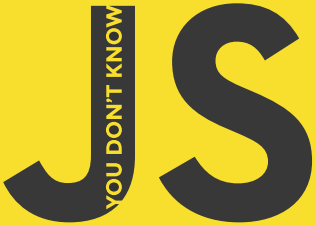
O'REILLY®

YOU DON'T KNOW JS



Itu ***ANDA TIDAK TAHU JS*** seri meliputi:

- *Up & Pergi*
- *Lingkup & Penutup*
- *ini & Prototip Obyek*
- *Jenis & Grammar*
- *Async & Kinerja*
- *ES6 & Beyond*

UP & GOING

Sangat mudah untuk belajar bagian dari JavaScript, namun jauh lebih sulit untuk belajar sepenuhnya-atau bahkan *cukup* -apakah Anda baru ke bahasa atau telah digunakan selama bertahun-tahun. Dengan "Anda Tidak Tahu JS" seri buku, Anda akan mendapatkan pemahaman yang lebih lengkap dari JavaScript, termasuk bagian rumit dari bahasa yang banyak mengalami programmer JavaScript hanya menghindari. buku pertama seri, ***Up & Going,*** memberikan latar belakang yang diperlukan bagi anda dengan pengalaman pemrograman terbatas. Dengan belajar blok bangunan dasar pemrograman, serta mekanisme inti JavaScript, Anda akan siap untuk menyelam ke yang lain, lebih mendalam buku dalam seri-dan berada di jalan Anda menuju benar JavaScript.

Dengan buku ini Anda akan:

- Pelajari blok bangunan pemrograman penting, termasuk operator, jenis, variabel, conditional, loop, dan fungsi
- Menjadi akrab dengan mekanisme inti JavaScript, seperti nilai-nilai, penutupan fungsi, *ini,* dan prototipe
- Mendapatkan gambaran dari buku-buku lain dalam seri-dan mempelajari mengapa hal itu penting untuk memahami *semua* bagian dari JavaScript

Kyle Simpson adalah Open Web Evangelist dari Austin, TX, yang bergairah tentang segala sesuatu JavaScript. Dia seorang penulis, workshop pelatih, pembicara teknologi, dan OSS kontributor / pemimpin.

JAVASCRIPT
JAVASCRIPT

US $ 4,99 CAN $ 5,99
ISBN: 978-1-491-92446-4

Twitter: @oreillymedia
facebook.com/oreilly

O'REILLY®
oreilly.com
YouDontKnowJS.com

Up & Pergi

Kyle Simpson

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Up & Pergi

oleh Kyle Simpson

Diterbitkan oleh O'Reilly Media, Inc., 1005 Gravenstein Jalan Raya Utara, Sebastopol, CA 95472.

buku O'Reilly dapat dibeli untuk pendidikan, bisnis, atau penggunaan promosi penjualan. edisi online juga tersedia untuk sebagian besar judul (*http://safaribooksonline.com*). Untuk lebih informasi, hubungi perusahaan kami / kelembagaan Departemen penjualan: 800-998-9938 atau *corporate@oreilly.com.*

editor: Simon St.laurent dan Brian MacDonald

Produksi Editor: kristen Brown

Pemeriksa naskah: Jasmine Kwityn

Korektor: amanda Kersey

Interior desainer: David Futato

Penutup Designer: karen Montgomery

Illustrator: Rebecca Demarest

April 2015: Edisi pertama

Sejarah revisi untuk Edisi Pertama

2015/03/17: Rilis Pertama

Lihat *http://oreilly.com/catalog/errata.csp?isbn=9781491924464* untuk rincian rilis.

978-1-491-92446-4 [LSI]

Daftar Isi

Kata pengantar

Apa hal baru terakhir yang Anda pelajari?

Mungkin itu bahasa asing, seperti Italia atau Jerman. Atau mungkin itu adalah editor grafis, seperti Photoshop. Atau teknik memasak atau woodworking atau latihan rutin. Aku ingin kau ingat perasaan bahwa ketika Anda akhirnya mendapatkannya: saat bola lampu. Ketika hal-hal pergi dari kabur ke jelas, karena Anda menguasai meja gergaji atau memahami perbedaan antara kata benda maskulin dan feminin dalam bahasa Prancis. Bagaimana rasanya? Cukup menakjubkan, bukan?

Sekarang saya ingin Anda untuk melakukan perjalanan kembali sedikit lebih lanjut dalam memori Anda ke kanan sebelum Anda belajar keterampilan baru Anda. bagaimana *bahwa* merasa? Mungkin sedikit menakutkan dan mungkin sedikit frustrasi, kan? Pada satu titik, kita semua tidak tahu hal-hal yang kita tahu sekarang, dan itu benar-benar OK; kita semua mulai di suatu tempat. Belajar materi baru adalah petualangan yang menyenangkan, terutama jika Anda mencari untuk belajar subjek efisien.

Saya mengajar banyak kelas coding pemula. Para siswa yang mengikuti kelas saya sering mencoba mengajar diri mata pelajaran seperti HTML atau JavaScript dengan membaca posting blog atau menyalin dan menyisipkan kode, tetapi mereka belum mampu untuk benar-benar menguasai materi yang akan memungkinkan mereka untuk kode hasil yang diinginkan. Dan karena mereka tidak benar-benar memahami seluk-beluk topik coding tertentu, mereka tidak bisa menulis kode erful pow- atau debug pekerjaan mereka sendiri karena mereka tidak benar-benar berdiri di bawah- apa yang terjadi.

Saya selalu percaya dalam mengajar kelas saya cara yang tepat, berarti saya mengajar standar web, markup semantik, kode baik berkomentar, dan praktik terbaik lainnya. Aku menutupi subjek dalam cara yang menyeluruh untuk menjelaskan bagaimana dan mengapa, tanpa hanya melempar kode untuk menyalin

dan paste. Ketika Anda berusaha untuk memahami kode Anda, Anda membuat pekerjaan lebih baik dan menjadi lebih baik apa yang Anda lakukan. kode ini tidak hanya Anda *pekerjaan* lagi, itu Anda *kerajinan.* Ini adalah mengapa saya mencintai *Up & Pergi.* Kyle membawa kita pada menyelam jauh melalui sintaks dan terminologi untuk memberikan pengantar untuk JavaScript tanpa memotong sudut. Buku ini tidak meluncur di atas permukaan tapi benar-benar memungkinkan kita untuk benar-benar memahami konsep-konsep.

Karena itu tidak cukup untuk dapat menduplikasi potongan jQuery ke situs web Anda, dengan cara yang sama itu tidak cukup untuk belajar bagaimana untuk membuka, menutup, dan menyimpan dokumen Photoshop. Tentu, setelah saya belajar beberapa dasar-dasar tentang program ini, saya bisa membuat dan berbagi desain yang saya buat. Tapi tanpa sah mengetahui alat dan apa yang di belakang mereka, bagaimana saya bisa menentukan grid, atau kerajinan sistem jenis dibaca, atau grafis mize opti- untuk digunakan web. Hal yang sama berlaku untuk JavaScript. Tanpa mengetahui bagaimana loop bekerja, atau bagaimana untuk mendefinisikan variabel, atau apa ruang lingkup adalah, kita tidak akan menulis kode terbaik yang kita bisa. Kami tidak ingin puas dengan apa pun kurang ini, setelah semua, kerajinan kami.

Semakin banyak Anda terkena JavaScript, yang jelas itu menjadi. Kata-kata seperti penutupan, benda, dan metode mungkin tampak luar jangkauan kepada Anda sekarang, tapi buku ini akan membantu istilah-istilah datang ke kejelasan. Saya ingin Anda untuk menjaga dua perasaan sebelum dan setelah Anda belajar sesuatu dalam pikiran ketika Anda mulai buku ini. Ini mungkin tampak menakutkan, tapi kau mengambil buku ini karena Anda memulai perjalanan yang mengagumkan untuk mengasah pengetahuan Anda. *Up & Pergi* adalah awal dari jalan kita untuk memahami pemrograman. Nikmati saat-saat bola lampu!

- *Jenn Lukas (http://jennlukas.com , @jennlukas),*
konsultan frontend

Kata pengantar

Saya yakin Anda perhatikan, tetapi “JS” dalam judul seri bukanlah tion abbrevia- untuk kata-kata yang digunakan untuk mengutuk tentang JavaScript, meskipun mengutuk kebiasaan bahasa adalah sesuatu yang mungkin kita semua bisa mengidentifikasi dengan! Dari hari-hari awal web, JavaScript telah menjadi teknologi tional founda- yang mendorong pengalaman interaktif sekitar tenda con yang kita konsumsi. Sementara berkedip-kedip jalan mouse dan menjengkelkan pop-up petunjuknya mungkin di mana JavaScript mulai, hampir dua dekade kemudian, teknologi dan kemampuan JavaScript telah berkembang banyak pesanan besarnya, dan beberapa meragukan pentingnya di jantung platform perangkat lunak yang paling banyak tersedia di dunia : Web. Tapi sebagai bahasa, telah terus-menerus menjadi target untuk banyak kritik, karena sebagian untuk warisan tetapi bahkan lebih untuk filosofi desain. Bahkan nama membangkitkan, sebagai Brendan Eich pernah dikatakan, status “anak saudara bodoh” di samping kakak lebih matang nya, Java. Tapi nama hanyalah sebuah kecelakaan politik dan pemasaran. Dua bahasa yang sangat berbeda dalam banyak hal penting. “JavaScript” adalah sebagai terkait dengan “Jawa” sebagai “Carnival” adalah untuk “Car.” Karena JavaScript meminjam konsep dan idiom sintaks dari bahasa eral sev-, termasuk C-style akar prosedural bangga serta halus, kurang jelas Scheme / Lisp akar fungsional -gaya, itu exceed- ingly didekati untuk khalayak luas pengembang, bahkan mereka dengan sedikit atau tidak ada pengalaman pemrograman. “Hello World” dari JavaScript sangat sederhana bahwa bahasa mengundang dan mudah untuk mendapatkan nyaman dengan paparan awal. Tapi nama hanyalah sebuah kecelakaan politik dan pemasaran. Dua bahasa yang sangat berbeda dalam banyak hal penting. “JavaScript” adalah sebagai terkait dengan “Jawa” sebagai “Carnival” adalah untuk “Car.” Karena JavaScript meminjam konsep dan idiom sintaks dari bahasa eral sev-, termasuk C-style akar prosedural bangga serta halus, kurang jelas Scheme / Lisp akar fungsional -gaya, itu exceed- ingly didekati untuk khalayak luas pengembang, bahkan mereka dengan sedikit atau tidak ada pengalaman pemrograman. “Hello World” dari JavaScript sangat sederhana bahwa bahasa mengundang dan mudah untuk mendapatkan nyaman dengan paparan awal. Tapi nama hanyalah sebuah kecelakaan politik dan pemasaran. Dua bahasa yang sangat berbeda dalam banyak hal penting. “JavaScript” ad

Sementara JavaScript mungkin salah satu bahasa yang paling mudah untuk bangun dan berjalan dengan, eksentrik yang membuat penguasaan solid dari LAN-gauge kejadian jauh kurang umum daripada di banyak LAN-lain

guages. Di mana dibutuhkan cukup mendalam pengetahuan tentang bahasa seperti C atau C ++ untuk menulis sebuah program skala penuh, produksi skala penuh dapat JavaScript, dan sering, hampir tidak menggores permukaan bahasa apa yang bisa dilakukan.

konsep canggih yang berakar ke dalam bahasa cenderung malah ke permukaan diri *tampaknya* cara sederhana, seperti lewat di sekitar berfungsi sebagai callback, yang mendorong pengembang Jawa-Script untuk hanya menggunakan bahasa seperti dan tidak terlalu khawatir tentang apa yang terjadi di bawah tenda.

Hal ini secara bersamaan sederhana, mudah digunakan bahasa yang memiliki daya tarik yang luas, dan koleksi yang kompleks dan bernuansa bahasa ics mechan- bahwa tanpa studi yang cermat akan menghindari *pemahaman yang benar* bahkan untuk yang paling berpengalaman pengembang JavaScript.

Disinilah letak paradoks JavaScript, tumit Achilles' dari sebersit, tantangan kami saat menangani. karena JavaScript *bisa* digunakan tanpa pemahaman, pemahaman sebersit sering tidak pernah tercapai.

Misi

Jika pada setiap titik yang Anda temui kejutan atau frustrasi dalam JavaScript, respon Anda adalah untuk menambahkannya ke dalam daftar hitam (karena beberapa accus- tomed untuk melakukan), Anda segera akan diturunkan ke cangkang kosong dari kekayaan JavaScript .

Sementara subset ini telah terkenal dijuluki “The Parts Bagus,” Saya akan mohon Anda, pembaca yang budiman, untuk bukan menganggapnya sebagai “The Easy Parts,” “The Safe Parts,” atau bahkan “The Incomplete Parts.” Ini *Anda Tidak Tahu JS* seri menawarkan tantangan sebaliknya: belajar dan sangat memahami *semua* JavaScript, bahkan dan terutama “The Tangguh Parts.”

Di sini, kita mengatasi kepala-on kecenderungan pengembang JS untuk belajar hanya cukup untuk mendapatkan oleh, tanpa pernah memaksa diri untuk belajar persis bagaimana dan mengapa bahasa berperilaku jalan tersebut. Selain itu, kami menjauhkan diri dari saran umum untuk mundur ketika jalan mendapat kasar.

Saya tidak puas, tidak seharusnya Anda berada, untuk menghentikan sekaligus sesuatu yang hanya bekerja dan tidak benar-benar mengetahui *Mengapa.* Aku lembut menantang Anda untuk perjalanan ke "jalan yang jarang dilalui" yang bergelombang dan merangkul semua bahwa JavaScript dan dapat melakukan. Dengan pengetahuan itu, ada teknik, tidak ada kerangka, dan tidak ada kata kunci akronim populer dalam seminggu akan berada di luar pemahaman Anda.

Buku-buku ini masing-masing mengambil bagian inti tertentu dari bahasa yang paling sering disalahpahami atau di bawah-dipahami, dan menyelam dalam dan mendalam ke dalamnya. Anda harus datang jauh dari ing read dengan keyakinan teguh dalam pemahaman Anda, bukan hanya dari teori, tetapi praktis "apa yang perlu Anda ketahui" bit. JavaScript Anda tahu sekarang mungkin bagian diturunkan kepada Anda oleh orang lain yang telah dibakar oleh pemahaman yang tidak lengkap.

Bahwa JavaScript hanyalah bayangan bahasa yang benar. Anda tidak benar-benar tahu JavaScript *namun,* tetapi jika Anda menggali ke dalam seri ini, Anda akan. Membaca, teman-teman saya. JavaScript menanti Anda.

Ulasan

JavaScript mengagumkan. Sangat mudah untuk belajar sebagian, dan jauh lebih sulit untuk belajar sepenuhnya (atau bahkan *cukup).* Ketika pengembang menghadapi kebingungan, mereka biasanya menyalahkan bahasa bukan kurangnya pemahaman mereka. Buku-buku ini bertujuan untuk memperbaiki itu, inspirasi apresiasi yang kuat untuk bahasa Anda bisa sekarang, dan *harus,* sangat tahu.

Banyak contoh dalam buku ini menganggap lingkungan JavaScript engine modern (dan masa depan yang luas), seperti ES6. Beberapa kode mungkin tidak bekerja seperti yang dijelaskan jika dijalankan di lebih tua (pra-ES6) mesin.

Konvensi Digunakan dalam Buku Ini

Konvensi ketik berikut digunakan dalam buku ini:

miring Menunjukkan hal baru, URL, alamat email, nama file, dan file yang

ekstensi.

lebar konstan

Digunakan untuk listing program, serta dalam paragraf untuk merujuk program elemen seperti variabel atau fungsi nama, basis data-, tipe data, variabel lingkungan, pernyataan, dan kata-kata kunci-.

lebar konstan tebal

Menunjukkan perintah atau teks lain yang harus diketik secara harfiah oleh pengguna.

Konstan lebar italic

Menunjukkan teks yang harus diganti dengan nilai-nilai yang disediakan pengguna atau oleh nilai-nilai yang ditentukan oleh konteks.

Elemen ini menandakan tip atau saran.

Elemen ini menandakan catatan umum.

Elemen ini menunjukkan peringatan atau hati-hati.

Menggunakan Contoh Kode

bahan tambahan (contoh kode, latihan, dll) yang tersedia untuk di-download di *http://bit.ly/ydkjs-up-go*

.

Buku ini hadir untuk membantu Anda mendapatkan pekerjaan Anda selesai. Secara umum, jika exam- kode ple ditawarkan dengan buku ini, Anda dapat menggunakannya dalam program dan dokumentasi. Anda tidak perlu menghubungi kami untuk izin kecuali Anda mereproduksi porsi yang signifikan dari kode. Misalnya, menulis sebuah program yang menggunakan beberapa potongan kode dari buku ini tidak memerlukan izin. Jual atau mendistribusikan CD-ROM dari contoh-contoh dari buku O'Reilly tidak memerlukan izin. Menjawab pertanyaan dengan mengutip buku ini dan mengutip contoh kode

tidak memerlukan izin. Menggabungkan sejumlah besar contoh kode dari buku ini ke dalam dokumentasi produk Anda tidak memerlukan izin.

Kami menghargai, tapi tidak memerlukan, atribusi. Atribusi usu- sekutu mencakup judul, penulis, penerbit, dan ISBN. Sebagai contoh: " *Anda Tidak Tahu JavaScript: Up & Pergi* oleh Kyle Simpson (O'Reilly). Copyright 2015 Getify Solutions, Inc., 978-1-491-92446-4."Jika Anda merasa penggunaan contoh kode jatuh digunakan di luar wajar atau misi per- diberikan di atas,

merasa ragu untuk menghubungi kami di *permis- sions@oreilly.com* .

Safari® Buku Online

Safari Buku Online adalah on-demand perpustakaan digital yang memberikan ahli kadar di kedua buku dan bentuk video dari dunia penulis ing lead dalam teknologi dan bisnis.

profesional teknologi, pengembang perangkat lunak, web desainer, dan bisnis dan kreatif profesional menggunakan Safari Buku Online sebagai sumber utama mereka untuk penelitian, pemecahan masalah, belajar, dan pelatihan ication certif-.

Safari Buku Online menawarkan berbagai rencana dan harga untuk sional . pemerintah . pendidikan , Dan individu.

