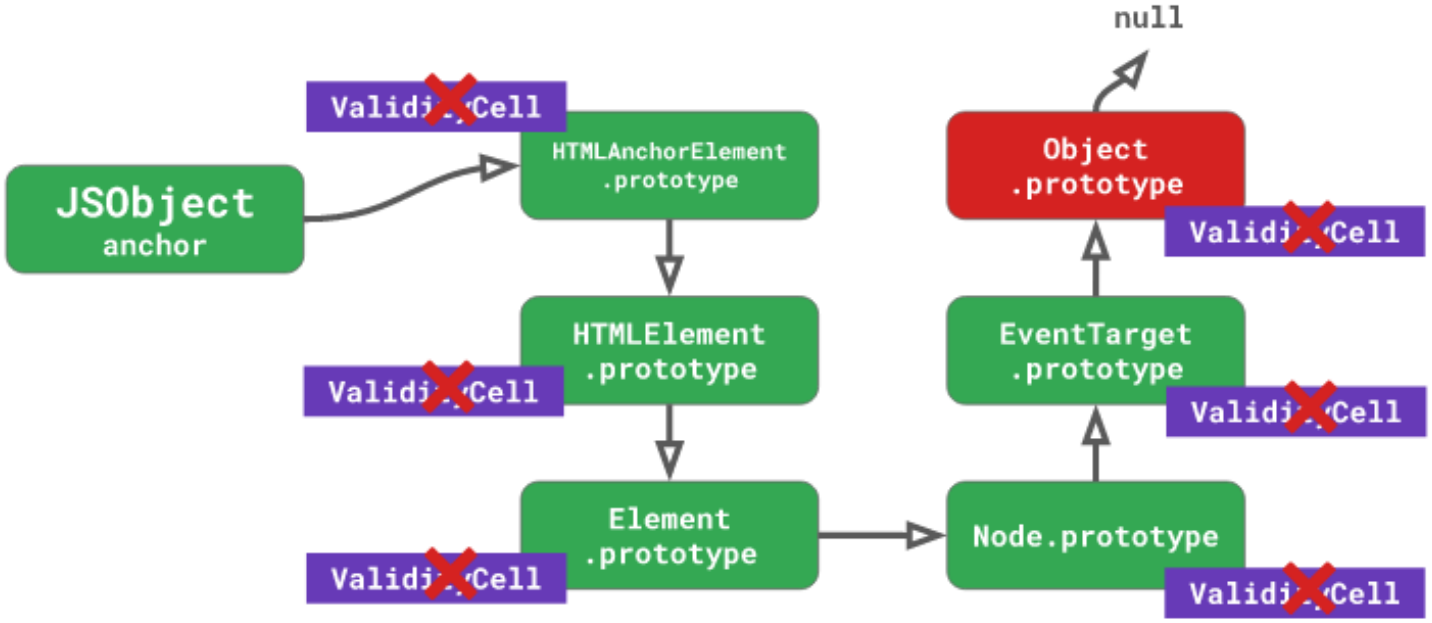


Vir slike: Mathias Bynens (https://mathiasbynens.be)

Prototipi

- Absurdni primer:
  če spremenimo korenski prototip vseh objektov (**Object.prototype**), potem se porušijo vsi predpomnilniki za vse objekte.