Anggota memiliki akses ke ribuan buku, video pelatihan, dan naskah prapublikasi dalam satu database sepenuhnya dicari dari penerbit seperti O'Reilly Media, Prentice Hall profesional, Addison- Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe press, FT press, Apress, Manning, Riders Baru, McGraw-Hill, Jones & Bartlett, Kursus Tech- nology, dan ratusan lebih . Untuk informasi lebih lanjut tentang Safari Buku Online, silahkan kunjungi kami on line .

Cara Hubungi Kami

Silahkan mengatasi komentar dan pertanyaan tentang buku ini ke penerbit:

O'Reilly Media, Inc.
1005 Gravenstein Jalan Raya Utara
Sebastopol, CA 95472
800-998-9938 (di Amerika Serikat atau Kanada)
707-829-0515 (internasional atau lokal) 707-829-0104 (faks)

Kami memiliki halaman web untuk buku ini, di mana kita daftar ralat, contoh, dan informasi tambahan. Anda dapat mengakses halaman ini di *http: // bit.ly/ydkjs_up-and-going* .

Untuk komentar atau mengajukan pertanyaan teknis tentang buku ini, kirim email ke *bookquestions@oreilly.com* .

Untuk informasi lebih lanjut tentang buku-buku kami, kursus, konferensi, dan berita, lihat website kami di *http://www.oreilly.com* .

Temukan kami di facebook: *http://facebook.com/oreilly*

Ikuti kami di Twitter: *http://twitter.com/oreillymedia*

Watch kami di YouTube: *http://www.youtube.com/oreillymedia*

BAB 1

ke Programming

Selamat datang ke *Anda Tidak Tahu JS (YDKJS)* seri.

Up & Pergi adalah pengenalan beberapa konsep dasar Program- ming-tentu saja kita bersandar ke arah JavaScript (sering disingkat JS) khusus-dan bagaimana pendekatan dan memahami sisa judul dalam seri ini. Terutama jika Anda baru saja masuk ke pemrograman dan / atau JavaScript, buku ini secara singkat akan mengeksplorasi apa yang Anda butuhkan untuk mendapatkan
dan pergi.

Buku ini dimulai menjelaskan prinsip-prinsip dasar pemrograman pada tingkat yang sangat tinggi. Ini lebih ditujukan jika Anda memulai *YDKJS*
dengan sedikit tidak pengalaman pemrograman sebelumnya, dan mencari untuk buku-buku ini untuk membantu Anda mulai sepanjang jalan untuk memahami pemrograman melalui lensa JavaScript.

Bab 1 harus didekati sebagai gambaran singkat dari hal-hal yang Anda akan ingin belajar lebih banyak tentang dan berlatih untuk mendapatkan *ke Program- ming.* Ada juga banyak sumber informasi lainnya pemrograman fantastis pengantar tion yang dapat membantu Anda menggali ke dalam topik ini lebih jauh, dan saya mendorong Anda untuk belajar dari mereka selain bab ini. Setelah Anda merasa nyaman dengan dasar-dasar pemrograman umum, Bab 2 akan membantu memandu Anda untuk keakraban dengan rasa JavaScript tentang pemrograman. Bab 2 memperkenalkan apa JavaScript adalah tentang, tapi sekali lagi, itu bukan panduan-yang komprehensif yang sisa

YDKJS buku adalah untuk!

Jika Anda sudah cukup nyaman dengan JavaScript, pertama periksa bagian 3 sebagai sekilas singkat apa yang diharapkan dari *YDKJS,* kemudian melompat tepat di!

Kode

Mari kita mulai dari awal. Sebuah program, sering disebut sebagai *Kode sumber* atau hanya *kode,* adalah satu set instruksi khusus untuk memberitahu komputer apa tugas untuk melakukan. Kode sekutu Usu- disimpan dalam sebuah file teks, walaupun dengan JavaScript Anda juga dapat mengetik kode langsung ke konsol pengembang di browser, yang kita akan membahas lama.

Aturan untuk format dan kombinasi instruksi valid disebut *bahasa komputer,* kadang-kadang disebut sebagai yang *sintaksis,* sama seperti bahasa Inggris memberitahu Anda bagaimana mengeja kata-kata dan cara membuat kalimat yang valid menggunakan kata-kata dan tanda baca.

laporan

Dalam bahasa komputer, kelompok kata, angka, dan operator yang melakukan tugas tertentu adalah *pernyataan.* Dalam JavaScript, pernyataan akan terlihat sebagai berikut:

```
Sebuah = b * 2;
```

karakter Sebuah dan b disebut *variabel (* Lihat "Variabel" pada halaman 14), Yang seperti kotak sederhana yang dapat menyimpan barang-barang Anda di. Dalam program, variabel memegang nilai-nilai (seperti nomor 42) yang akan digunakan oleh program. Menganggap mereka sebagai penampung simbolis untuk ues val- sendiri. Sebaliknya, 2 hanya nilai itu sendiri, yang disebut *nilai literal,* karena berdiri sendiri tanpa disimpan dalam variabel. = Dan karakter * adalah *operator (* Lihat "Operator" di halaman 8) - mereka melakukan tindakan dengan nilai-nilai dan variabel seperti ment menetapkan- dan perkalian matematika.

Kebanyakan pernyataan dalam JavaScript menyimpulkan dengan titik koma (;) di akhir.

Pernyataan a = b * 2; memberitahu komputer, kasar, untuk mendapatkan nilai saat ini disimpan dalam variabel b, kalikan nilai dengan 2, kemudian menyimpan hasilnya kembali ke variabel lain yang kita sebut Sebuah.

Program hanya koleksi banyak pernyataan tersebut, yang bersama-sama menggambarkan semua langkah-langkah yang dibutuhkan untuk melakukan tujuan pro gram Anda.

ekspresi

Laporan terdiri dari satu atau lebih *ekspresi.* Sebuah ekspresi adalah referensi ke variabel atau nilai, atau satu set variabel (s) dan nilai (s) dikombinasikan dengan operator. Sebagai contoh:

```
Sebuah = b * 2 ;
```

Pernyataan ini memiliki empat ekspresi di dalamnya:

- 2 adalah *nilai literal ekspresi.*
- b adalah *variabel ekspresi,* yang berarti untuk mengambil nilai saat ini.
- b * 2 adalah *ekspresi aritmatika,* yang berarti untuk melakukan lipatan multi.
- a = b * 2 adalah *ekspresi tugas,* yang berarti untuk menetapkan hasil b * 2 ekspresi variabel Sebuah (lebih pada tugas nanti).

Ekspresi umum yang berdiri sendiri juga disebut *pernyataan ekspresi,* seperti berikut:

```
b * 2 ;
```

rasa ini pernyataan ekspresi tidak sangat umum atau berguna, seperti umumnya tidak akan memiliki efek pada jalannya program

- itu akan mengambil nilai dari b dan kalikan dengan 2, tapi kemudian tidak akan melakukan apa-apa dengan hasil itu. Sebuah pernyataan ekspresi yang lebih umum adalah *ekspresi panggilan* Pernyataan (lihat "Fungsi" pada halaman 22), Seperti seluruh pernyataan adalah ekspresi fungsi panggilan itu sendiri:

```
waspada ( Sebuah );
```

Pelaksana Program a

Bagaimana mereka koleksi laporan pemrograman memberitahu com- puter apa yang harus dilakukan? Program ini perlu *dieksekusi,* juga disebut sebagai *menjalankan program.*

pernyataan seperti a = b * 2 membantu untuk pengembang ketika membaca dan menulis, tetapi tidak benar-benar dalam bentuk komputer dapat langsung mengerti. Jadi utilitas khusus di komputer (baik *penerjemah* atau *penyusun)* digunakan untuk menerjemahkan kode yang anda tulis ke mands com- komputer dapat memahami.

Untuk beberapa bahasa komputer, terjemahan ini perintah adalah Cally typi- dilakukan dari atas ke bawah, baris demi baris, setiap kali program dijalankan, yang biasanya disebut *menafsirkan* Kode.

Untuk bahasa lain, terjemahan dilakukan dari waktu ke depan, disebut *kompilasi* kode, sehingga ketika program *berjalan* kemudian, apa yang berjalan sebenarnya adalah instruksi komputer sudah dikompilasi siap untuk pergi. Ini biasanya menegaskan bahwa JavaScript adalah *ditafsirkan,* karena kode sumber Jawa-Script Anda diproses setiap kali dijalankan. Tapi itu tidak sepenuhnya akurat. Mesin JavaScript sebenarnya *kompilasi* gram pro dengan cepat dan kemudian segera menjalankan kode dikompilasi.

Untuk informasi lebih lanjut tentang kompilasi JavaScript, melihat dua bab pertama dari *Lingkup & Penutup* judul seri ini.

Cobalah sendiri

Bab ini akan memperkenalkan konsep masing-masing pemrograman dengan potongan kode sederhana, semua yang ditulis dalam JavaScript (jelas!). Hal ini tidak dapat ditekankan cukup: saat Anda pergi melalui bab ini

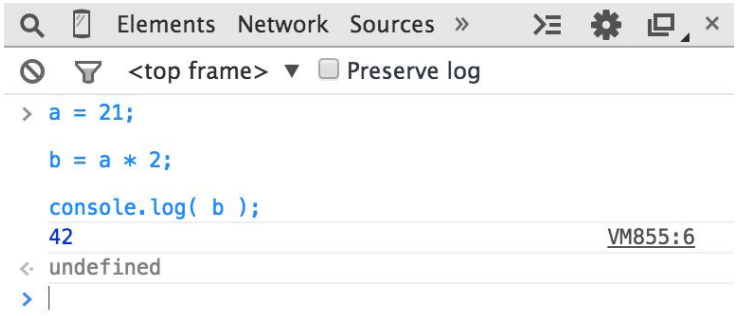
- dan Anda mungkin perlu untuk menghabiskan waktu untuk pergi di atasnya beberapa kali-Anda harus berlatih masing-masing konsep dengan mengetikkan kode Anda- diri. Cara termudah untuk melakukannya adalah untuk membuka alat pengembang satunya con di browser terdekat (Firefox, Chrome, IE, dll).

Biasanya, Anda dapat meluncurkan konsol pengembang dengan shortcut keyboard atau dari item menu. Untuk informasi lebih lanjut tentang peluncuran dan menggunakan konsol di browser favorit Anda, lihat "Menguasai Tools Konsol Pengembang" . Untuk mengetik beberapa baris ke konsol sekaligus, gunakan < pergeseran> + <enter> untuk berpindah ke baris baru berikutnya. Setelah anda menekan < enter> dengan sendirinya, konsol akan menjalankan segala sesuatu yang baru saja diketik.

Mari kita mendapatkan akrab dengan proses menjalankan kode di konsol. Pertama, saya sarankan membuka tab kosong di browser Anda. Saya lebih suka untuk melakukan hal ini dengan mengetikkan tentang kosong ke address bar. Kemudian, pastikan konsol pengembang Anda terbuka, karena kami hanya disebutkan. Sekarang, ketik kode ini dan melihat bagaimana itu berjalan:

```
Sebuah = 21 ;

b = Sebuah * 2 ;

menghibur . mencatat ( b );
```

Mengetik kode sebelumnya ke konsol di Chrome harus pro Duce sesuatu seperti berikut:



Ayo, mencobanya. Cara terbaik untuk belajar pemrograman adalah untuk memulai coding!

Keluaran

Dalam potongan kode sebelumnya, kami menggunakan console.log (..). Secara singkat, mari kita lihat apa yang baris kode adalah semua tentang.

Anda mungkin telah menebak, tapi itulah bagaimana kita mencetak teks (alias *put out* untuk pengguna) di konsol pengembang. Ada dua tics characteris- dari pernyataan bahwa kita harus menjelaskan. Pertama, log (b) Bagian ini disebut sebagai panggilan fungsi (lihat "Tions Func-" pada halaman 22). Apa yang terjadi adalah kita menyerahkan b variabel untuk fungsi itu, yang meminta untuk mengambil nilai b dan mencetaknya ke konsol. Kedua, menghibur. Bagian adalah sebuah referensi obyek mana log (..)

Fungsi berada. Kami meliputi benda dan sifat mereka secara lebih rinci dalam Bab 2 .

Cara lain untuk menciptakan output yang dapat Anda lihat adalah dengan menjalankan waspada(..) pernyataan. Sebagai contoh:

```
waspada ( b );
```

Jika Anda menjalankan itu, Anda akan melihat bahwa bukannya mencetak output ke konsol, hal itu menunjukkan pop-up "OK" kotak dengan isi dari b variabel. Namun, dengan menggunakan console.log (..) umumnya akan membuat belajar tentang coding dan menjalankan program Anda di satu-satunya con- lebih mudah daripada menggunakan waspada(..) karena Anda dapat output banyak ues val- sekaligus tanpa mengganggu antarmuka browser. Untuk buku ini, kita akan menggunakan console.log (..) untuk output.

Memasukkan

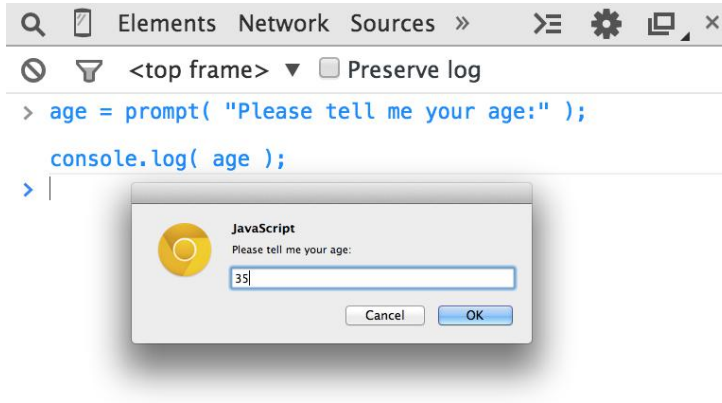
Sementara kita membahas output, Anda juga mungkin bertanya-tanya tentang *memasukkan* (Yaitu, menerima informasi dari pengguna).

Cara yang paling umum yang terjadi adalah untuk halaman HTML untuk menampilkan elemen bentuk (seperti kotak teks) untuk pengguna bahwa dia dapat mengetik ke dalam, dan kemudian menggunakan JS untuk membaca nilai-nilai tersebut ke dalam variabel program anda. Tapi ada cara yang lebih mudah untuk mendapatkan masukan untuk tujuan pembelajaran sederhana dan onstration dem- seperti apa yang Anda akan lakukan sepanjang buku ini. Menggunakan cepat(..) fungsi:

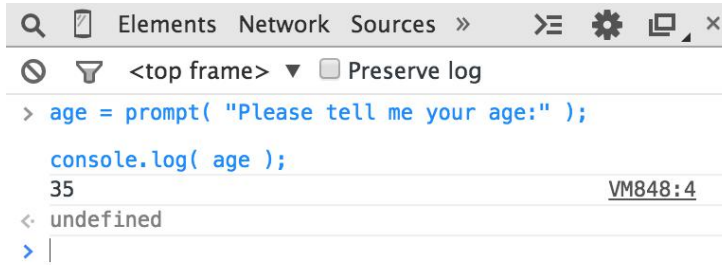
```
usia = cepat ( "Tolong beritahu saya usia Anda:" );

menghibur . mencatat ( usia );
```

Seperti yang bisa Anda tebak, pesan Anda lolos ke cepat(..) -pada kasus ini, " Tolong beritahu saya usia Anda:" -Apakah dicetak ke dalam pop up.

Ini akan terlihat mirip dengan berikut ini:



Setelah Anda mengirimkan input teks dengan mengklik “OK,” Anda akan melihat bahwa nilai Anda mengetik disimpan dalam usia variabel, yang kemudian kita *put out* dengan console.log (..):



Untuk menjaga hal-hal sederhana sementara kita belajar dasar cepts pemrograman con, contoh dalam buku ini tidak akan memerlukan input. Tapi sekarang bahwa Anda telah melihat bagaimana menggunakan cepat(..), jika Anda ingin menantang itu- diri, Anda dapat mencoba menggunakan masukan dalam eksplorasi Anda dari contoh.

operator

Operator bagaimana kita melakukan tindakan pada variabel dan nilai-nilai. Kita sudah melihat dua operator JavaScript, yang = dan *. The * operator melakukan perkalian matematika. Cukup sederhana, kan?

The = equals operator yang digunakan untuk *tugas* -kita pertama menghitung nilai pada *sisi kanan (* nilai sumber) dari = dan kemudian memasukkannya ke dalam variabel yang kita tentukan pada *sisi kiri (* Target variabel- mampu).

Hal ini mungkin tampak seperti urutan terbalik aneh untuk menentukan tugas. Dari pada a = 42, beberapa mungkin lebih memilih untuk membalik urutan sehingga nilai sumber di sebelah kiri dan variabel target di sebelah kanan, seperti 42 -> a (ini tidak valid JavaScript!). Sayangnya a = 42 memerintahkan bentuk, dan variasi serupa, cukup lazim dalam bahasa pemrograman modern. Jika terasa tidak wajar, hanya menghabiskan waktu berlatih agar dalam pikiran Anda untuk terbiasa dengan itu.

Mempertimbangkan:

```
Sebuah = 2;
b = Sebuah + 1;
```

Di sini, kita menetapkan 2 nilai ke Sebuah variabel. Kemudian, kita mendapatkan nilai Sebuah variabel (masih 2), menambahkan 1 untuk itu mengakibatkan nilai 3, kemudian menyimpan bahwa nilai dalam b variabel.

Meskipun tidak secara teknis operator, Anda harus kata kunci var dalam setiap program, karena itu cara utama Anda *menyatakan (* aka *buat) var* iables (lihat "Variabel" pada halaman 14).

Anda harus selalu mendeklarasikan variabel dengan nama sebelum Anda menggunakannya. Tapi Anda hanya perlu mendeklarasikan variabel sekali untuk setiap *lingkup (* Lihat "Lingkup" pada halaman 24); dapat digunakan sebanyak setelah itu yang diperlukan. Sebagai contoh:

```
var Sebuah = 20;

Sebuah = Sebuah + 1;
Sebuah = Sebuah * 2;
```

```
menghibur . mencatat ( Sebuah ); // 42
```

Berikut adalah beberapa operator yang paling umum dalam JavaScript:

Tugas

=, seperti dalam a = 2.

matematika

\+ (Selain itu), - (pengurangan), * (perkalian), dan / (sion divi-), seperti dalam a * 3.

senyawa tugas

+ =, - =, * =, dan / = adalah operator senyawa yang menggabungkan operasi matematika dengan tugas, seperti di a + = 2 (sama dengan a = a + 2).

Kenaikan / pengurangan

+ + (Increment), - (pengurangan), seperti dalam a ++ (mirip dengan a = a + 1).

akses properti objek

. seperti dalam console.log ().

Benda adalah nilai-nilai yang memegang nilai-nilai lain di tertentu DENGAN LOKASI bernama disebut sifat. obj.a berarti nilai objek yang disebut obj dengan properti dari nama Sebuah. Properti dapat alternatif diakses sebagai obj ["a"]. Lihat Bab 2 .

Persamaan

== (longgar-sama), === (ketat-equals),! = (longgar tidak-sama),! == (ketat tidak-sama), seperti dalam a == b.

Lihat “Nilai & Jenis” pada halaman 10 dan Bab 2 .

Perbandingan

<(kurang dari),> (lebih besar dari), <= (kurang dari atau longgar-sama dengan),> = (lebih besar atau longgar-sama), seperti dalam a <= b.

Lihat “Nilai & Jenis” pada halaman 10 dan Bab 2 .

Logis

&& (dan), || (Atau), seperti dalam a || b yang memilih baik Sebuah *atau* b.

Operator ini digunakan untuk mengekspresikan conditional senyawa (lihat “Pengandaian” pada halaman 18), Seperti jika baik Sebuah *atau* b adalah benar.

Untuk lebih detail, dan cakupan operator tidak disebutkan di sini, lihat Mozilla Developer Network (MDN) 's "Ekspresi dan Operator" .

Nilai & Jenis

Jika Anda bertanya seorang karyawan di sebuah toko telepon berapa banyak biaya telepon tertentu, dan ia mengatakan "sembilan puluh sembilan, sembilan puluh sembilan" (yaitu, $ 99,99), dia giv- ing Anda seorang tokoh dolar numerik yang sebenarnya yang mewakili apa yang akan Anda perlu membayar (plus pajak) untuk membelinya. Jika Anda ingin membeli dua ponsel tersebut, Anda dapat dengan mudah melakukan matematika mental untuk dua kali lipat nilai untuk mendapatkan $ 199,98 untuk biaya dasar Anda.

Jika bahwa karyawan yang sama mengangkat telepon lain yang serupa tetapi mengatakan itu "bebas" (mungkin dengan kutipan udara), dia tidak memberikan angka, melainkan jenis lain dari representasi biaya yang diharapkan ($ 0.00)

- kata "bebas."

Ketika nanti menanyakan apakah telepon termasuk pengisi daya, jawabannya hanya bisa "ya" atau "tidak."

Dalam cara yang sangat mirip, ketika Anda mengekspresikan nilai-nilai dalam program, Anda memilih representasi yang berbeda untuk nilai-nilai berdasarkan pada apa yang Anda rencanakan dengan mereka.

representasi ini berbeda untuk nilai-nilai yang disebut *jenis* di pro gramming terminologi. JavaScript memiliki built-in jenis untuk masing-masing disebut ini *primitif* nilai-nilai:

- Bila Anda perlu melakukan matematika, Anda ingin jumlah.
- Bila Anda perlu untuk mencetak nilai pada layar, Anda perlu String (satu atau lebih karakter, kata, atau kalimat).
- Bila Anda perlu membuat keputusan dalam program Anda, Anda perlu boolean (true atau Salah).

Nilai-nilai yang disertakan langsung dalam kode sumber disebut *harafiah als.* tali literal dikelilingi oleh tanda kutip ganda ("...") atau tanda kutip gle sin- ('...') - satu-satunya perbedaan adalah preferensi gaya.

jumlah dan boolean literal hanya disajikan sebagai adalah (misalnya, 42, benar, dll).

Mempertimbangkan:

```
"Saya string" ;
'Saya juga string' ;

42 ;

benar ;
Salah ;
```

Luar string / nomor / boolean jenis nilai, itu umum untuk bahasa pro gramming untuk memberikan *array, benda, fungsi,* dan banyak lagi. Kita akan membahas lebih banyak tentang nilai-nilai dan jenis seluruh bab ini dan berikutnya.

Konversi Antara Jenis

Jika Anda memiliki jumlah tetapi perlu untuk mencetaknya di layar, Anda perlu mengkonversi nilai ke tali, dan dalam JavaScript konversi ini disebut "paksaan." Demikian pula, jika seseorang memasuki serangkaian karakter numerik ke dalam formulir di halaman e-commerce, itu tali, tetapi jika Anda perlu untuk kemudian menggunakan nilai bahwa untuk melakukan operasi matematika, Anda perlu

memaksa ke sebuah jumlah.

JavaScript menyediakan beberapa fasilitas yang berbeda untuk secara paksa memaksa antara *jenis.* Sebagai contoh:

```
var Sebuah = "42" ;
var b = Jumlah ( Sebuah );

menghibur . mencatat ( Sebuah ); // " 42"
menghibur . mencatat ( b ); // 42
```

menggunakan Jumlah (..) (built-in fungsi) seperti yang ditunjukkan adalah *eksplisit* cion coer- dari jenis lain dengan jumlah mengetik. Itu harus cukup sederhana.

Tapi topik yang kontroversial adalah apa yang terjadi ketika Anda mencoba untuk membandingkan dua nilai yang belum dari jenis yang sama, yang akan membutuhkan *implisit* paksaan. Ketika membandingkan string " 99.99" ke nomor 99.99, kebanyakan orang akan setuju mereka setara. Tapi mereka tidak persis sama, mereka? Ini nilai yang sama dalam dua representasi yang berbeda, dua yang berbeda *jenis.* Anda bisa mengatakan mereka "longgar yang sama," tidak bisa Anda?

Untuk membantu Anda dalam situasi-situasi umum, JavaScript akan kadang kali menendang dan *implisit* memaksa nilai ke jenis yang cocok. Jadi, jika Anda menggunakan == longgar-sama operator untuk membuat perbandingan "99.99" == 99.99, JavaScript akan mengkonversi sisi kiri "99,99" untuk yang jumlah setara 99.99. perbandingan kemudian menjadi 99.99 == 99.99, yang tentu saja benar.

Sementara dirancang untuk membantu Anda, pemaksaan implisit dapat membuat kebingungan jika Anda tidak mengambil waktu untuk mempelajari aturan-aturan yang mengatur IOR prilaku nya. Sebagian besar pengembang JS tidak pernah memiliki, sehingga perasaan umum adalah bahwa pemaksaan implisit membingungkan dan merugikan program dengan bug yang tak terduga, dan dengan demikian harus dihindari. Itu bahkan kadang-kadang disebut cacat dalam desain bahasa.

Namun, pemaksaan implisit adalah mekanisme yang *bisa dipelajari,* dan terlebih lagi *harus dipelajari* oleh siapa saja yang ingin mengambil pemrograman JavaScript serius. Tidak hanya itu tidak membingungkan setelah Anda mempelajari aturan, itu benar-benar dapat membuat program Anda lebih baik! Upaya ini juga worth it.

Untuk informasi lebih lanjut tentang pemaksaan, lihat Bab 2 dari judul ini dan Bab 4 dari *Jenis & Grammar* judul seri ini.

kode Komentar

Karyawan toko telepon mungkin menuliskan beberapa catatan pada membangun struktur kendala pada aspek dari ponsel yang baru dirilis atau rencana baru perusahaannya menawarkan. Catatan ini hanya untuk karyawan-mereka bukan untuk tomers cus- untuk membaca. Namun demikian, catatan ini membantu karyawan melakukan pekerjaan dengan lebih baik dengan mendokumentasikan bagaimana dan mengapa dari apa yang harus dia katakan pelanggan.

Salah satu pelajaran yang paling penting Anda dapat mempelajari tentang menulis kode adalah bahwa itu bukan hanya untuk komputer. Kode adalah setiap bit sebagai banyak, jika tidak lebih, untuk pengembang seperti itu untuk compiler.

komputer Anda hanya peduli tentang kode mesin, serangkaian 0s biner dan 1s, yang berasal dari *kompilasi.* Ada ber NUM hampir tak terbatas program Anda bisa menulis yang menghasilkan seri yang sama dari 0s dan 1s. Pilihan yang Anda buat tentang bagaimana menulis materi program anda

- tidak hanya untuk Anda, tetapi untuk anggota tim lain dan bahkan ke masa depan diri Anda.

Anda harus berusaha tidak hanya untuk menulis program yang bekerja dengan benar, tapi program yang masuk akal ketika diperiksa. Anda dapat pergi jauh dalam upaya itu dengan memilih **nama-nama yang baik untuk variabel Anda (lihat "Variabel- ables" pada halaman 14) Dan fungsi (lihat "Fungsi" pada halaman 22). Tapi bagian penting lain adalah kode komentar. Ini adalah** potongan-potongan teks dalam program Anda yang dimasukkan murni untuk menjelaskan hal-hal untuk manusia. Juru / compiler akan selalu mengabaikan komentar-komentar.

Ada banyak opini yang objektif tentang apa yang membuat kode baik berkomentar; kita tidak bisa benar-benar menentukan aturan universal yang absolut. Tetapi beberapa pengamatan dan pedoman yang cukup berguna:

- Kode tanpa komentar adalah suboptimal.
- Terlalu banyak komentar (satu per baris, misalnya) mungkin merupakan tanda kode yang ditulis dengan buruk.
- **Komentar harus menjelaskan *Mengapa,* tidak *apa.* Mereka opsional bisa menjelaskan *bagaimana***

jika apa yang ditulis sangat membingungkan. Dalam JavaScript, ada dua jenis komentar yang mungkin: komentar single-line dan komentar multiline. Mempertimbangkan:

```
// Ini adalah single-line komentar

/* Tapi ini adalah
        multiline sebuah
                komentar.
                        * /
```

The // single-line komentar yang tepat jika Anda akan menempatkan komentar tepat di atas pernyataan tunggal, atau bahkan di akhir baris. Semuanya pada baris setelah // diperlakukan sebagai komentar (dan dengan demikian diabaikan oleh compiler), semua jalan ke akhir baris. Tidak ada batasan untuk apa yang dapat muncul di dalam single-line ment com-. Mempertimbangkan:

```
var Sebuah = 42 ;          // 42 adalah makna hidup
```

The / * .. * / Multiline komentar adalah tepat jika Anda memiliki beberapa baris senilai penjelasan untuk membuat dalam komentar Anda. Berikut adalah penggunaan umum dari komentar multiline:

```
/ * Nilai berikut digunakan karena
     telah menunjukkan bahwa hal itu menjawab setiap
     pertanyaan di alam semesta. * /
var Sebuah = 42 ;
```

Hal ini juga dapat muncul di mana saja pada baris, bahkan di tengah-tengah garis, karena * / berakhir itu. Sebagai contoh:

```
var Sebuah = / * sewenang-wenang nilai * / 42 ;

menghibur . mencatat ( Sebuah ); // 42
```

Satu-satunya hal yang tidak bisa muncul di dalam komentar multiline adalah * /, karena itu akan ditafsirkan untuk mengakhiri komentar. Anda pasti akan ingin mulai belajar Anda pemrograman dengan memulai off dengan kebiasaan komentar kode. Sepanjang sisa bab ini, Anda akan melihat saya menggunakan komentar untuk menjelaskan hal-hal, sehingga melakukan hal yang sama dalam praktek Anda sendiri. Percayalah, semua orang yang membaca kode Anda akan terima kasih!

variabel

program yang paling berguna perlu melacak nilai karena perubahan selama program, menjalani operasi yang berbeda seperti yang disebut oleh tugas-tugas program anda dimaksudkan.

Cara termudah untuk pergi tentang itu dalam program Anda adalah untuk memberikan nilai pada wadah simbolik, yang disebut *variabel* -jadi disebut karena nilai dalam wadah ini bisa *berbeda* dari waktu ke waktu sesuai kebutuhan. Dalam beberapa bahasa pemrograman, Anda mendeklarasikan variabel (wadah) untuk mengadakan jenis tertentu nilai, seperti jumlah atau tali. *Statis ing typ-,* atau dikenal sebagai *penegakan jenis,* biasanya dikutip sebagai efit ben untuk ketepatan program dengan mencegah konversi nilai yang tidak diinginkan.

Bahasa lainnya menekankan jenis untuk nilai bukan variabel. *mengetik lemah,* atau dikenal sebagai *mengetik dinamis,* memungkinkan sebuah variabel untuk memegang jenis nilai setiap saat. Ini biasanya disebut sebagai manfaat untuk fleksibilitas program dengan memungkinkan variabel tunggal untuk mewakili

nilai tidak peduli apa jenis bentuk nilai yang dapat mengambil pada saat tertentu di alur logika program.

JavaScript menggunakan pendekatan yang terakhir, *mengetik dinamis,* berarti ables variabel- dapat memegang nilai-nilai dari setiap *mengetik* Tanpa apapun *mengetik* pelaksanaan. Seperti disebutkan sebelumnya, kita mendeklarasikan variabel menggunakan var pernyataan

- melihat tidak ada lain *mengetik* informasi dalam deklarasi. Con- Sider program sederhana ini:

```
var jumlah = 99.99 ;

jumlah = jumlah * 2 ;

menghibur . mencatat ( jumlah );                    // 199,98

// mengkonversi `amount` ke string, dan // tambahkan
"$" di awal
jumlah = "$" + Tali ( jumlah );

menghibur . mencatat ( jumlah );                    // "$ 199,98"
```

Itu jumlah variabel dimulai memegang nomor 99.99, dan kemudian memegang jumlah Hasil dari Jumlah * 2, yang mana 199,98.

Pertama console.log (..) perintah harus *implisit* memaksa bahwa jumlah nilai ke tali untuk mencetaknya. Kemudian pernyataan Jumlah = "$" + String (jumlah) *secara eksplisit* memaksa para 199,98 nilai ke tali dan menambahkan "$" karakter ke awal. Pada saat ini, jumlah sekarang memegang tali nilai "$ 199,98", sehingga kedua console.log (..) Pernyataan tidak perlu melakukan pemaksaan untuk mencetaknya.

pengembang JavaScript akan mencatat fleksibilitas menggunakan jumlah variabel untuk masing-masing 99.99, 199,98, dan "$ 199,98" nilai-nilai. penggemar statis-mengetik akan lebih memilih variabel yang terpisah seperti amountStr untuk memegang akhir "$ 199,98" representasi dari nilai, karena tipe yang berbeda. Either way, Anda akan perhatikan bahwa jumlah memegang nilai berjalan yang berubah selama program, yang menggambarkan tujuan utama dari variabel: Program pengelolaan *negara.*

Dengan kata lain, *negara* melacak perubahan nilai-nilai sebagai gram berjalan pro Anda.

penggunaan umum lain variabel adalah untuk memusatkan pengaturan nilai. Ini lebih biasanya disebut *konstanta,* ketika Anda mendeklarasikan variabel dengan nilai dan berniat untuk nilai itu untuk *tidak berubah* seluruh program.

Anda menyatakan konstanta ini, sering di bagian atas program, sehingga lebih mudah bagi Anda untuk memiliki satu tempat untuk pergi untuk mengubah nilai jika Anda perlu. Dengan konvensi, variabel JavaScript sebagai konstanta biasanya dikapitalisasi, dengan garis bawah _ antara beberapa kata. Berikut adalah contoh konyol:

```
var PERSENTASE PAJAK = 0,08 ;          // pajak penjualan 8%

var jumlah = 99.99 ;

jumlah = jumlah * 2 ;

jumlah = jumlah + ( jumlah * PERSENTASE PAJAK );

menghibur . mencatat ( jumlah );                    // 215.9784
menghibur . mencatat ( jumlah . toFixed ( 2 ) ); // " 215,98"
```

Mirip dengan bagaimana console.log (..) adalah fungsi log (..) diakses sebagai properti objek pada menghibur nilai, toFixed (..) di sini adalah fungsi yang bisa diakses di jumlah nilai-nilai. JavaScript jumlah s tidak secara otomatis diformat untuk lars-the dol- mesin tidak tahu apa tujuan Anda adalah, dan tidak ada jenis mata uang. toFixed (..) memungkinkan kita menentukan berapa banyak tempat desimal kami ingin yang jumlah dibulatkan menjadi, dan itu menghasilkan tali seperlunya.

Itu PERSENTASE PAJAK variabel hanya *konstan* dengan konvensi-ada noth- ing khusus dalam program ini yang mencegah dari yang berubah. Tetapi jika kota meningkatkan tarif pajak penjualan untuk 9%, kita masih dapat dengan mudah memperbarui program kami dengan menetapkan PERSENTASE PAJAK nilai yang diberikan untuk 0,09 di satu tempat, bukan mencari banyak kejadian nilai 0,08 bertebaran di seluruh program dan memperbarui mereka semua.

Versi terbaru dari JavaScript pada saat ini menulis (monly com- disebut “ES6”) termasuk cara baru untuk menyatakan konstanta, dengan menggunakan const dari pada var:

```
// pada ES6:
const PERSENTASE PAJAK = 0,08 ;

var jumlah = 99.99 ;

// ..
```

Konstanta berguna seperti variabel dengan nilai-nilai tidak berubah, kecuali bahwa konstanta juga mencegah sengaja mengubah nilai-kadang di mana lagi setelah setting awal. Jika Anda mencoba untuk menetapkan nilai apapun yang berbeda untuk PERSENTASE PAJAK setelah itu deklarasi pertama, program anda akan menolak perubahan (dan dalam modus ketat, gagal dengan kesalahan-lihat "Ketat Mode" pada halaman 45 di Bab 2).

By the way, semacam "perlindungan" terhadap kesalahan mirip dengan statis-mengetik jenis penegakan hukum, sehingga Anda dapat melihat mengapa jenis statis dalam bahasa lain dapat menarik!

Untuk informasi lebih lanjut tentang bagaimana yang berbeda val- ues di variabel dapat digunakan dalam program Anda, lihat *Jenis & Grammar* judul seri ini.

blok

Karyawan toko ponsel harus melalui serangkaian langkah-langkah untuk com- plete kasir saat Anda membeli ponsel baru Anda.

Demikian pula, dalam kode kita sering perlu untuk kelompok serangkaian pernyataan bersama, yang sering kita sebut *blok*. Dalam JavaScript, blok didefinisikan dengan membungkus satu atau lebih pernyataan di dalam sepasang keriting brace {..}. Mempertimbangkan:

```
var jumlah = 99.99 ;

// blok umum
{
    jumlah = jumlah * 2 ;
    menghibur . mencatat ( jumlah ); // 199,98
}
```

Ini jenis mandiri { .. } Blok umum berlaku, tetapi tidak seperti yang sering terlihat dalam program JS. Biasanya, blok yang melekat pada beberapa pernyataan kontrol lainnya, seperti jika Pernyataan (lihat "Tionals menderita penyakit" pada halaman 18) Atau loop (lihat "Loops" pada halaman 20). Sebagai contoh:

```
var jumlah = 99.99 ;

// adalah jumlah yang cukup besar?
jika ( jumlah > 10 ) {                    // <- blok yang melekat pada `if`
        jumlah = jumlah * 2 ;
        menghibur . mencatat ( jumlah ); // 199,98
}
```

Kami akan menjelaskan jika pernyataan dalam bagian berikutnya, tetapi karena Anda dapat melihat, { .. } Blok dengan dua pernyataan yang melekat jika (jumlah > 10); pernyataan di dalam blok hanya akan diproses jika melewati bersyarat.