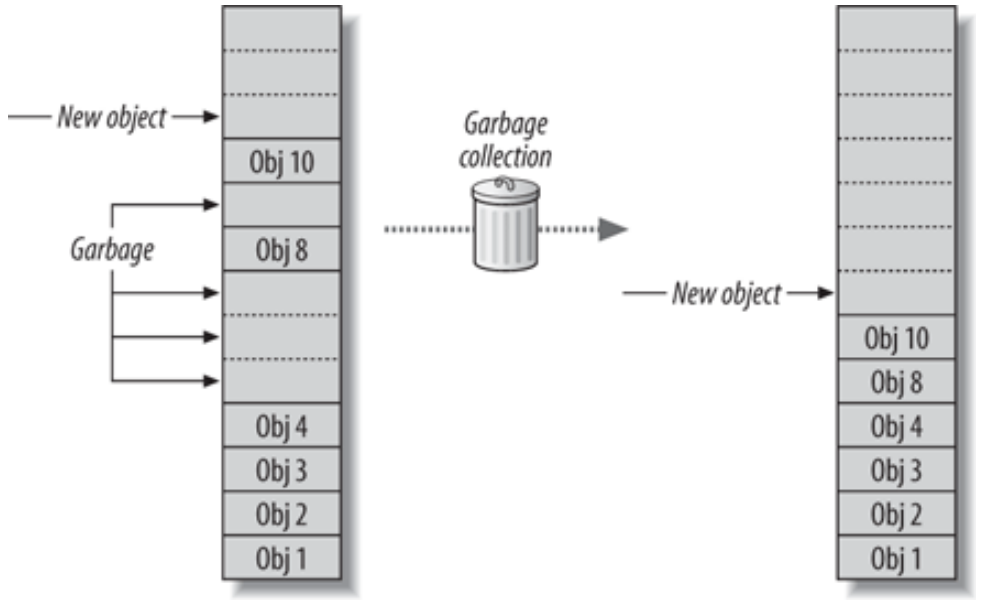
```
Object.prototype.x = 42;
```



Vir slike: Mathias Bynens
(https://mathiasbynens.be)

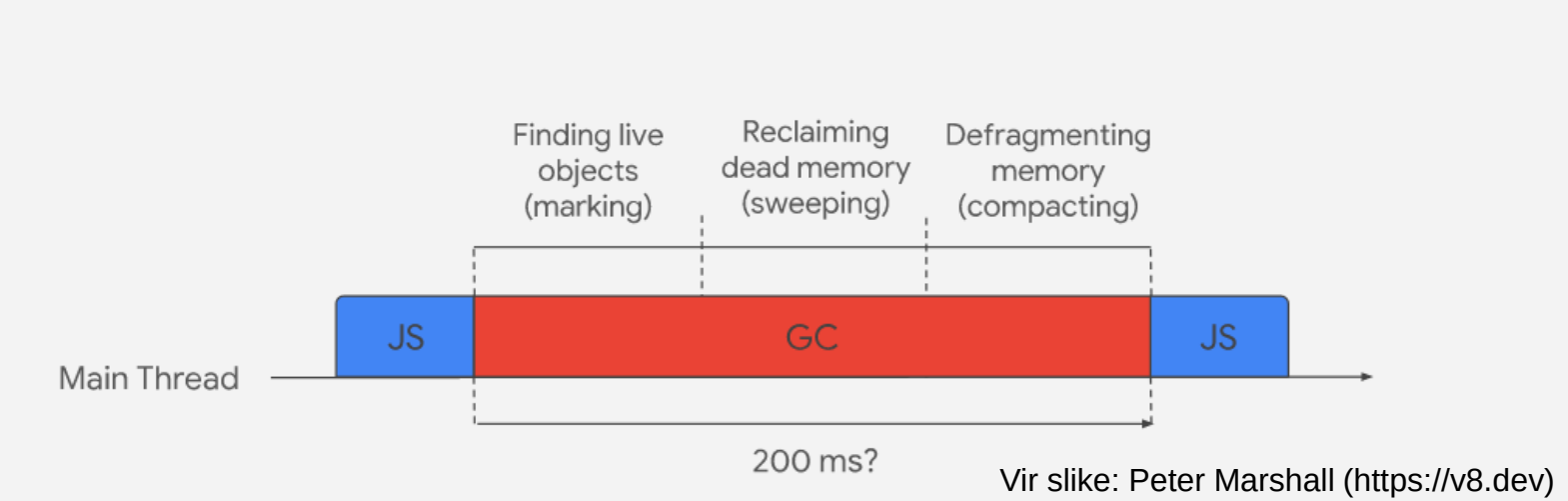
Garbage collection

- Visokonivojski jeziki kot je JavaScript imajo vgrajen mehanizem za **sprotno brisanje alociranega dinamičnega pomnilnika** izven dosega z ti. **Garbage collection (GC)**.
- Do sedaj smo govorili o performančnem vidiku JS, pri čemer pa ne smemo pozabit na **optimizacijo pomnilniškega prostora**.



Garbage collection

- GC algoritem običajno deluje po naslednjih korakih:
  - Označevanje (**marking**): pregledamo vse objekte in njihovo dosegljivost.
  - Pometanje (**sweeping**): sprostimo (označimo) pomnilnik vseh “mrtvih” objektov.
    - Običajno se za **proste sekcije pomnilnika vodi seznam**, za lažji namen ponovne uporabe.
  - Defragmentacija (**compact/defragment**): proste dele pomnilnika **grupiramo**, kot tudi polne.

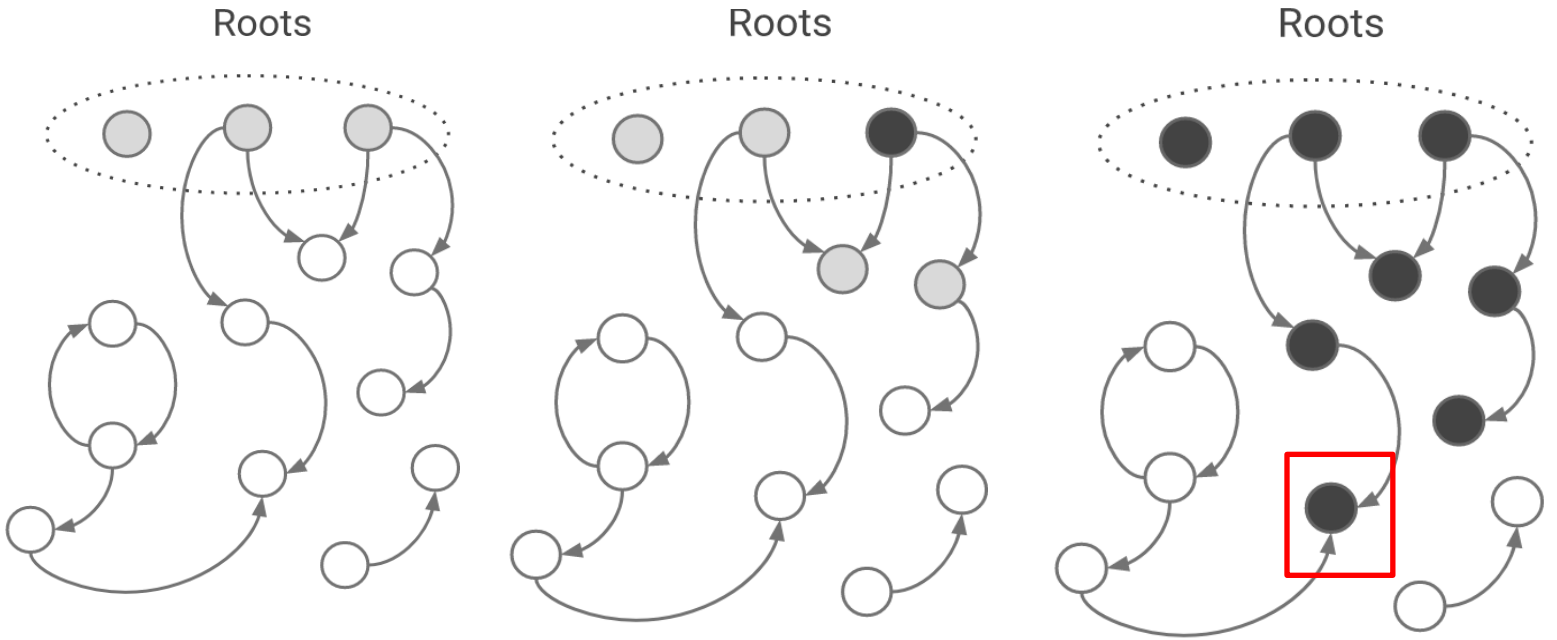


Vir slike: Peter Marshall (https://v8.dev)

- GC ustavi glavno nit JS pogona, zakaj? Potrebno preuredit reference do objektov.