Tidak seperti kebanyakan pernyataan lainnya seperti menipu sole.log (jumlah) ;, pernyataan blok tidak perlu titik koma (;) untuk menyimpulkan hal itu.

conditional

"Apakah Anda ingin menambahkan pada pelindung layar ekstra untuk mengejar pur- Anda, untuk $ 9,99?" The karyawan toko ponsel membantu telah meminta Anda untuk membuat keputusan. Dan Anda mungkin harus terlebih dahulu berkonsultasi saat ini *negara* dari dompet atau rekening bank untuk menjawab pertanyaan itu. Tapi jelas, ini hanya sederhana "ya atau tidak" pertanyaan. Ada beberapa cara kita dapat mengekspresikan *conditional (*alias diskusi-deci-) dalam program kami. Yang paling umum adalah jika pernyataan. Pada dasarnya, Anda katakan-ing, " *Jika* Kondisi ini benar, lakukan hal berikut ...". Sebagai contoh:

```
var saldo bank = 302,13 ;
var jumlah = 99.99 ;

jika ( jumlah < saldo bank ) {
        menghibur . mencatat ( "Saya ingin membeli ponsel ini!" );
}
```

Itu jika Pernyataan membutuhkan ekspresi di antara tanda kurung () yang dapat diperlakukan sebagai baik benar atau Salah. Dalam program ini, kami menyediakan ekspresi jumlah <bank_balance, yang memang baik akan mengevaluasi untuk benar atau Salah, tergantung pada jumlah di saldo bank variabel.

Anda bahkan dapat memberikan alternatif jika kondisi tidak benar, disebut lain ayat.
Mempertimbangkan:

```
const ACCESSORY_PRICE = 9.99 ;

var saldo bank = 302,13 ;
var jumlah = 99.99 ;

jumlah = jumlah * 2 ;

// kita mampu pembelian tambahan?
jika ( jumlah < saldo bank ) {
        menghibur . mencatat ( "Aku akan mengambil aksesoris!" );
        jumlah = jumlah + ACCESSORY_PRICE ; }

// jika tidak:
lain {
        menghibur . mencatat ( "Tidak, terima kasih." );
}
```

Di sini, jika jumlah <bank_balance aku s benar, kami akan mencetak " Aku akan mengambil aksesoris!" dan tambahkan 9.99 untuk kami jumlah variabel. Lain-bijaksana, lain klausul mengatakan kita akan hanya sopan merespon dengan " Tidak, terima kasih." dan pergi jumlah tidak berubah. Sebagaimana kita bahas pada "Nilai & Jenis" pada halaman 10 , Nilai-nilai yang belum dari jenis yang diharapkan sering dipaksa untuk tipe tersebut. Itu jika

Pernyataan mengharapkan boolean, tetapi jika Anda lulus sesuatu yang belum boolean, pemaksaan akan terjadi.

JavaScript mendefinisikan daftar nilai-nilai tertentu yang dianggap "falsy" karena ketika dipaksa untuk boolean, mereka menjadi Salah -ini termasuk nilai-nilai seperti 0 dan "". Nilai lain tidak ada dalam daftar "falsy" secara otomatis "truthy" -ketika dipaksa untuk boolean mereka menjadi benar. nilai-nilai truthy mencakup hal-hal seperti 99.99 dan " bebas". Lihat "Mu Tru- & falsy" pada halaman 36 di Bab 2 untuk informasi lebih lanjut.

conditional ada dalam bentuk lain selain jika. Sebagai contoh, beralih Pernyataan dapat digunakan sebagai singkatan untuk serangkaian if..else pernyataan (lihat Bab 2). Loops (lihat "Loops" pada halaman 20) Gunakan *bersyarat* untuk menentukan apakah loop harus terus atau berhenti.

Untuk informasi lebih dalam tentang coercions yang dapat terjadi secara implisit dalam ekspresi uji *conditional,* lihat Bab 4 dari *Jenis & Gram mar* judul seri ini.

loops

Selama masa sibuk, ada daftar tunggu untuk pelanggan yang membutuhkan untuk berbicara dengan karyawan toko ponsel. Sementara masih ada orang dalam daftar itu, dia hanya perlu tetap melayani pelanggan berikutnya. Mengulang serangkaian tindakan sampai kondisi tertentu gagal-dengan kata lain, mengulangi hanya sementara kondisi memegang-adalah pekerjaan loop pro gramming; loop dapat mengambil bentuk yang berbeda, tetapi mereka semua memenuhi perilaku dasar ini.

Sebuah loop meliputi kondisi pengujian serta blok (biasanya sebagai {..}). Setiap kali blok lingkaran mengeksekusi, yang disebut *perulangan*.

Sebagai contoh, sementara loop dan do..while bentuk lingkaran menggambarkan konsep mengulang blok pernyataan sampai kondisi tidak lagi mengevaluasi benar:

```
sementara ( numOfCustomers > 0 ) {
    menghibur . mencatat ( "Bagaimana saya bisa membantu Anda?" );

    // membantu pelanggan ...

    numOfCustomers = numOfCustomers - 1 ; }


// melawan:

melakukan {
    menghibur . mencatat ( "Bagaimana saya bisa membantu Anda?" );

    // membantu pelanggan ...

    numOfCustomers = numOfCustomers - 1 ; } sementara ( numOfCustomers
> 0 );
```

Satu-satunya perbedaan praktis antara loop ini adalah apakah bersyarat diuji sebelum iterasi pertama (sementara) atau setelah iterasi pertama (do..while).

Baik dalam bentuk, jika tes kondisional sebagai Salah, iterasi berikutnya tidak akan berjalan. Itu berarti jika kondisi ini awalnya Salah, Sebuah sementara lingkaran tidak akan lari, tapi sebuah do..while lingkaran akan berjalan hanya pertama kali.

Kadang-kadang Anda perulangan untuk tujuan yang dimaksudkan menghitung satu set tertentu dari angka, seperti dari 0 untuk 9 (10 nomor). Anda dapat melakukannya dengan menetapkan loop iterasi variabel seperti saya pada nilai 0 dan incre- menting dengan 1 setiap iterasi.

Untuk berbagai alasan historis, bahasa pemrograman hampir selalu mengandalkan hal-hal secara berbasis nol, yang berarti dimulai dengan 0 dari pada 1. Jika Anda tidak terbiasa dengan itu cara berpikir, itu bisa sangat membingungkan pada awalnya. Luangkan waktu untuk berlatih menghitung dimulai dengan 0 untuk menjadi lebih nyaman dengan itu!

Kondisional diuji pada setiap iterasi, sebanyak jika ada tersirat jika Pernyataan di dalam lingkaran. Kita dapat menggunakan JavaScript ini istirahat Pernyataan untuk menghentikan lingkaran. Juga, kita dapat mengamati bahwa itu sangat mudah untuk membuat loop yang lain akan berjalan selamanya tanpa istirahat Mekanisme ing. Mari kita menggambarkan:

```
var saya = 0 ;

// sebuah `lingkaran while..true` akan berjalan selamanya, kan?
sementara ( benar ) {
        // terus loop pergi?
        jika ( saya <= 9 ) {
                menghibur . mencatat ( saya );
                saya = saya + 1 ; }

        // waktu untuk menghentikan loop!
        lain {
                istirahat ;
        }}

// 0 1 2 3 4 5 6 7 8 9
```

Ini tidak selalu bentuk praktis Anda ingin menggunakan untuk loop Anda. Ini yang disajikan di sini untuk tujuan ilustrasi saja.

sementara sementara (atau do..while) dapat menyelesaikan tugas secara manual, ada bentuk sintaksis lain yang disebut untuk loop untuk hanya tujuan itu:

```
untuk ( var saya = 0 ; saya <= 9 ; saya = saya + 1 ) {
    menghibur . mencatat ( saya );
}
// 0 1 2 3 4 5 6 7 8 9
```

Seperti yang Anda lihat, dalam kedua kasus kondisional i <= 9 aku s benar untuk 10 iterasi pertama (saya nilai-nilai 0 melalui 9) baik berupa lingkaran, tetapi menjadi Salah sekali saya adalah nilai 10.

Itu untuk lingkaran memiliki tiga klausa: klausa inisialisasi (var i = 0), bersyarat tes ayat (i <= 9), dan update ayat (i = i + 1). Jadi, jika Anda akan melakukan penghitungan dengan iterasi loop, untuk adalah bentuk yang lebih kompak dan sering lebih mudah untuk memahami dan menulis. Ada bentuk lingkaran lainnya khusus yang dimaksudkan untuk iterate atas nilai-nilai tertentu, seperti sifat-sifat suatu objek (lihat Bab 2) Di mana tes kondisional tersirat hanya apakah semua erties prop- telah diproses. The “loop sampai kondisi gagal” konsep memegang tidak peduli apa bentuk loop.

fungsi

The karyawan toko ponsel mungkin tidak membawa sekitar tor calcula- untuk mengetahui pajak dan jumlah pembelian akhir. Itu tugas dia perlu mendefinisikan sekali dan menggunakan kembali lagi dan lagi. Kemungkinannya, perusahaan memiliki checkout register (komputer, tablet, dll) dengan orang-orang “fungsi” yang dibangun di.

Demikian pula, program anda akan hampir pasti ingin memecah tugas kode ke dalam potongan dapat digunakan kembali, bukannya berulang kali mengulang sendiri tele (pun intended!). Cara untuk melakukan ini adalah untuk menentukan fungsi.

Fungsi umumnya merupakan bagian bernama kode yang dapat “disebut” dengan nama, dan kode di dalamnya akan dijalankan setiap kali. Mempertimbangkan:

```
fungsi printAmount () {
    menghibur . mencatat ( jumlah . toFixed ( 2 ) );
}

var jumlah = 99.99 ;

printAmount (); // " 99.99"
```

```
jumlah = jumlah * 2 ;

printAmount (); // " 199,98"
```

Fungsi opsional dapat mengambil argumen (alias parameter) -values Anda lulus dalam Dan mereka juga dapat opsional mengembalikan nilai kembali.:

```
fungsi printAmount ( amt ) {
    menghibur . mencatat ( amt . toFixed ( 2 ) );
}

fungsi formatAmount () {
    kembali "$" + jumlah . toFixed ( 2 );
}

var jumlah = 99.99 ;

printAmount ( jumlah * 2 );                    // "199,98"

jumlah = formatAmount ();
menghibur . mencatat ( jumlah );               // "$ 99,99"
```

fungsi printAmount (..) mengambil parameter yang kita sebut amt.

fungsi formatAmount () mengembalikan nilai. Tentu saja, Anda juga dapat menggabungkan dua teknik dalam fungsi yang sama. Fungsi yang sering digunakan untuk kode yang Anda berencana untuk memanggil beberapa kali, tetapi mereka juga dapat berguna hanya untuk mengatur bit terkait kode ke nama koleksi, bahkan jika Anda hanya berencana untuk memanggil mereka sekali. Mempertimbangkan:

```
const PERSENTASE PAJAK = 0,08 ;

fungsi calculateFinalPurchaseAmount ( amt ) {
    // menghitung jumlah baru dengan pajak
    amt = amt + ( amt * PERSENTASE PAJAK );

    // mengembalikan jumlah yang baru
    kembali amt ; }


var jumlah = 99.99 ;

jumlah = calculateFinalPurchaseAmount ( jumlah );

menghibur . mencatat ( jumlah . toFixed ( 2 ) );          // "107.99"
```

Meskipun calculateFinalPurchaseAmount (..) hanya dipanggil sekali, pengorganisasian perilaku menjadi fungsi bernama terpisah membuat kode yang menggunakan logika (yang Jumlah = calculateFinal ... negara-

ment) bersih. Jika fungsi memiliki lebih banyak pernyataan di dalamnya, manfaat akan lebih jelas.

Cakupan

Jika Anda meminta karyawan toko ponsel untuk model telepon yang tokonya tidak membawa, dia tidak akan dapat menjual telepon yang Anda inginkan. Dia hanya memiliki akses ke ponsel dalam persediaan tokonya. Anda harus mencoba toko lain untuk melihat apakah Anda dapat menemukan ponsel Anda cari.

Pemrograman memiliki istilah untuk konsep ini: *lingkup (* teknis disebut *lingkup leksikal).* Dalam JavaScript, masing-masing fungsi mendapat ruang lingkup sendiri. Ruang lingkup pada dasarnya adalah kumpulan dari variabel serta aturan untuk bagaimana variabel-variabel yang diakses oleh nama. Hanya kode di dalam fungsi yang dapat mengakses fungsi ini *scoped* variabel.

Sebuah nama variabel harus unik dalam yang sama lingkup-tidak mungkin ada dua yang berbeda Sebuah variabel duduk tepat di samping satu sama lain. Tapi nama variabel yang sama Sebuah bisa muncul dalam lingkup yang berbeda:

```
fungsi satu () {
    // ini `a` hanya milik` satu () `fungsi
    var Sebuah = 1 ;
    menghibur . mencatat ( Sebuah );
}

fungsi dua () {
    // ini `a` hanya milik` dua () `fungsi
    var Sebuah = 2 ;
    menghibur . mencatat ( Sebuah );
}

satu ();          // 1
dua ();           // 2
```

Juga, ruang lingkup dapat bersarang di dalam ruang lingkup lain, seperti jika badut di pesta ulang tahun meledak satu balon dalam balon lain. Jika salah satu ruang lingkup yang bersarang di dalam yang lain, kode di dalam ruang lingkup terdalam dapat mengakses variabel baik dari lingkup. Mempertimbangkan:

```
fungsi luar () {
    var Sebuah = 1 ;

    fungsi batin () {
        var b = 2 ;

        // kita bisa mengakses kedua `a` dan` B` disini
        menghibur . mencatat ( Sebuah + b ); // 3
    }

    batin ();

    // kita hanya bisa akses `A` disini
    menghibur . mencatat ( Sebuah );                    // 1
}

luar ();
```

aturan lingkup leksikal mengatakan bahwa kode dalam satu ruang lingkup dapat mengakses variabel baik lingkup yang atau ruang lingkup di luar itu. Jadi, kode di dalam batin() fungsi memiliki akses ke kedua variabel Sebuah dan b, tapi kode hanya dalam luar() memiliki akses hanya untuk Sebuah -itu tidak dapat mengakses b karena variabel yang hanya di dalam batin().

Ingat ini potongan kode dari sebelumnya:

```
const PERSENTASE PAJAK = 0,08 ;

fungsi calculateFinalPurchaseAmount ( amt ) {
    // menghitung jumlah baru dengan pajak
    amt = amt + ( amt * PERSENTASE PAJAK );

    // mengembalikan jumlah yang baru
    kembali amt ; }
```

Itu PERSENTASE PAJAK konstan (variabel) dapat diakses dari dalam CalculateFinalPurchaseAmount (..) fungsi, meskipun kita tidak lulus dalam, karena ruang lingkup leksikal.

Untuk informasi lebih lanjut tentang ruang lingkup leksikal, lihat tiga bab pertama dari *Lingkup & Penutup* judul seri ini.

Praktek

Sama sekali tidak ada pengganti untuk latihan dalam belajar Program- ming. Tidak ada jumlah menulis mengartikulasikan di pihak saya sendiri akan membuat Anda seorang programmer.

Dengan itu dalam pikiran, mari kita coba berlatih beberapa konsep yang kita pelajari di sini dalam bab ini. Saya akan memberikan "persyaratan," dan Anda mencobanya terlebih dahulu. Kemudian berkonsultasi kode listing di bawah ini untuk melihat bagaimana saya mendekatinya:

- Tulis program untuk menghitung total harga ponsel pur- mengejar. Anda akan tetap ponsel pembelian (petunjuk: loop) sampai Anda kehabisan uang di rekening bank Anda. Anda juga akan membeli accesso- Ries untuk setiap telepon selama jumlah pembelian Anda di bawah ambang batas pengeluaran mental Anda.

- Setelah dihitung jumlah pembelian Anda, menambahkan pajak, kemudian mencetak jumlah pembelian dihitung, benar untuk- kusut.

- Akhirnya, periksa jumlah terhadap saldo rekening bank Anda untuk melihat apakah Anda mampu membelinya atau tidak.
- Anda harus menyiapkan beberapa konstanta untuk "tarif pajak," "harga telepon," "harga aksesori," dan "belanja ambang batas," serta variabel untuk Anda "saldo rekening bank."
- Anda harus mendefinisikan fungsi untuk menghitung pajak dan untuk anyaman untuk-harga dengan "$" dan pembulatan ke dua desimal.

- **Bonus Tantangan:** Cobalah untuk menggabungkan masukan ke dalam program ini, mungkin dengan cepat(..) tercakup dalam "Input" pada halaman 6 . Anda mungkin meminta pengguna untuk saldo rekening bank mereka, untuk ple exam-. Bersenang-senang dan menjadi kreatif!

OK, pergi ke depan. Cobalah. Jangan mengintip kode listing saya sampai Anda telah diberikan tembakan sendiri!

Karena ini adalah buku JavaScript, saya jelas akan memecahkan latihan praktek dalam JavaScript. Tapi Anda bisa melakukannya dalam bahasa lain untuk saat ini jika Anda merasa lebih nyaman.

Berikut solusi JavaScript saya untuk latihan ini:

```
const SPENDING_THRESHOLD = 200 ;
const PERSENTASE PAJAK = 0,08 ;
const PHONE_PRICE = 99.99 ;
const ACCESSORY_PRICE = 9.99 ;

var saldo bank = 303,91 ;
var jumlah = 0 ;

fungsi calculateTax ( jumlah ) {
    kembali jumlah * PERSENTASE PAJAK ; }


fungsi formatAmount ( jumlah ) {
    kembali "$" + jumlah . toFixed ( 2 );
}

// tetap membeli ponsel sementara Anda masih memiliki uang
sementara ( jumlah < saldo bank ) {
    // membeli telepon baru!
    jumlah = jumlah + PHONE_PRICE ;

    // kita mampu aksesori?
    jika ( jumlah < SPENDING_THRESHOLD ) {
        jumlah = jumlah + ACCESSORY_PRICE ; }}



// jangan lupa untuk membayar pemerintah, juga
jumlah = jumlah + calculateTax ( jumlah );

menghibur . mencatat (
    "Pembelian Anda: " + formatAmount ( jumlah )
);
// Pembelian Anda: $ 334,76

// dapat Anda benar-benar mampu pembelian ini?
jika ( jumlah > saldo bank ) {
    menghibur . mencatat (
        "Anda tidak mampu pembelian ini. :("
    ); }

// Anda tidak mampu pembelian ini. :(
```

Cara paling mudah untuk menjalankan program JavaScript ini adalah untuk jenis ke dalam konsol pengembang browser terdekat Anda.