Garbage collection

- Faza označevanja deluje na naslednji način:
  1. Označimo vse aktivne in globalne objekte, ki predstavljajo ti. korenske objekte v verigi referenciranja (angl. roots)
  2. Za vsak korenski objekt preiščemo vse referencirane objekte, ki so še "živi".
     Uporabimo algoritme za preiskovanje grafov (npr. BFS, DFS).
  3. Na koncu ostanejo objekti, ki jih lahko označimo za "**mrtve**".



Korenski objekti

Dostopanje referenc

Črni = "živi", Beli = "mrtvi" objekti

Vir slik: Peter Marshall (https://v8.dev)

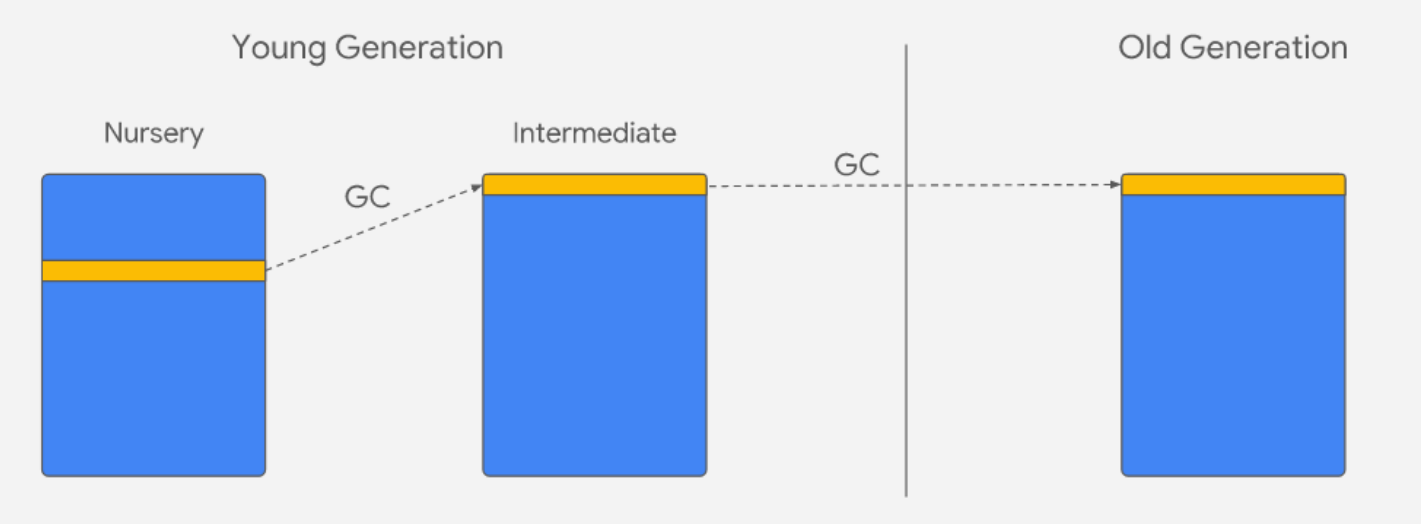
Garbage collection

- Slabost ustavljanja glavne niti JS pogona je očitna, precej v spletnih brskalnikih kjer **pade uporabniška interaktivnost**.
- Podrobneje si bomo pogledali delovanje **GC Orinoco** od **Google V8** pogona, ki razdeli dinamični pomnilnik na dva glavna dela (ti. mlada in stara generacija)

V8 Orinoco

Garbage collection (V8 Orinoco)