Bagaimana kamu melakukannya? Tidak ada salahnya untuk mencoba lagi sekarang bahwa Anda telah melihat kode saya. Dan bermain-main dengan mengubah beberapa stants con untuk melihat bagaimana program berjalan dengan nilai yang berbeda.

Ulasan

Belajar pemrograman tidak harus menjadi proses whelming kompleks dan berlebihan. Hanya ada beberapa konsep dasar yang Anda butuhkan untuk membungkus kepala Anda sekitar.

Ini bertindak seperti blok bangunan. Untuk membangun sebuah menara tinggi, Anda mulai pertama dengan menempatkan blok di atas blok di atas blok. Hal yang sama berlaku dengan pemrograman. Berikut adalah beberapa blok bangunan pemrograman penting:

- Anda perlu *operator* untuk melakukan tindakan pada.
- Anda perlu nilai-nilai dan *jenis* untuk melakukan berbagai jenis tindakan seperti matematika di jumlah s atau output dengan tali s.
- Anda perlu *variabel* untuk menyimpan data (alias *negara)* selama eksekusi pro gram Anda.

- Anda perlu *conditional* seperti jika pernyataan untuk membuat keputusan.
- Anda perlu *loop* mengulangi tugas sampai kondisi berhenti menjadi benar.
- Anda perlu *fungsi* untuk mengatur kode Anda ke logis dan reusa- potongan ble.

Kode komentar adalah salah satu cara yang efektif untuk menulis kode lebih mudah dibaca, yang membuat program Anda lebih mudah untuk memahami, menjaga, dan memperbaiki nanti jika ada masalah.

Akhirnya, jangan mengabaikan kekuatan praktek. Cara terbaik untuk mempelajari bagaimana menulis kode untuk menulis kode.

Saya senang Anda baik pada cara untuk belajar bagaimana kode, sekarang! Teruskan. Jangan lupa untuk memeriksa sumber pemrograman pemula lainnya (buku, blog, pelatihan online, dll). Bab ini dan buku ini adalah awal yang baik, tetapi mereka hanya pengenalan singkat. Bab berikutnya akan meninjau banyak konsep dari bab ini, tetapi dari perspektif yang lebih JavaScript spesifik, yang akan menyoroti sebagian besar topik utama yang dibahas secara rinci lebih dalam sepanjang sisa seri.

BAB 2

ke JavaScript

Dalam bab sebelumnya, saya memperkenalkan blok bangunan dasar pemrograman, seperti variabel, loop, conditional, dan fungsi. Tentu saja, semua kode yang ditunjukkan telah di JavaScript. Namun dalam bab ini, kami ingin fokus khusus pada hal-hal yang perlu Anda ketahui tentang JavaScript untuk bangun dan pergi sebagai pengembang JS. Kami akan memperkenalkan beberapa konsep dalam bab ini yang tidak akan sepenuhnya dieksplorasi sampai berikutnya *YDKJS* buku. Anda dapat menganggap bab ini sebagai gambaran dari topik yang dibahas secara rinci melalui-sisa seri ini.

Terutama jika Anda baru untuk JavaScript, Anda harus berharap untuk menghabiskan sedikit waktu meninjau konsep dan contoh kode di sini beberapa kali. Setiap dasar yang baik diletakkan bata demi bata, jadi jangan berharap bahwa Anda akan segera memahami itu semua lulus pertama melalui.

Perjalanan Anda ke dalam belajar JavaScript dimulai di sini.

Seperti yang saya katakan di Bab 1 , Anda pasti harus mencoba semua kode ini sendiri saat Anda membaca dan bekerja melalui bab ini. Sadarilah bahwa beberapa kode di sini mengasumsikan kemampuan diperkenalkan dalam versi terbaru dari JavaScript pada saat tulisan ini (sering disebut sebagai “ES6” untuk edisi 6 ECMAScript-nama resmi dari spesifikasi JS). Jika Anda kebetulan menggunakan lebih tua, pra-ES6 peramban, kode mungkin tidak bekerja. Sebuah update terbaru dari browser modern (seperti Chrome, Firefox, atau IE) harus digunakan.

Nilai & Jenis

Seperti kita menegaskan di Bab 1 , JavaScript telah diketik nilai-nilai, tidak diketik variabel. Berikut built-in jenis yang tersedia:

- tali
- jumlah
- boolean
- batal dan tidak terdefinisi
- obyek
- simbol (baru untuk ES6) JavaScript menyediakan jenis operator yang dapat memeriksa nilai

dan memberitahu Anda apa jenis itu:

```
var Sebuah ;
jenis Sebuah ;                  // "tidak terdefinisi"

Sebuah = "Halo Dunia" ;
jenis Sebuah ;                  // "string"

Sebuah = 42 ;
jenis Sebuah ;                  // "nomor"

Sebuah = benar ;
jenis Sebuah ;                  // "boolean"

Sebuah = batal ;
jenis Sebuah ;                  // "objek" - aneh, bug

Sebuah = tidak terdefinisi ;
jenis Sebuah ;                  // "tidak terdefinisi"
```

```
Sebuah = { b : "C" };
jenis Sebuah ;                  // "objek"
```

Nilai kembali dari jenis Operator selalu salah satu dari enam (tujuh per ES6!) nilai-nilai string. Itu adalah, typeof "abc" pengembalian "tali", tidak tali.

Perhatikan bagaimana dalam potongan yang Sebuah variabel memegang setiap jenis yang berbeda dari nilai, dan bahwa meskipun penampilan, typeof sebuah tidak meminta untuk "jenis Sebuah, "Melainkan untuk"jenis nilai saat ini di Sebuah. "Hanya nilai-nilai memiliki jenis dalam JavaScript; variabel tainers con- hanya sederhana untuk nilai-nilai.

typeof nol merupakan kasus yang menarik karena errantly kembali "obyek" ketika Anda harapkan untuk kembali " batal".

Ini adalah bug lama di JS, tapi salah satu yang mungkin tidak akan pernah diperbaiki. Terlalu banyak kode di Web bergantung pada bug, dan dengan demikian memperbaiki itu akan menyebabkan lebih banyak bug!

Juga mencatat a = tidak terdefinisi. Kami secara eksplisit menetapkan Sebuah ke tidak terdefinisi nilai, tapi itu perilaku tidak berbeda dari variabel yang tidak memiliki nilai yang ditetapkan belum, seperti dengan var a; baris di bagian atas hewan peliharaan snip-. Sebuah variabel dapat mendapatkan ini negara nilai "undefined" dalam beberapa cara ferent dif-, termasuk fungsi yang kembali tidak ada nilai-nilai dan penggunaan kosong operator.

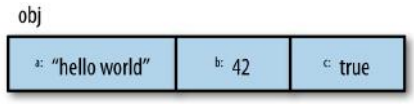
benda

Itu obyek tipe mengacu pada nilai senyawa di mana Anda dapat mengatur properti (bernama lokasi) yang masing-masing memegang nilai-nilai mereka sendiri dari jenis apa pun. Ini mungkin salah satu jenis nilai yang paling berguna dalam semua Jawa-Script:

```
var obj = {
    Sebuah : "Halo Dunia" .
    b : 42 .
    c : benar
};

obj . Sebuah ;      // "Halo Dunia"
obj . b ;           // 42
obj . c ;           // benar
```

```
obj [ "Sebuah" ]; // " Halo Dunia"
obj [ "B" ]; // 42
obj [ "C" ]; // benar
```

Ini mungkin membantu untuk memikirkan ini obj nilai visual:



Sifat baik dapat diakses dengan *notasi titik (* yaitu, obj.a) atau *notasi braket (* yaitu, obj ["a"]). notasi dot yang lebih pendek dan Star Excursion Balance Test sekutu mudah dibaca, dan dengan demikian lebih disukai bila memungkinkan. notasi braket berguna jika Anda memiliki nama properti yang telah spesifiknya karakter resmi di dalamnya, seperti obj ["hello world!"] sifat -seperti yang sering disebut sebagai *kunci-kunci* ketika diakses melalui notasi braket. The [] notasi membutuhkan baik variabel (dijelaskan berikutnya) atau tali *eral lit- (* yang perlu dibungkus " .. " atau ' .. '). Tentu saja, notasi braket juga berguna jika Anda ingin mengakses properti / kunci tetapi nama disimpan dalam variabel lain, seperti:

```
var obj = {
    Sebuah : "Halo Dunia" .
    b : 42
};

var b = "Sebuah" ;

obj [ b ];          // "Halo Dunia"
obj [ "B" ];        // 42
```

Untuk informasi lebih lanjut tentang JavaScript obyek s, lihat *ini & Prototip Obyek* judul seri ini, khususnya Bab 3.

Ada beberapa jenis nilai lain yang Anda umumnya akan berinteraksi dengan program JavaScript: *susunan* dan *fungsi.* Tapi bukannya tepat built-in jenis, ini harus dianggap lebih seperti versi subtipe-khusus dari obyek mengetik.

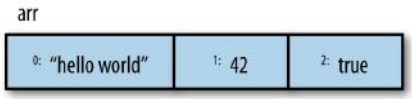
array

Array adalah obyek yang memegang nilai-nilai (jenis apapun) tidak khususnya di properti bernama / kunci, melainkan dalam tions posi- numerik diindeks. Sebagai contoh:

```
var arr = [
    "Halo Dunia" .
    42 .
    benar
];

arr [ 0 ];              // "Halo Dunia"
arr [ 1 ];              // 42
arr [ 2 ];              // benar
arr . panjangnya ;      // 3

jenis arr ;             // "objek"
```

Bahasa yang mulai menghitung nol, seperti JS tidak, penggunaan 0 sebagai indeks dari elemen pertama dalam array.

Ini mungkin membantu untuk memikirkan arr visual:



Karena array adalah objek khusus (sebagai jenis menyiratkan), mereka juga dapat memiliki sifat, termasuk otomatis diperbarui panjangnya erty prop-.

Anda secara teoritis bisa menggunakan array sebagai objek normal dengan nama properti Anda sendiri, atau Anda bisa menggunakan obyek tetapi hanya memberikan sifat numerik (0, 1, dll) mirip dengan array. Namun, ini umumnya akan dianggap penggunaan yang tidak tepat dari jenis masing-masing.

Pendekatan terbaik dan paling alami adalah dengan menggunakan array untuk nilai-nilai numerik diposisikan dan penggunaan obyek s untuk bernama properti.

fungsi

Yang lain obyek subtipe Anda akan menggunakan seluruh program JS Anda adalah fungsi:

```
fungsi foo () {
    kembali 42 ; }

foo . bar = "Halo Dunia" ;

jenis foo ;                 // "fungsi"
jenis foo ();               // "nomor"
jenis foo . bar ;           // "string"
```

Sekali lagi, fungsi adalah subtipe dari benda - jenis kembali " func tion", yang menyiratkan bahwa fungsi adalah utama jenis-dan dengan demikian dapat memiliki sifat, tetapi Anda biasanya hanya akan menggunakan properti fungsi objek (seperti foo.bar) dalam kasus-kasus yang terbatas.

Untuk informasi lebih lanjut tentang nilai-nilai JS dan jenis mereka, melihat dua bab pertama dari *Jenis & Grammar* judul seri ini.

Built-In Jenis Metode

Built-in jenis dan subtipe baru saja kita bahas memiliki perilaku terkena sebagai properti dan metode yang cukup kuat dan digunakan-ful.

Sebagai contoh:

```
var Sebuah = "Halo Dunia" ;
var b = 3.14159 ;

Sebuah . panjangnya ;           // 11
Sebuah . toUpperCase ();        // "HALO DUNIA"
b . toFixed ( 4 );              // "3,1416"
```

"Bagaimana" belakang bisa memanggil a.toUpperCase () lebih com- plicated dari itu metode yang ada pada nilai. Secara singkat, ada String (modal S) objek bentuk pembungkus, biasanya disebut "asli," bahwa pasangan dengan primitif tali mengetik; itu ini wrapper objek yang mendefinisikan toUpperCase () Metode pada jenis proto-nya.

Ketika Anda menggunakan nilai primitif seperti " Halo Dunia" sebagai obyek dengan referensi properti atau metode (misalnya, a.toUpperCase () di pra vious snippet), JS otomatis “kotak” nilai untuk objeknya wrap per rekan (tersembunyi di bawah selimut). SEBUAH tali nilai dapat dibungkus oleh Tali objek, jumlah dapat dibungkus oleh Jumlah objek, dan boolean dapat dibungkus oleh

boolean obyek. Untuk sebagian besar, Anda tidak perlu khawatir tentang atau langsung menggunakan bentuk objek wrapper ini dari nilai-nilai-lebih memilih bentuk nilai primitif di hampir semua kasus dan JavaScript akan mengurus sisanya untuk Anda.

Untuk informasi lebih lanjut tentang pribumi JS dan “ing kotak-,” lihat Bab 3 dari *Jenis & Grammar* judul seri ini. Untuk lebih memahami prototipe dari sebuah objek, lihat Bab 5 dari *ini & Prototip Obyek* judul seri ini.

membandingkan Nilai

Ada dua jenis utama dari perbandingan nilai yang akan Anda butuhkan untuk membuat program JS Anda: *persamaan* dan *ketidaksamaan.* Hasil perbandingan apapun adalah ketat boolean nilai (benar atau Salah), terlepas dari apa jenis nilai dibandingkan.

Paksaan

Kami berbicara secara singkat tentang pemaksaan di Bab 1 , Tapi mari kita kembali di sini. Pemaksaan datang dalam dua bentuk dalam JavaScript: *eksplisit* dan *implisit.*

pemaksaan eksplisit hanyalah bahwa Anda dapat melihat dari kode yang konversi dari satu jenis yang lain akan terjadi, sedangkan pemaksaan implisit adalah ketika jenis konversi dapat terjadi sebagai lebih dari efek samping yang jelas non beberapa operasi lainnya.

Anda mungkin pernah mendengar sentimen seperti “pemaksaan adalah jahat” diambil dari fakta bahwa ada jelas tempat di mana paksaan dapat menghasilkan beberapa hasil yang mengejutkan. Mungkin tidak ada membangkitkan frustrasi dari pengembang lebih daripada ketika bahasa mengejutkan mereka. Pemaksaan tidak jahat, juga tidak harus mengejutkan. Bahkan, sebagian besar kasus, Anda dapat membangun dengan jenis paksaan cukup masuk akal dan dapat dipahami, dan bahkan dapat digunakan untuk *memperbaiki* pembacaan kode Anda. Tapi kita tidak akan pergi lebih jauh ke dalam

Perdebatan-Bab 4 dari *Jenis & Grammar* judul seri ini mencakup semua sisi.

Berikut adalah contoh *eksplisit* paksaan:

```
var Sebuah = "42" ;

var b = Jumlah ( Sebuah );

Sebuah ;                    // "42"
b ;                         // 42 - nomor!
```

Dan inilah contoh dari *implisit* paksaan:

```
var Sebuah = "42" ;

var b = Sebuah * 1 ; // " 42" secara implisit dipaksa untuk 42 disini

Sebuah ;                    // "42"
b ;                         // 42 - nomor!
```

Truthy & falsy

Di Bab 1 , Kita secara singkat disebutkan sifat “truthy” dan “falsy” nilai-nilai: ketika non boolean nilai yang dipaksa untuk boolean, apakah itu menjadi benar atau Salah, masing-masing?

Daftar spesifik nilai-nilai “falsy” dalam JavaScript adalah sebagai berikut:

- "" (string kosong)
- 0, -0, NaN (cacat jumlah)
- batal, tidak terdefinisi
- Salah

Setiap nilai yang tidak pada daftar ini “falsy” adalah “truthy.” Berikut adalah beberapa contoh dari mereka:

- "Halo"
- 42
- benar
- [], [1, "2", 3] (array)
- {}, {A: 42} (benda)
- fungsi foo () {..} (fungsi)

Sangat penting untuk diingat bahwa non boolean nilai hanya mengikuti ini “truthy” / “falsy” pemaksaan jika itu benar-benar dipaksa untuk boolean. Ini tidak semua yang sulit untuk membingungkan diri dengan situasi yang tampaknya seperti itu memaksa nilai ke boolean ketika itu tidak.

Persamaan

Ada empat operator kesetaraan: ==, ===, =, dan ==!!. The! bentuk yang tentu saja simetris “tidak sama” versi bagian kontra mereka; *non-kesetaraan* tidak harus bingung dengan *ketidaksamaan.*

Perbedaan antara == dan === biasanya ditandai bahwa == cek untuk nilai kesetaraan dan === pemeriksaan untuk kedua nilai dan ketik kesetaraan. Namun, ini tidak akurat. Cara yang tepat untuk menggambarkan mereka adalah bahwa cek == untuk nilai kesetaraan dengan pemaksaan diperbolehkan, dan
=== pemeriksaan untuk nilai kesetaraan tanpa membiarkan paksaan; === sering disebut “kesetaraan ketat” untuk alasan ini.

Pertimbangkan paksaan implisit yang diizinkan oleh == loose- perbandingan kesetaraan dan tidak diperbolehkan dengan === ketat-kesetaraan:

```
var Sebuah = "42" ;
var b = 42 ;

Sebuah == b ;          // benar
Sebuah === b ;         // Salah
```

Dalam a == b perbandingan, JS pemberitahuan bahwa jenis tidak cocok, sehingga ia pergi melalui serangkaian memerintahkan langkah-langkah untuk memaksa salah satu atau kedua val- UES untuk jenis yang berbeda sampai jenis cocok, di mana kemudian nilai kesetaraan sederhana dapat diperiksa.