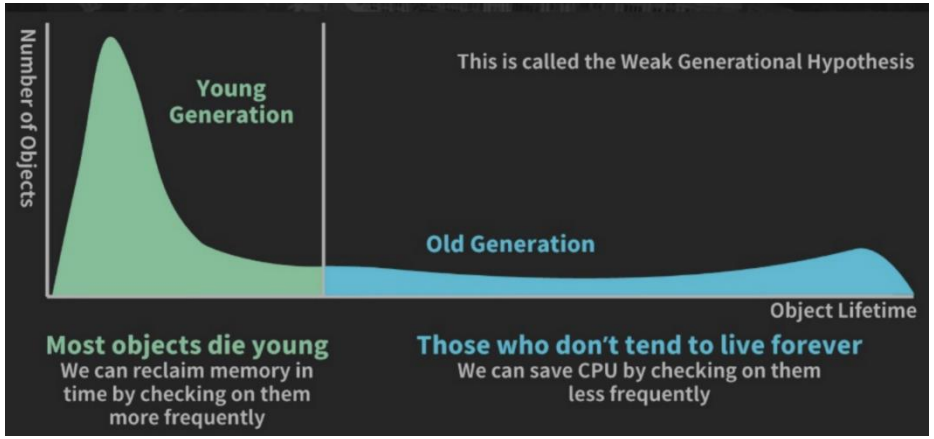
- **Mlada generacija** predstavlja večino JS objektov in je razdeljena na dva dela: **inkubator in vmesni pomnilnik.**
- Če JS objekt preživi GC v inkubatorju, se ta prestavi v vmesni pomnilnik, ter če tam preživi GC se premakne v prostor **stare generacije** (npr. **globalni objekti kot window**).



Vir slike: Peter Marshall (https://v8.dev)

Garbage collection (V8 Orinoco)

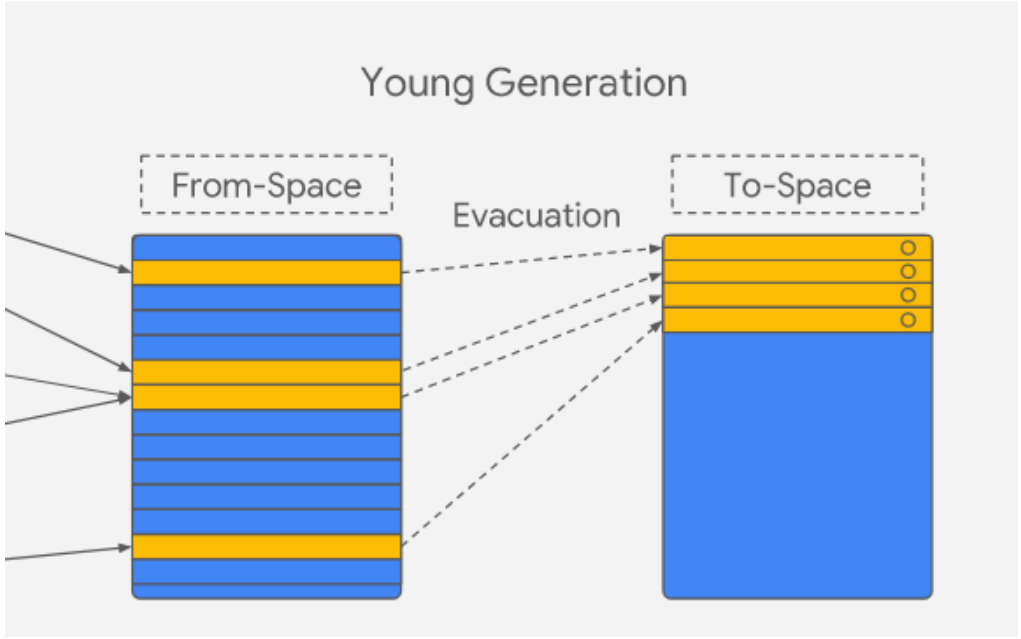
- V V8 se v osnovi uporablja **dva ločena GC algoritma** (ti. **manjši & veliki GC** - angl. minor & major).
- **Manjši GC** operira večkrat nad pomnilnikom **mlade generacije**, **veliki GC** pa nad pomnilnikom **stare generacije**.
- V splošnem velja, da večina objektov "**umre mladih**" oz. **hitro izgubijo dosegljivost**, zato je bolj smiselno premaknit objekte, ki preživijo, zakaj?
  - Ti. Hipoteza slabe generacije



Vir slike: Nikolay Veretelnyk (V8)

Garbage collection (V8 Orinoco)