Jika Anda berpikir tentang hal itu, ada dua cara yang mungkin a == b bisa memberikan benar melalui pemaksaan. Entah perbandingan bisa berakhir sebagai 42 == 42 atau bisa juga " 42" == "42". Jadi mana yang benar? Jawabannya: " 42" menjadi 42, untuk membuat perbandingan 42 == 42. Dalam sebuah contoh sederhana, itu tidak benar-benar tampaknya peduli jalan mana yang proses yang berlangsung, sebagai hasil akhirnya adalah sama. Ada kasus plex lebih com- mana itu penting bukan hanya apa hasil akhir dari parison com-, tetapi *bagaimana* Anda sampai di sana. Itu a === b menghasilkan Salah, karena pemaksaan tidak diperbolehkan, sehingga nilai perbandingan sederhana jelas gagal. Banyak pengembang merasa bahwa === lebih diprediksi, sehingga mereka menganjurkan selalu menggunakan formulir itu dan tinggal jauh dari ==. Saya pikir pandangan ini sangat picik. saya

percaya == adalah alat yang ampuh yang membantu program Anda, *jika Anda meluangkan waktu untuk mempelajari cara kerjanya.*

Kami tidak akan menutup semua detail seluk-beluk bagaimana cion coer- dalam perbandingan == bekerja di sini. Banyak dari itu cukup masuk akal, tetapi ada beberapa kasus sudut penting untuk berhati-hati. Anda dapat membaca bagian 11.9.3 dari ES5 spesifikasi untuk melihat aturan yang tepat, dan Anda akan terkejut melihat betapa sederhana mekanisme ini, dibandingkan dengan semua hype negatif sekitarnya.

Sampai mendidih seluruh banyak rincian untuk takeaways sederhana, dan membantu Anda mengetahui apakah akan menggunakan == atau === dalam berbagai situasi, berikut adalah aturan sederhana saya:

- Jika salah satu nilai (sisi alias) dibandingkan bisa menjadi benar atau Salah nilai, menghindari == dan menggunakan ===.
- Jika salah satu nilai dalam perbandingan bisa menjadi nilai-nilai tertentu (0, "", atau [] -empty array), hindari == dan menggunakan ===.
- Di *semua* kasus lain, Anda aman untuk digunakan ==. Tidak hanya aman, tetapi dalam banyak kasus menyederhanakan kode Anda dengan cara yang meningkatkan kemampuan read.

Apa aturan ini mendidih ke yang membutuhkan Anda untuk berpikir kritis tentang kode Anda dan tentang apa jenis nilai dapat datang melalui variabel yang bisa dibandingkan untuk kesetaraan. Jika Anda bisa yakin tentang nilai-nilai, dan == aman, menggunakannya! Jika Anda tidak bisa yakin tentang ues val-, gunakan ===. Ini sederhana itu.

The! = Non-kesetaraan bentuk berpasangan dengan ==, dan! == bentuk berpasangan dengan ===. Semua aturan dan pengamatan saja kita bahas terus Cally symmetri- untuk perbandingan non-kesetaraan ini.

Anda harus mengambil catatan khusus dari == dan === aturan perbandingan jika Anda membandingkan dua nilai non-primitif, seperti obyek s (termasuk fungsi dan array). Karena nilai-nilai yang benar-benar dipegang oleh selisih ref-, baik == dan === perbandingan hanya akan memeriksa apakah referensi sesuai, tidak apa-apa tentang nilai-nilai yang mendasari. Sebagai contoh, susunan s secara default dipaksa untuk tali s dengan hanya bergabung semua nilai dengan koma (,) di antara. Anda mungkin berpikir bahwa dua susunan s dengan isi yang sama akan == sama, tetapi mereka tidak:

```
var Sebuah = [ 1 . 2 . 3 ];
var b = [ 1 . 2 . 3 ];
var c = "1,2,3" ;

Sebuah == c ;        // benar
b == c ;             // benar
Sebuah == b ;        // Salah
```

Untuk informasi lebih lanjut tentang aturan perbandingan == kesetaraan, lihat ES5 spesifikasi (sek- tion 11.9.3) dan juga berkonsultasi Bab 4 dari *Jenis & Grammar* judul seri ini; lihat Bab 2 untuk informasi lebih lanjut tentang nilai-nilai dibandingkan perbedaan-perbedaan ref-.

Ketidaksamaan

The <,>, <=, dan> = operator digunakan untuk ketidaksetaraan, sebagaimana dimaksud dalam spesifikasi sebagai "perbandingan relasional." Biasanya mereka akan digunakan dengan nilai-nilai ordinally sebanding seperti jumlah s. Sangat mudah untuk memahami bahwa 3 <4.

tapi JavaScript tali nilai-nilai juga dapat dibandingkan untuk ketidaksetaraan, menggunakan aturan abjad khas (" bar"< "foo").

Bagaimana pemaksaan? aturan serupa sebagai pembanding == (meskipun tidak persis sama!) berlaku untuk operator ketidaksetaraan. Terutama, tidak ada "ketimpangan yang ketat" operator yang akan melarang pemaksaan dengan cara yang sama === "kesetaraan yang ketat" tidak. Mempertimbangkan:

```
var Sebuah = 41 ;
var b = "42" ;
var c = "43" ;

Sebuah < b ;         // benar
b < c ;              // benar
```

Apa yang terjadi di sini? Pada bagian 11.8.5 dari spesifikasi ES5, ia mengatakan bahwa jika kedua nilai-nilai dalam <perbandingan yang tali s, karena dengan b < c, perbandingan dibuat leksikografis (alias abjad seperti kamus). Tetapi jika salah satu atau kedua tidak tali, karena dengan <b, maka kedua nilai-nilai yang dipaksa untuk menjadi jumlah s, dan perbandingan numerik yang khas terjadi.

Gotcha terbesar Anda mungkin mengalami di sini dengan perbandingan antara nilai potensial yang berbeda jenis-ingat, tidak ada "ketimpangan yang ketat" bentuk untuk digunakan-adalah ketika salah satu nilai tidak dapat dibuat menjadi angka yang benar, seperti:

```
var Sebuah = 42;
var b = "Foo";

Sebuah < b;     // Salah
Sebuah > b;     // Salah
Sebuah == b;    // Salah
```

Tunggu, bagaimana ketiga perbandingan tersebut dapat Salah? Karena b nilai sedang dipaksa untuk "tidak valid nilai angka" NaN di <dan> perbandingan, dan spesifikasi mengatakan bahwa NaN adalah tidak lebih besar dari atau kurang dari nilai lain. Perbandingan == gagal untuk alasan yang berbeda. a == b bisa gagal jika itu diartikan baik sebagai 42 == NaN atau " 42" == "foo" -seperti yang kita dijelaskan sebelumnya, yang pertama adalah kasus.

Untuk informasi lebih lanjut tentang ketidaksamaan com- aturan parison, lihat bagian 11.8.5 dari ES5 speci- fikasi dan juga berkonsultasi Bab 4 dari *Jenis & Grammar* judul seri ini.

variabel

Dalam JavaScript, nama variabel (termasuk nama fungsi) harus valid *pengidentifikasi*. Aturan ketat dan lengkap untuk karakter yang valid dalam pengidentifikasi adalah sedikit rumit ketika Anda mempertimbangkan karakter non-tradisional seperti Unicode. Jika Anda hanya mempertimbangkan karakter ASCII alfanumerik khas, meskipun, aturan sederhana. Sebuah identifier harus dimulai dengan az, AZ, $, atau _. Kemudian dapat berisi karakter-karakter ditambah angka 0-9.

Umumnya, aturan yang sama berlaku untuk nama properti untuk pengenal variabel. Namun, kata-kata tertentu tidak dapat digunakan sebagai variabel, tetapi OK sebagai nama properti. Kata-kata ini disebut "kata reserved," dan memasukkan kata kunci JS (untuk, di, jika, dll) serta null, benar, dan Salah.

Untuk informasi lebih lanjut tentang kata-kata dicadangkan, lihat Lampiran A dari *Jenis & Grammar* judul seri ini.

fungsi Lingkup

Anda menggunakan var kata kunci untuk mendeklarasikan variabel yang akan termasuk dalam ruang lingkup fungsi saat ini, atau lingkup global jika di tingkat atas di luar fungsi apapun.

mengangkat

dimanapun var muncul di dalam lingkup, deklarasi yang diambil milik seluruh lingkup dan dapat diakses di mana-mana di seluruh. Metaforis, perilaku ini disebut *mengangkat,* Ketika sebuah var tion declara- secara konseptual "pindah" ke bagian atas ruang lingkup melampirkan. Tech- nically, proses ini lebih akurat dijelaskan oleh bagaimana kode dikompilasi, tapi kita bisa melewatkan rincian mereka untuk saat ini. Mempertimbangkan:

```
var Sebuah = 2 ;

foo ();                         // bekerja karena `foo ()` // deklarasi
                                "dikibarkan"

fungsi foo () {
    Sebuah = 3 ;

    menghibur . mencatat ( Sebuah );    // 3

    var Sebuah ;                // deklarasi "mengangkat" // ke puncak
                                `foo ()`
}

menghibur . mencatat ( Sebuah ); // 2
```

Ini tidak umum atau ide yang baik untuk mengandalkan variabel- mampu *hoisting* menggunakan variabel sebelumnya dalam lingkup dibandingkan var deklarasi muncul; itu bisa sangat membingungkan.

Ini jauh lebih umum dan diterima untuk menggunakan *dikibarkan* deklarasi fungsi, seperti yang kita lakukan dengan foo () sebut muncul sebelum deklarasi formal.

cakupan bersarang

Ketika Anda mendeklarasikan variabel, tersedia di mana saja di lingkup itu, serta lebih rendah / lingkup batin. Sebagai contoh:

```
fungsi foo () {
    var Sebuah = 1 ;

    fungsi bar () {
        var b = 2 ;

        fungsi baz () {
            var c = 3 ;

            menghibur . mencatat ( Sebuah . b . c ); // 1 2 3
        }

        baz ();
        menghibur . mencatat ( Sebuah . b );                // 1 2
    }

    bar ();
    menghibur . mencatat ( Sebuah );                        // 1
}

foo ();
```

Perhatikan itu c tidak tersedia dalam bar(), karena itu menyatakan hanya di dalam batin baz () ruang lingkup, dan bahwa b tidak tersedia untuk foo () untuk alasan yang sama.

Jika Anda mencoba untuk mengakses nilai variabel dalam lingkup di mana itu tidak avail- mampu, Anda akan mendapatkan ReferenceError dilemparkan. Jika Anda mencoba untuk menetapkan variabel yang belum dideklarasikan, Anda akan baik berakhir menciptakan sebuah variabel dalam lingkup global tingkat atas (buruk!) Atau mendapatkan kesalahan, tergantung pada "modus ketat" (lihat "Ketat Mode" pada halaman 45). Mari lihat:

```
fungsi foo () {
    Sebuah = 1 ; // ` a` tidak secara resmi menyatakan
}

foo ();
Sebuah ;            // 1 - oops, auto variabel global :(
```

Ini adalah praktek yang sangat buruk. Jangan lakukan itu! Selalu resmi mendeklarasikan variabel Anda.

Selain menciptakan deklarasi untuk variabel di tingkat fungsi, ES6 *memungkinkan* Anda mendeklarasikan variabel milik blok individu (pasang { .. }), menggunakan membiarkan kata kunci. Selain beberapa bernuansa

detail, aturan scoping akan berperilaku kira-kira sama seperti yang kita hanya melihat dengan fungsi:

```
fungsi foo () {
    var Sebuah = 1 ;

    jika ( Sebuah > = 1 ) {
        membiarkan b = 2 ;

        sementara ( b < 5 ) {
            membiarkan c = b * 2 ;
            b ++ ;

            menghibur . mencatat ( Sebuah + c );
        }}}

foo ();
// 5 7 9
```

Karena menggunakan membiarkan dari pada var, b akan hanya milik jika Pernyataan dan dengan demikian tidak seluruh foo () Fungsi ini lingkup. Iarly Serupa, c hanya milik sementara lingkaran. Blok scoping sangat berguna untuk mengelola lingkup variabel Anda dengan cara yang lebih halus, yang dapat membuat kode Anda lebih mudah untuk mempertahankan dari waktu ke waktu.

Untuk informasi lebih lanjut tentang ruang lingkup, lihat *Lingkup & Penutup* judul seri ini. Lihat *ES6 & Beyond* judul seri ini untuk informasi lebih lanjut tentang membiarkan blok scoping.

conditional

Selain jika Pernyataan kami memperkenalkan secara singkat di Bab 1 , JavaScript menyediakan beberapa mekanisme conditional lain bahwa kita harus melihat pada.

Kadang-kadang Anda mungkin menemukan diri Anda menulis serangkaian if..else..if pernyataan seperti ini:

```
jika ( Sebuah == 2 ) {
    // lakukan sesuatu
}
lain jika ( Sebuah == 10 ) {
    // melakukan hal lain
}
```

```
lain jika ( Sebuah == 42 ) {
    // lakukan belum hal lain
}
lain {
    // mundur ke sini
}
```

Struktur ini bekerja, tapi itu sedikit verbose karena Anda perlu menentukan Sebuah tes untuk setiap kasus. Berikut pilihan lain, yang beralih pernyataan:

```
beralih ( Sebuah ) {
    kasus 2 :
        // lakukan sesuatu
        istirahat ;
    kasus 10 :
        // melakukan hal lain
        istirahat ;
    kasus 42 :
        // lakukan belum hal lain
        istirahat ;
    kegagalan :
        // mundur ke sini
}
```

Itu istirahat adalah penting jika Anda ingin hanya pernyataan (s) dalam satu kasus untuk berlari. Jika Anda menghilangkan istirahat dari kasus, dan itu kasus pertandingan atau berjalan, eksekusi akan dilanjutkan dengan berikutnya kasus 'Laporan terlepas dari itu kasus sesuai. Ini disebut “jatuh melalui” kadang-kadang berguna / diinginkan:

```
beralih ( Sebuah ) {
    kasus 2 :
    kasus 10 :
        // beberapa hal keren
        istirahat ;
    kasus 42 :
        // hal-hal lain
        istirahat ;
    kegagalan :
        // fallback
}
```

Di sini, jika Sebuah adalah baik 2 atau 10, itu akan mengeksekusi “beberapa hal keren” pernyataan kode.

Bentuk lain dari kondisional dalam JavaScript adalah “operator kondisional,” sering disebut “operator ternary.” Ini seperti bentuk yang lebih ringkas dari satu if..else pernyataan, seperti:

```
var Sebuah = 42 ;

var b = ( Sebuah > 41 ) ? "Halo" : "dunia" ;

// mirip dengan:

// jika (a> 41) {//
        b = "hello"; // } //
lain { //

        b = "dunia"; //}
```

Jika tes ekspresi (a> 41 di sini) mengevaluasi sebagai benar, klausa pertama (" Halo") hasil; jika tidak, klausa kedua (" dunia") hasil, dan apa pun hasilnya kemudian akan ditugaskan untuk b.

Operator kondisional tidak harus digunakan dalam tugas, tapi itu pasti penggunaan yang paling umum.

Untuk informasi lebih lanjut tentang kondisi pengujian dan pola lainnya untuk beralih dan? :, Lihat *Jenis & Grammar* judul seri ini.

Modus yang ketat

ES5 menambahkan “modus ketat” untuk bahasa, yang mengencangkan aturan untuk perilaku tertentu. Umumnya, pembatasan ini dilihat sebagai keep- ing kode untuk satu set yang lebih aman dan lebih tepat pedoman. Juga, mengikuti modus yang ketat membuat kode Anda umumnya lebih optimiza- ble oleh mesin. Modus yang ketat adalah kemenangan besar untuk kode, dan Anda harus menggunakannya untuk semua program Anda.

Anda dapat ikut serta dalam mode ketat untuk fungsi individu, atau seluruh file, tergantung di mana Anda meletakkan mode pragma yang ketat:

```
fungsi foo () {
    "Menggunakan ketat" ;

    // kode ini adalah modus ketat

    fungsi bar () {
        // kode ini adalah modus ketat
    }}
```

```
// kode ini tidak modus ketat
```

Bandingkan dengan:

```
"Menggunakan ketat" ;

fungsi foo () {
    // kode ini adalah modus ketat

    fungsi bar () {
        // kode ini adalah modus ketat
    }}


// kode ini adalah modus ketat
```

Salah satu perbedaan utama (perbaikan!) Dengan modus ketat melarang implisit auto-variabel global deklarasi dari menghilangkan var:

```
fungsi foo () {
    "Menggunakan ketat" ; // mengaktifkan mode ketat
    Sebuah = 1 ;                // `var` hilang, ReferenceError
}

foo ();
```

Jika Anda mengaktifkan modus yang ketat dalam kode Anda, dan Anda mendapatkan error, atau kode mulai berperilaku kereta, godaan Anda mungkin untuk menghindari modus ketat. Tapi itu naluri akan menjadi ide yang buruk untuk memanjakan. Jika modus ketat menyebabkan masalah dalam program Anda, itu hampir pasti tanda bahwa Anda memiliki hal-hal dalam program Anda, Anda harus memperbaiki.

Tidak hanya akan modus ketat menjaga kode Anda ke jalur yang lebih aman, dan tidak hanya akan membuat kode Anda lebih optimizable, tetapi juga mewakili arah masa depan bahasa. Akan lebih mudah pada Anda untuk mendapatkan digunakan untuk modus ketat sekarang daripada terus menundanya-itu akan hanya mendapatkan lebih sulit untuk mengkonversi nanti!

Untuk informasi lebih lanjut tentang mode ketat, lihat Bab 5 dari *Jenis & Grammar* judul seri ini.

Berfungsi sebagai Nilai

Sejauh ini, kita telah membahas berfungsi sebagai mekanisme utama *cakupan*
dalam JavaScript. Anda ingat khas fungsi sintaks deklarasi sebagai posisi terendah fol-:

```
fungsi foo () {
    // ..
}
```

Meskipun mungkin tidak tampak jelas dari sintaks itu, foo pada dasarnya hanya variabel dalam lingkup melampirkan luar yang diberikan referensi ke fungsi dinyatakan. Itu adalah fungsi itu sendiri adalah nilai, seperti 42 atau [1,2,3] akan menjadi.

Hal ini mungkin terdengar seperti sebuah konsep yang aneh pada awalnya, jadi luangkan waktu untuk merenungkan hal itu. Tidak hanya dapat Anda melewati nilai (argumen) *untuk* fungsi, tapi *fungsi itu sendiri bisa menjadi nilai* yang ditugaskan untuk variabel atau melewati atau kembali dari fungsi lainnya.

Dengan demikian, nilai fungsi harus dianggap sebagai ekspresi, seperti nilai lain atau ekspresi. Mempertimbangkan:

```
var foo = fungsi () {
    // ..
};

var x = fungsi bar () {
    // ..
};
```

Pertama ekspresi fungsi yang ditetapkan ke foo variabel disebut *anonim* karena tidak memiliki nama.