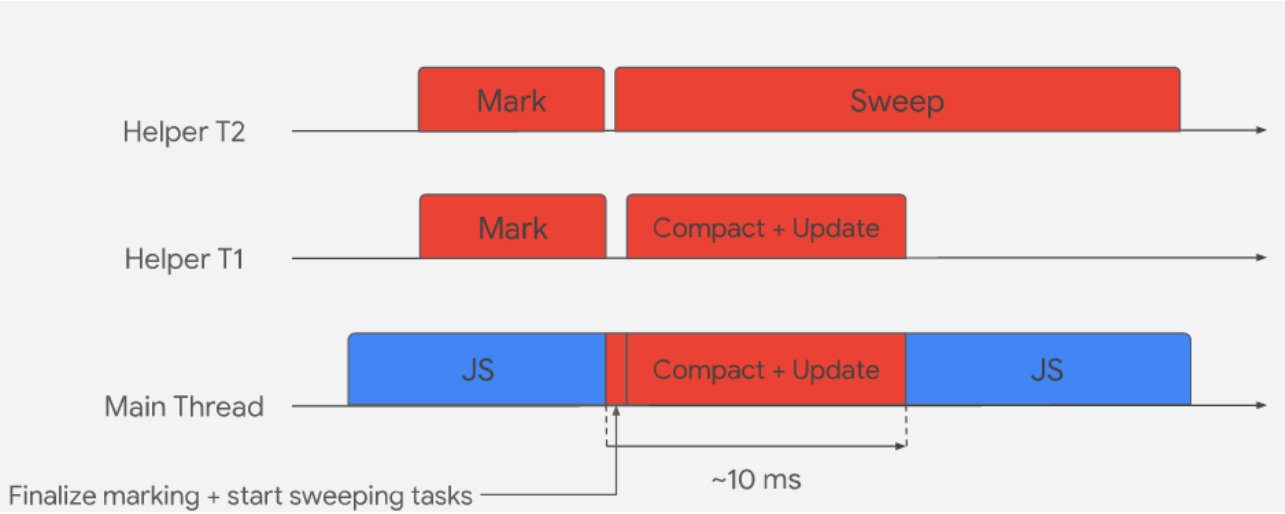
- V splošnem velja, da večina objektov “umre mladih” oz. **hitro izgubijo dosegljivost**, zato je bolj smiselno premaknit objekte, ki preživijo, zakaj?
  - Ostane nam torej prostor mrtvih objektov.
- Primer manjšega GC:

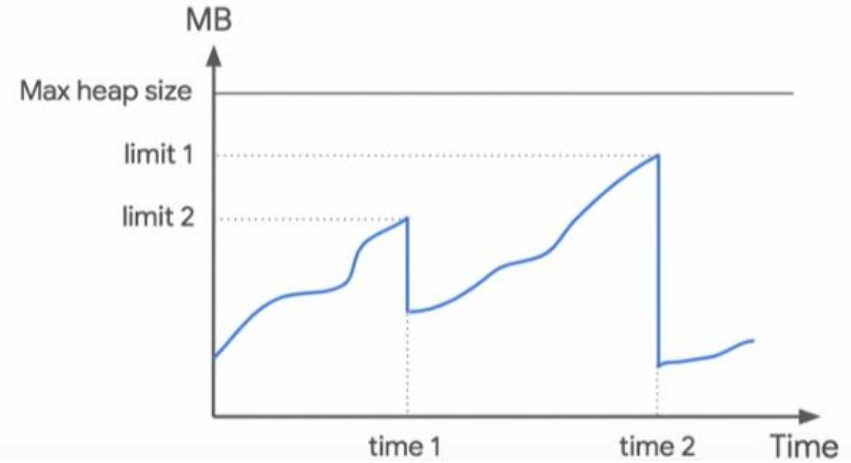


Vir slike: Peter Marshall (https://v8.dev)

Garbage collection (V8 Orinoco)

- Pri V8 se večji GC izvaja najprej v ločenih nitih, kjer izvaja operacijo beleženja objektov v prostoru objektov stare generacije.
- Nato se pometanje in defragmentacija pomnilniškega prostora izvede skupaj z glavno nitjo pogona.
- Večji GC se sproži, kadar poraba pomnilnika doseže določeno dinamično mejo.