Ekspresi Fungsi kedua adalah *bernama (* bar), bahkan sebagai referensi untuk itu juga ditugaskan untuk x variabel. *ekspresi fungsi bernama* umumnya lebih disukai, meskipun *ekspresi fungsi anonim*
masih sangat umum. Untuk informasi lebih lanjut, lihat *Lingkup & Penutup* judul seri ini.

Segera Dipanggil Fungsi Ekspresi (IIFEs)

Dalam potongan sebelumnya, tak satu pun dari ekspresi fungsi yang exe- cuted-kita bisa jika kita termasuk foo () atau x (), contohnya.

Ada cara lain untuk mengeksekusi ekspresi fungsi, yang merupakan typi- Cally disebut sebagai *segera dipanggil ekspresi fungsi* (life):

```
( fungsi life () {
        menghibur . mencatat ( "Halo!" );
}) ();
// "Hello!"
```

The luar (..) Yang mengelilingi (Fungsi life () {..})
ekspresi fungsi hanya nuansa JS tata bahasa yang diperlukan untuk mencegah agar dari yang diperlakukan sebagai fungsi deklarasi normal. Final () pada akhir ekspresi-the}) (); line-adalah apa yang sebenarnya mengeksekusi ekspresi fungsi dirujuk segera sebelum.

Yang mungkin tampak aneh, tapi itu tidak asing sebagai pandangan pertama. Con Sider kesamaan antara foo dan life sini:

```
fungsi foo () { ..}

// ekspresi `referensi fungsi foo`, // maka` ()
`mengeksekusinya
foo ();

// `IIFE` fungsi ekspresi, // maka` ()
`mengeksekusinya
( fungsi life () { ..}) ();
```

Seperti yang Anda lihat, daftar (Fungsi life () {..}) sebelum cuting exe- nya () pada dasarnya sama dengan termasuk foo sebelum ing execut- nya (); dalam kedua kasus, referensi fungsi dijalankan dengan () segera setelah.

Karena sebuah life hanya fungsi, dan fungsi membuat variabel *cakupan,* menggunakan life dengan cara ini sering digunakan untuk mendeklarasikan variabel yang tidak akan mempengaruhi kode sekitarnya luar life yang:

```
var Sebuah = 42 ;

( fungsi life () {
        var Sebuah = 10 ;
        menghibur . mencatat ( Sebuah ); // 10
}) ();

menghibur . mencatat ( Sebuah );              // 42
```

IIFEs juga dapat memiliki kembali nilai-nilai:

```
var x = ( fungsi life () {
        kembali 42 ; })
();

x ; // 42
```

Itu 42 nilai mendapat kembali ed dari IIFE- bernama fungsi dieksekusi, dan kemudian ditugaskan untuk x.

Penutupan

Penutupan adalah salah satu konsep yang paling penting, dan sering paling sedikit dipahami, dalam JavaScript. Aku tidak akan menutupinya secara rinci dalam sini, dan bukannya merujuk Anda ke *Lingkup & Penutup* judul seri ini. Tapi saya ingin mengatakan beberapa hal tentang hal itu sehingga Anda memahami kecuali bahwa con umum. Ini akan menjadi salah satu teknik yang paling penting dalam JS Anda Skill-ditetapkan.

Anda dapat menganggap penutupan sebagai cara untuk “mengingat” dan terus mengakses ruang lingkup (variabel nya) fungsi ini bahkan sekali fungsi selesai berjalan. Mempertimbangkan:

```
fungsi makeAdder ( x ) {
        // parameter `x` adalah variabel dalam

        // fungsi batin `menambahkan ()` menggunakan `x`, sehingga //
        memiliki "penutupan" di atasnya
        fungsi menambahkan ( y ) {
                kembali y + x ; };


        kembali menambahkan
; }
```

Referensi ke dalam menambahkan(..) Fungsi yang akan kembali dengan setiap panggilan ke luar yang makeAdder (..) mampu mengingat apa pun x
nilai disahkan ke makeAdder (..). Sekarang, mari kita gunakan makeAd der (..):

```
// `plusOne` mendapat referensi ke dalam` menambahkan (..) `// fungsi
dengan penutupan selama parameter` x` dari // terluar `makeAdder (..)`

var tambah satu = makeAdder ( 1 );

// `plusTen` mendapat referensi ke dalam` menambahkan (..) `// fungsi dengan
penutupan selama parameter` x` dari // terluar `makeAdder (..)`
```

```
var plusTen = makeAdder ( 10 );

tambah satu ( 3 );          // 4 <- 1 + 3
tambah satu ( 41 );         // 42 <- 1 + 41

plusTen ( 13 );             // 23 <- 10 + 13
```

Lebih lanjut tentang bagaimana kode ini bekerja:

1. Ketika kita sebut makeAdder (1), kita kembali referensi untuk nya batin
   menambahkan(..) yang mengingat x sebagai 1. Kami menyebutnya referensi fungsi ini
   tambah satu(..).
2. Ketika kita sebut makeAdder (10), kita kembali referensi lain untuk
   nya dalam menambahkan(..) yang mengingat x sebagai 10. Kami menyebutnya referensi fungsi ini plusTen
   (..).
3. Ketika kita sebut Plusone (3), itu menambah 3 (nya dalam y) ke 1
   (Diingat oleh x), dan kita mendapatkan 4 hasilnya.
4. Ketika kita sebut plusTen (13), itu menambah 13 (nya dalam y) ke 10
   (Diingat oleh x), dan kita mendapatkan 23 hasilnya.

Jangan khawatir jika ini tampaknya aneh dan membingungkan pada awalnya-bisa! Ini akan mengambil banyak latihan untuk memahami sepenuhnya.

Tapi percayalah, sekali Anda melakukannya, itu salah satu teknik yang paling kuat dan berguna dalam semua program. Ini pasti worth upaya untuk membiarkan mendidih otak Anda pada penutupan untuk sedikit. Pada bagian berikutnya, kita akan mendapatkan lebih banyak latihan kecil dengan penutupan.

modul

penggunaan yang paling umum dari penutupan di JavaScript adalah modul Pat-tern. Modul membiarkan Anda menentukan rincian pribadi pelaksanaan (ables variabel-, fungsi) yang tersembunyi dari dunia luar, serta API publik yang *aku s* diakses dari luar.
Mempertimbangkan:

```
fungsi pemakai () {
    var nama pengguna . kata sandi ;

    fungsi doLogin ( pemakai . pw ) {
        nama pengguna = pemakai ;
        kata sandi = pw ;

        // melakukan sisa pekerjaan masuk
    }
```

```
    var publicAPI = {
        masuk : doLogin
    };

    kembali publicAPI ; }

// membuat `user` modul misalnya
var fred = pemakai ();

fred . masuk ( "Fred" . "12Battery34!" );
```

Itu Pengguna () Fungsi berfungsi sebagai lingkup luar yang memegang ables variabel- nama pengguna dan kata sandi, serta dalam doLogin () tion func-; ini semua rincian dalam pribadi ini pemakai modul yang tidak dapat diakses dari dunia luar.

Kami tidak callling pengguna baru() di sini, di berpose pur-, meskipun fakta bahwa mungkin tampaknya lebih umum untuk sebagian besar pembaca. Pengguna () hanya tion Fungsi, bukan kelas untuk dipakai, jadi itu hanya disebut normal. menggunakan baru akan inappropri- makan dan benar-benar membuang-buang sumber daya.

Pelaksana Pengguna () menciptakan *contoh* dari pemakai modul-lingkup baru dibuat, dan dengan demikian salinan baru dari masing-masing variabel dalam / fungsi. Kami menetapkan hal ini untuk fred. Jika kita menjalankan Pengguna () lagi, kita akan mendapatkan contoh baru yang sama sekali terpisah dari fred.

batin doLogin () fungsi memiliki penutupan lebih nama pengguna dan lulus kata, berarti ia akan mempertahankan akses ke mereka bahkan setelah Pengguna () Fungsi selesai berjalan.

publicAPI adalah obyek dengan satu properti / metode di atasnya, masuk, yang merupakan referensi ke dalam doLogin () fungsi. Ketika kita kembali publicAPI dari Pengguna (), menjadi contoh yang kita sebut fred.

Pada titik ini, terluar Pengguna () Fungsi telah selesai mengeksekusi. mally normalisasi, Anda akan berpikir variabel dalam seperti nama pengguna dan kata sandi telah pergi. Tapi di sini mereka tidak, karena ada penutupan di masuk() fungsi menjaga mereka hidup.

Itu sebabnya kita dapat memanggil fred.login (..) -the sama seperti memanggil batin doLogin (..) -dan itu masih dapat akses nama pengguna dan kata sandi variabel batin.

Ada kesempatan baik bahwa hanya dengan sekilas ini singkat di penutupan dan pola modul, beberapa masih sedikit membingungkan. Tidak apa-apa! Dibutuhkan beberapa pekerjaan untuk membungkus otak Anda di sekitarnya. Dari sini, pergi membaca *Lingkup & Penutup* judul seri ini untuk lebih eksplorasi mendalam.

Identifier ini

Konsep lain yang sangat umum disalahpahami dalam JavaScript adalah ini kata kunci. Sekali lagi, ada beberapa bab di dalam *ini & Prototip Obyek* judul seri ini, jadi di sini kita hanya akan sebentar intro- Duce konsep.

Walaupun mungkin sering tampak bahwa ini terkait dengan "terns Pat-berorientasi objek," di JS ini adalah mekanisme yang berbeda. Jika fungsi memiliki ini referensi di dalamnya, bahwa ini referensi poin sekutu usu- ke obyek. tapi yang obyek menunjuk ke tergantung pada bagaimana fungsi dipanggil. Sangat penting untuk menyadari bahwa ini *tidak* mengacu pada fungsi itu sendiri, seperti kesalahpahaman yang paling umum. Berikut adalah ilustrasi cepat:

```
fungsi foo () {
        menghibur . mencatat ( ini . bar );
}

var bar = "global" ;

var obj1 = {
        bar : "Obj1" .
        foo : foo
};

var obj2 = {
        bar : "Obj2"
};

// --------

foo ();                         // "global"
```

```
obj1 . foo ();                  // "obj1"
foo . panggilan ( obj2 ); // " obj2"
baru foo ();                    // tidak terdefinisi
```

Ada empat aturan untuk bagaimana ini mendapat mengatur, dan mereka ditampilkan di empat baris terakhir dari potongan bahwa:

1. foo () berakhir pengaturan ini ke obyek global di non-ketat

   Modus-modus yang ketat, ini akan menjadi tidak terdefinisi dan Anda akan mendapatkan error dalam mengakses bar properti-begitu " global" adalah nilai ditemukan this.bar.

2. obj1.foo () set ini ke obj1 obyek.
3. foo.call (obj2) set ini ke obj2 obyek.
4. foo baru () set ini untuk merek obyek kosong baru. Intinya: untuk memahami apa ini poin,

Anda harus memeriksa sampel bagaimana fungsi tersebut dipanggil. Ini akan menjadi salah satu dari empat cara hanya ditampilkan, dan yang kemudian akan menjawab apa ini aku s.

Untuk informasi lebih lanjut tentang ini, lihat Bab 1 dan 2 dari *ini & Prototip Obyek* judul seri ini.

prototip

Mekanisme prototipe dalam JavaScript cukup rumit. Kami hanya akan meliriknya sini. Anda akan ingin menghabiskan banyak waktu meninjau Bab 4-6 dari *ini & Prototip Obyek* judul seri ini untuk semua rincian.

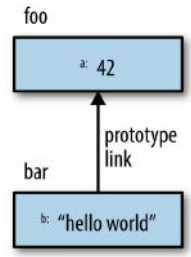
Ketika Anda referensi properti pada objek, jika properti yang tidak ada, JavaScript akan secara otomatis menggunakan referensi jenis proto internal yang objek untuk menemukan objek lain untuk mencari properti pada. Anda bisa memikirkan ini hampir sebagai fallback jika properti ing miss.

Internal linkage referensi prototipe dari satu objek ke belakang fall- yang terjadi pada saat objek dibuat. Cara paling sederhana untuk menggambarkan itu adalah dengan utilitas built-in yang disebut Object.create (..).

Mempertimbangkan:

```
var foo = {
        Sebuah : 42
};

// membuat `bar` dan link ke `foo`
var bar = Obyek . membuat ( foo );

bar . b = "Halo Dunia" ;

bar . b ;           // "Halo Dunia"
bar . Sebuah ;      // 42 <- didelegasikan kepada `foo`
```

Ini dapat membantu untuk memvisualisasikan foo dan bar objek dan hubungan-kapal mereka:



Itu Sebuah properti tidak benar-benar ada di bar objek, tetapi karena bar adalah prototipe-terkait dengan foo, JavaScript secara otomatis jatuh kembali ke mencari Sebuah pada foo objek, di mana itu ditemukan.

linkage ini mungkin tampak seperti fitur aneh bahasa. Cara yang paling umum fitur ini digunakan-dan saya berpendapat, disalahgunakan

- adalah mencoba untuk meniru / palsu "kelas" mekanisme dengan "warisan." Tapi cara yang lebih alami menerapkan prototipe adalah pola yang disebut "perilaku delegasi," di mana Anda sengaja merancang benda Anda terkait untuk dapat *melimpahkan* dari satu ke yang lain untuk bagian-bagian dari perilaku yang dibutuhkan.

Untuk informasi lebih lanjut tentang prototipe dan perilaku delegasi, lihat Bab 4-6 dari *ini & Prototip Obyek* judul seri ini.

Old & New

Beberapa fitur JS kita sudah tertutup, dan tentunya banyak fitur tercakup dalam sisa seri ini, adalah tambahan baru dan belum tentu tersedia di browser lama. Bahkan, beberapa fitur terbaru dalam spesifikasi bahkan tidak diimplementasikan dalam browser belum stabil.

Jadi, apa yang Anda lakukan dengan hal-hal baru? Apakah Anda hanya harus menunggu sekitar selama bertahun-tahun atau puluhan tahun untuk semua browser lama memudar ke dalam ketidakjelasan?

Itulah cara banyak orang berpikir tentang situasi, tapi itu benar-benar tidak pendekatan yang sehat untuk JS.

Ada dua teknik utama yang dapat Anda gunakan untuk "membawa" yang lebih baru hal JavaScript untuk browser lama: polyfilling dan transpiling.

Polyfilling

Kata "polyfill" adalah Istilah diciptakan (oleh Remy Sharp) digunakan untuk merujuk mengambil definisi fitur baru dan memproduksi sepotong kode yang setara dengan perilaku, tetapi mampu berjalan di lingkungan JS tua.

Misalnya, ES6 mendefinisikan sebuah utilitas yang disebut Number.isNaN (..) untuk pro vide akurat, non-kereta cek NaN nilai-nilai, mencela asli Isnan (..) utilitas. Tapi mudah untuk Polyfill utilitas itu sehingga Anda dapat mulai menggunakannya dalam kode Anda terlepas dari apakah pengguna akhir adalah di browser ES6 atau tidak. Mempertimbangkan:

```
jika ( ! Jumlah . Isnan ) {
    Jumlah . Isnan = fungsi Isnan ( x ) {
        kembali x ! == x ; }; }
```

Itu jika penjaga pernyataan terhadap menerapkan definisi polyfill di ES6 browser mana itu sudah akan ada. Jika tidak sudah ada, kita mendefinisikan Number.isNaN (..).

Cek kita lakukan di sini mengambil keuntungan dari permainan kata-kata dengan NaN nilai-nilai, yaitu bahwa mereka satu-satunya nilai di seluruh bahasa yang tidak sama dengan dirinya sendiri. Sehingga NaN nilai adalah satu-satunya yang akan membuat x! == x menjadi benar.

Tidak semua fitur baru sepenuhnya polyfillable. Kadang-kadang sebagian perilaku dapat polyfilled, tapi masih ada penyimpangan kecil. Anda harus benar-benar, benar-benar berhati-hati dalam menerapkan polyfill sendiri, untuk memastikan Anda mengikuti spesifikasi sebagai ble ketat sebagai KEMUNGKINAN.

Atau lebih baik lagi, menggunakan set yang sudah diperiksa dari polyfills yang dapat Anda percaya, seperti yang disediakan oleh ES5-Shim dan ES6-Shim .

Transpiling

Tidak ada cara untuk Polyfill sintaks baru yang telah ditambahkan ke sebersit. Sintaks baru akan melemparkan kesalahan dalam mesin JS tua yang belum diakui / tidak valid.

Jadi pilihan yang lebih baik adalah dengan menggunakan alat yang mengubah kode baru Anda menjadi setara kode yang lebih tua. Proses ini biasa disebut “spiling tran-,” sebuah istilah untuk mengubah + kompilasi.

Pada dasarnya, kode sumber Anda menulis dalam bentuk sintaks baru, tapi apa yang Anda menyebarkan ke browser adalah kode transpiled dalam bentuk sintaks tua. Anda biasanya memasukkan transpiler ke proses membangun, mirip dengan linter kode atau minifier Anda.

Anda mungkin bertanya-tanya mengapa Anda akan pergi ke kesulitan untuk menulis sintaks baru hanya untuk memilikinya transpiled pergi ke yang lebih tua kode-mengapa tidak hanya menulis kode yang lebih tua secara langsung?

Ada beberapa alasan penting yang harus peduli tentang transpi- ling:

- Sintaks baru ditambahkan ke bahasa dirancang untuk membuat kode Anda lebih mudah dibaca dan dipertahankan. Setara tua sering jauh lebih rumit. Anda harus memilih menulis lebih baru dan bersih sintaks, tidak hanya untuk diri sendiri tetapi untuk semua bers mem- lain dari tim pengembangan.

- Jika Anda transpile hanya untuk browser lama, tapi melayani pajak syn baru ke browser terbaru, Anda bisa mengambil keuntungan dari peramban

optimasi kinerja dengan sintaks baru. Hal ini juga memungkinkan pembuat browser memiliki lebih kode di dunia nyata untuk menguji mentations diimple- dan optimasi pada.

- Menggunakan sintaks baru awal memungkinkan untuk diuji lebih bersemangat di dunia nyata, yang memberikan umpan balik sebelumnya kepada komite Jawa-Script (TC39). Jika masalah yang ditemukan cukup dini, mereka dapat diubah / diperbaiki sebelum kesalahan-kesalahan desain bahasa menjadi permanen.