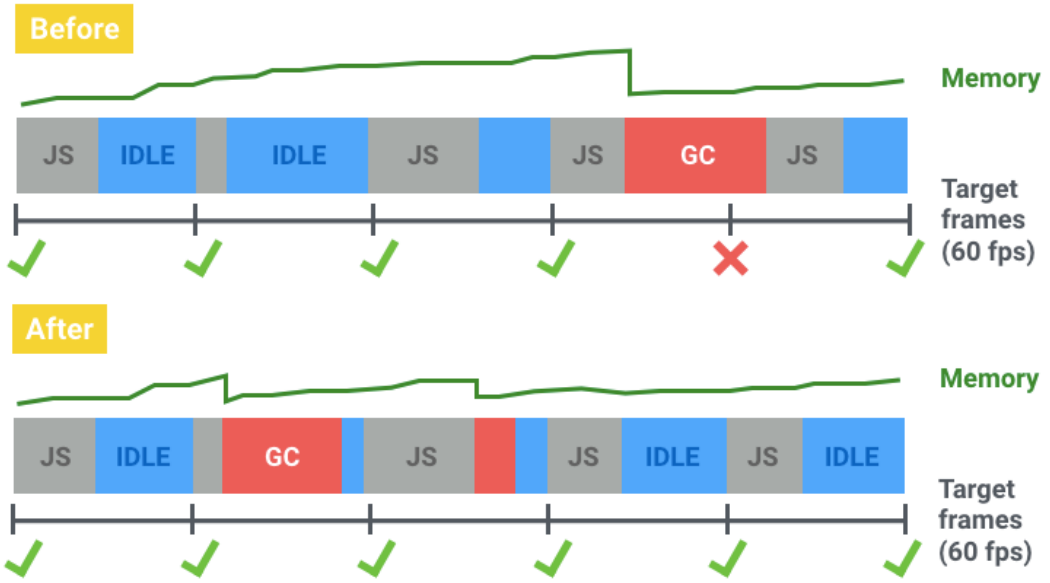




Vir slik: Peter Marshall (https://v8.dev)

Garbage collection (V8 Orinoco)

- Kdaj je optimalno izvest GC?
  - Pri V8 spremljajo zasedenost glavne niti (poraba CPE itd) in kadar se zazna nedejavnost (**idle time**) se sproži GC.
- V brskalniku je to običajno ob gledanju videa, branja daljšega besedila itd.



Vir slike: Seth Thompson (https://blog.chromium.org)

Uhajanje pomnilnika

- Sedaj, ko poznamo delovanje GC je naloga programerja čimmanj časa zadrževat nepotrebne objekte v pomnilniku.
- Ena izmed večjih napak je alokacija globalnih objektov, ki so dosegljivi čez celotno JS programsko kodo. Npr. nesrečna definicija globalnih spremenljivk:

```
function f() {
    i = 1;
}
f();

console.log("I see you i: " + i);
```

- V brskalnikih bo objekt **i** avtomatsko pripadal objektu **window**.
- Rešitev ? Uporabimo let ali var (razlika?). Kaj pa v primeru this?

Uhajanje pomnilnika

- Do uhajanja pomnilniškega prostora, ki ga GC nikakor ne more pobrisat, pride velikokrat zaradi referenciranja zunanjih objektov **v povratnih (callback) asinhronih funkcijah**.
- Primer, če dostopamo do nedosegljivega elementa Node:

```
someResource = getVeryLargeData();
setInterval(function() {
    node = document.getElementById('Node');
    if(node) {
        node.innerHTML = JSON.stringify(someResource));
    }
}, 1000);
```

  - GC ne more pobrisat someResource dokler se funkcija ne zaključi.

Dodatni viri

- Profiliranje V8 pomnilnika in GC v Chrome DevTools:
  - https://developer.chrome.com/devtools/docs/demos/memory
- V8 blog in podrobni opis delovanja Orinoco GC:
  - https://v8.dev/blog/orinoco-parallel-scavenger
  - https://v8.dev/blog/orinoco
  - https://v8.dev/blog/jank-busters

Vir: XKCD