Berikut adalah contoh cepat transpiling. ES6 menambahkan fitur yang disebut “nilai-nilai parameter default.” Ini terlihat seperti ini:

```
fungsi foo ( Sebuah = 2 ) {
    menghibur . mencatat ( Sebuah );
}

foo ();             // 2
foo ( 42 ); // 42
```

Sederhana, kan? Membantu, juga! Tapi itu sintaks baru yang tidak valid dalam mesin ES6 pra. Jadi apa yang akan transpiler lakukan dengan kode itu untuk membuatnya berjalan di lingkungan yang lebih tua?

```
fungsi foo () {
    var Sebuah = argumen [ 0 ] ! == ( kosong 0 ) ? argumen [ 0 ] : 2 ;
    menghibur . mencatat ( Sebuah );
}
```

Seperti yang Anda lihat, itu memeriksa untuk melihat apakah argumen [0] nilai adalah void 0 (aka tidak terdefinisi), dan jika demikian menyediakan 2 nilai default; jika tidak, itu memberikan apa pun yang berlalu.

Selain bisa sekarang menggunakan sintaks lebih bagus bahkan di browser lama, melihat kode transpiled sebenarnya menjelaskan perilaku yang dimaksudkan lebih jelas.

Anda mungkin tidak menyadari hanya dari melihat versi ES6 yang tidak terdefinisi adalah satu-satunya nilai yang tidak bisa secara eksplisit lulus dalam untuk parameter default-nilai, tapi kode transpiled membuat yang jauh lebih jelas.

Detail penting terakhir untuk menekankan tentang transpilers adalah bahwa mereka sekarang harus dianggap sebagai bagian standar dari ekosistem pengembangan JS dan proses. JS akan terus berkembang, jauh lebih cepat dari sebelumnya, jadi setiap beberapa bulan sintaks baru dan membangun struktur kendala pada aspek baru akan ditambahkan.

Jika Anda menggunakan transpiler secara default, Anda akan selalu dapat beralih bahwa untuk sintaks yang lebih baru setiap kali Anda merasa berguna, daripada selalu menunggu selama bertahun-tahun untuk browser saat ini untuk phase out. Ada cukup transpilers besar beberapa bagi Anda untuk memilih dari. Berikut adalah beberapa pilihan yang baik pada saat tulisan ini:

babel (Sebelumnya 6to5)
: Transpiles ES6 + ke ES5

Traceur
: Transpiles ES6, ES7, dan seterusnya ke ES5

Non-JavaScript

So far, the only things we've covered are in the JS language itself. The reality is that most JS is written to run in and interact with environ- ments like browsers. A good chunk of the stuff that you write in your code is, strictly speaking, not directly controlled by JavaScript. That probably sounds a little strange.

The most common non-JavaScript JavaScript you'll encounter is the DOM API. For example:

```
var el = document . getElementById ( "foo" );
```

The document variable exists as a global variable when your code is running in a browser. It's not provided by the JS engine, nor is it particularly controlled by the JavaScript specification. It takes the form of something that looks an awful lot like a normal JS object,
but it's not really exactly that. It's a special object, often called a "host object." Moreover, the getElementById(..) method on document looks like a normal JS function, but it's just a thinly exposed interface to a built-in method provided by the DOM from your browser. In some (newer-generation) browsers, this layer may also be in JS, but tradi- tionally the DOM and its behavior is implemented in something more like C/C++.

Another example is with input/output (I/O). Everyone's favorite alert(..) pops up a message box in the user's browser window. alert(..) is provided to your JS program by the browser, not by the JS engine itself. The call you make sends the

message to the browser internals and it handles drawing and dis- playing the message box. The same goes with `console.log(..)`; your browser provides such mechanisms and hooks them up to the developer tools. This book, and this whole series, focuses on JavaScript the language. That's why you don't see any substantial coverage of these non- JavaScript JavaScript mechanisms. Nevertheless, you need to be aware of them, as they'll be in every JS program you write!

Review

The first step to learning JavaScript's flavor of programming is to get a basic understanding of its core mechanisms like values, types, function closures, `this`, and prototypes.

Of course, each of these topics deserves much greater coverage than you've seen here, but that's why they have chapters and books dedi- cated to them throughout the rest of this series. After you feel pretty comfortable with the concepts and code samples in this chapter, the rest of the series awaits you to really dig in and get to know the lan- guage deeply.

The final chapter of this book will briefly summarize each of the other titles in the series and the other concepts they cover besides what we've already explored.

CHAPTER 3

Into YDKJS

What is this series all about? Put simply, it's about taking seriously the task of learning *all parts of JavaScript,* not just some subset of the language that someone called "the good parts," and not just what- ever minimal amount you need to get your job done at work. Serious developers in other languages expect to put in the effort to learn most or all of the language(s) they primarily write in, but JS developers seem to stand out from the crowd in the sense of typi- cally not learning very much of the language. This is not a good thing, and it's not something we should continue to allow to be the norm. The *You Don't Know JS (YDKJS)* series stands in stark contrast to the typical approaches to learning JS, and is unlike almost any other JS books you will read. It challenges you to go beyond your comfort zone and to ask the deeper "why" questions for every single behav- ior you encounter. Are you up for that challenge?

I'm going to use this final chapter to briefly summarize what to expect from the rest of the books in the series, and how to most effectively go about building a foundation of JS learning on top of *YDKJS.*

Scope & Closures

Perhaps one of the most fundamental things you'll need to quickly come to terms with is how scoping of variables really works in Java- Script. It's not enough to have anecdotal fuzzy *beliefs* about scope.

The *Scope & Closures* title starts by debunking the common miscon- ception that JS is an "interpreted language" and therefore not com- piled. Nope.

The JS engine compiles your code right before (and sometimes dur- ing!) execution. So we use some deeper understanding of the com- piler's approach to our code to understand how it finds and deals with variable and function declarations. Along the way, we see the typical metaphor for JS variable scope management, "hoisting." This critical understanding of "lexical scope" is what we then base our exploration of closure on for the last chapter of the book. Clo- sure is perhaps the single most important concept in all of JS, but if you haven't first grasped firmly how scope works, closure will likely remain beyond your grasp.

One important application of closure is the module pattern, as we briefly introduced in this book in Chapter 2 . The module pattern is perhaps the most prevalent code organization pattern in all of Java- Script; deep understanding of it should be one of your highest prior- ities.

this & Object Prototypes

Perhaps one of the most widespread and persistent mistruths about JavaScript is that the this keyword refers to the function it appears in. Terribly mistaken. The this keyword is dynamically bound based on how the function in question is executed, and it turns out there are four simple rules to understand and fully determine this binding. Closely related to the this keyword is the object prototype mecha- nism, which is a look-up chain for properties, similar to how lexical scope variables are found. But wrapped up in the prototypes is the other huge miscue about JS: the idea of emulating (fake) classes and (so-called "prototypal") inheritance.

Unfortunately, the desire to bring class and inheritance design pat- tern thinking to JavaScript is just about the worst thing you could try to do, because while the syntax may trick you into thinking there's something like classes present, in fact the prototype mecha- nism is fundamentally opposite in its behavior.

What's at issue is whether it's better to ignore the mismatch and pre- tend that what you're implementing is "inheritance," or whether it's more appropriate to learn and embrace how the object prototype system actually works. The latter is more appropriately named "behavior delegation."

This is more than syntactic preference. Delegation is an entirely dif- ferent, and more powerful, design pattern, one that replaces the need to design with classes and inheritance. But these assertions will absolutely fly in the face of nearly every other blog post, book, and conference talk on the subject for the entirety of JavaScript's lifetime. The claims I make regarding delegation versus inheritance come not from a dislike of the language and its syntax, but from the desire to see the true capability of the language properly leveraged and the endless confusion and frustration wiped away.

But the case I make regarding prototypes and delegation is a much more involved one than what I will indulge here. If you're ready to reconsider everything you think you know about JavaScript "classes" and "inheritance," I offer you the chance to "take the red pill" (*The Matrix,* 1999) and check out Chapters 4-6 of the *this & Object Proto- types* title of this series.

Types & Grammar

The third title in this series primarily focuses on tackling yet another highly controversial topic: type coercion. Perhaps no topic causes more frustration with JS developers than when you talk about the confusions surrounding implicit coercion.

By far, the conventional wisdom is that implicit coercion is a "bad part" of the language and should be avoided at all costs. In fact, some have gone so far as to call it a "flaw" in the design of the lan- guage. Indeed, there are tools whose entire job is to do nothing but scan your code and complain if you're doing anything even remotely like coercion.

But is coercion really so confusing, so bad, so treacherous, that your code is doomed from the start if you use it?

I say no. After having built up an understanding of how types and values really work in Chapters 1-3, Chapter 4 takes on this debate and fully explains how coercion works, in all its nooks and crevices.

We see just what parts of coercion really are surprising and what parts actually make complete sense if given the time to learn. But I'm not merely suggesting that coercion is sensible and learna- ble; I'm asserting that coercion is an incredibly useful and totally underestimated tool that *you should be using in your code*. I'm saying that coercion, when used properly, not only works, but makes your code better. All the naysayers and doubters will surely scoff at such a position, but I believe it's one of the main keys to upping your JS game.

Do you want to just keep following what the crowd says, or are you willing to set all the assumptions aside and look at coercion with a fresh perspective? The *Types & Grammar* title of this series will coerce your thinking.

Async & Performance

The first three titles of this series focus on the core mechanics of the language, but the fourth title branches out slightly to cover patterns on top of the language mechanics for managing asynchronous pro- gramming. Asynchrony is not only critical to the performance of our applications, it's increasingly becoming *the* critical factor in writability and maintainability.

The book starts first by clearing up a lot of terminology and concept confusion around things like "async," "parallel," and "concurrent," and explains in depth how such things do and do not apply to JS. Then we move into examining callbacks as the primary method of enabling asynchrony. But it's here that we quickly see that the call- back alone is hopelessly insufficient for the modern demands of asynchronous programming. We identify two major deficiencies of callbacks-only coding: *Inversion of Control (* IoC) trust loss and lack of linear reason-ability.

To address these two major deficiencies, ES6 introduces two new mechanisms (and indeed, patterns): *promises* and *generators*.

Promises are a time-independent wrapper around a "future value," which lets you reason about and compose them regardless of if the value is ready or not yet. Moreover, they effectively solve the IoC trust issues by routing callbacks through a trustable and composable promise mechanism.

Generators introduce a new mode of execution for JS functions, whereby the generator can be paused at yield points and be resumed asynchronously later. The pause-and-resume capability enables synchronous, sequential-looking code in the generator to be processed asynchronously behind the scenes. By doing so, we address the non-linear, non-local-jump confusions of callbacks and thereby make our asynchronous code sync-looking so as to be more reason-able.

But it's the combination of promises and generators that "yields" our most effective asynchronous coding pattern to date in JavaScript. In fact, much of the future sophistication of asynchrony coming in ES7 and later will certainly be built on this foundation. To be serious about programming effectively in an async world, you're going to need to get really comfortable with combining promises and genera- tors.

If promises and generators are about expressing patterns that let our programs run more concurrently and thus get more processing accomplished in a shorter period, JS has many other facets of perfor- mance optimization worth exploring.

Chapter 5 delves into topics like program parallelism with Web Workers and data parallelism with SIMD, as well as low-level opti- mization techniques like ASM.js. Chapter 6 takes a look at perfor- mance optimization from the perspective of proper benchmarking techniques, including what kinds of performance to worry about and what to ignore.

Writing JavaScript effectively means writing code that can break the constraint barriers of being run dynamically in a wide range of browsers and other environments. It requires a lot of intricate and detailed planning and effort on our parts to take a program from "it works" to "it works well." The *Async & Performance* title is designed to give you all the tools and skills you need to write reasonable and performant JavaScript code.

ES6 & Beyond

No matter how much you feel you've mastered JavaScript to this point, the truth is that JavaScript is never going to stop evolving, and moreover, the rate of evolution is increasing rapidly. This fact is

almost a metaphor for the spirit of this series, to embrace that we'll never fully *know* every part of JS, because as soon as you master it all, there's going to be new stuff coming down the line that you'll need to learn.

This title is dedicated to both the short- and mid-term visions of where the language is headed, not just the *known* stuff like ES6 but the *likely* stuff beyond.

While all the titles of this series embrace the state of JavaScript at the time of this writing, which is midway through ES6 adoption, the pri- mary focus in the series has been more on ES5. Now, we want to turn our attention to ES6, ES7, and beyond…

Since ES6 is nearly complete at the time of this writing, *ES6 & Beyond* starts by dividing up the concrete stuff from the ES6 land- scape into several key categories, including new syntax, new data structures (collections), and new processing capabilities and APIs. We cover each of these new ES6 features, in varying levels of detail, including reviewing details that are touched on in other books of this series.

Some exciting ES6 things to look forward to reading about: destruc- turing, default parameter values, symbols, concise methods, compu- ted properties, arrow functions, block scoping, promises, generators, iterators, modules, proxies, weakmaps, and much, much more! Phew, ES6 packs quite a punch!

The first part of the book is a roadmap for all the stuff you need to learn to get ready for the new and improved JavaScript you'll be writing and exploring over the next couple of years. The latter part of the book turns attention to briefly glance at things that we can likely expect to see in the near future of JavaScript. The most important realization here is that post-ES6, JS is likely going to evolve feature by feature rather than version by version, which means we can expect to see these near-future things coming much sooner than you might imagine.

The future for JavaScript is bright. Isn't it time we start learning it?

Review

The *YDKJS* series is dedicated to the proposition that all JS develop- ers can and should learn all of the parts of this great language. No person's opinion, no framework's assumptions, and no project's deadline should be the excuse for why you never learn and deeply understand JavaScript.

We take each important area of focus in the language and dedicate a short but very dense book to fully explore all the parts of it that you perhaps thought you knew but probably didn't fully. "You Don't Know JS" isn't a criticism or an insult. It's a realization that all of us, myself included, must come to terms with. Learning JavaScript isn't an end goal but a process. We don't know JavaScript, yet. But we will!

APPENDIX A

Acknowledgments

I have many people to thank for making this book title and the over- all series happen.

First, I must thank my wife Christen Simpson, and my two kids Ethan and Emily, for putting up with Dad always pecking away at the computer. Even when not writing books, my obsession with JavaScript glues my eyes to the screen far more than it should. That time I borrow from my family is the reason these books can so deeply and completely explain JavaScript to you, the reader. I owe my family everything.

I'd like to thank my editors at O'Reilly, namely Simon St.Laurent and Brian MacDonald, as well as the rest of the editorial and marketing staff. They are fantastic to work with, and have been especially accommodating during this experiment into "open source" book writing, editing, and production.

Thank you to the many folks who have participated in making this series better by providing editorial suggestions and corrections, including Shelley Powers, Tim Ferro, Evan Borden, Forrest L. Nor- vell, Jennifer Davis, Jesse Harlin, Kris Kowal, Rick Waldron, Jordan Harband, Benjamin Gruenbaum, Vyacheslav Egorov, David Nolen, and many others. A big thank you to Jenn Lukas for writing the foreword for this title.

Thank you to the countless folks in the community, including mem- bers of the TC39 committee, who have shared so much knowledge with the rest of us, and especially tolerated my incessant questions

and explorations with patience and detail. John-David Dalton, Juriy “kangax” Zaytsev, Mathias Bynens, Axel Rauschmayer, Nicholas Zakas, Angus Croll, Reginald Braithwaite, Dave Herman, Brendan Eich, Allen Wirfs-Brock, Bradley Meck, Domenic Denicola, David Walsh, Tim Disney, Peter van der Zee, Andrea Giammarchi, Kit Cambridge, Eric Elliott, and so many others, I can’t even scratch the surface. Since the *You Don’t Know JS* series was born on Kickstarter, I also wish to thank all my (nearly) 500 generous backers, without whom this series could not have happened:

Jan Szpila, nokiko, Murali Krishnamoorthy, Ryan Joy, Craig Patch- ett, pdqtrader, Dale Fukami, ray hatfield, R0drigo Perez [Mx], Dan Petitt, Jack Franklin, Andrew Berry, Brian Grinstead, Rob Suther- land, Sergi Meseguer, Phillip Gourley, Mark Watson, Jeff Carouth, Alfredo Sumaran, Martin Sachse, Marcio Barrios, Dan, Aimely- neM, Matt Sullivan, Delnatte Pierre-Antoine, Jake Smith, Eugen Tudorancea, Iris, David Trinh, simonstl, Ray Daly, Uros Gruber, Justin Myers, Shai Zonis, Mom & Dad, Devin Clark, Dennis Palmer, Brian Panahi Johnson, Josh Marshall, Marshall, Dennis Kerr, Matt Steele, Erik Slagter, Sacah, Justin Rainbow, Christian Nilsson, Delapouite, D.Pereira, Nicolas Hoizey, George V. Reilly, Dan Reeves, Bruno Laturner, Chad Jennings, Shane King, Jeremiah Lee Cohick, od3n, Stan Yamane, Marko Vucinic, Jim B, Stephen Collins, Ægir Þorsteinsson, Eric Pederson, Owain, Nathan Smith, Jeanetteurphy, Alexandre ELISÉ, Chris Peterson, Rik Watson, Luke Matthews, Justin Lowery, Morten Nielsen, Vernon Kesner, Chetan Shenoy, Paul Tregoing, Marc Grabanski, Dion Almaer, Andrew Sul- livan, Keith Elsass, Tom Burke, Brian Ashenfelter, David Stuart, Karl Swedberg, Graeme, Brandon Hays, John Christopher, Gior, manoj reddy, Chad Smith, Jared Harbour, Minoru TODA, Chris Wigley, Daniel Mee, Mike, Handyface, Alex Jahraus, Carl Furrow, Rob Foulkrod, Max Shishkin, Leigh Penny Jr., Robert Ferguson, Mike van Hoenselaar, Hasse Schougaard, rajan venkataguru, Jeff Adams, Trae Robbins, Rolf Langenhuijzen, Jorge Antunes, Alex Koloskov, Hugh Greenish, Tim Jones, Jose Ochoa, Michael Brennan-White, Naga Harish Muvva, Barkóczi Dávid, Kitt Hods- den, Paul McGraw, Sascha Goldhofer, Andrew Metcalf, Markus Krogh, Michael Mathews, Matt Jared, Juanfran, Georgie Kirschner, Kenny Lee, Ted Zhang, Amit Pahwa, Inbal Sinai, Dan Raine, Schabse Laks, Michael Tervoort, Alexandre Abreu, Alan Joseph Williams, NicolasD, Cindy Wong, Reg Braithwaite, LocalPCGuy, Jon Friskics, Chris Merriman, John Pena, Jacob Katz, Sue Lock- wood, Magnus Johansson, Jeremy Crapsey, Grzegorz Pawłowski, nico nuzzaci, Christine Wilks, Hans Bergren, charles montgomery,

Ariel בר- לבב Fogel, Ivan Kolev, Daniel Campos, Hugh Wood, Christian Bradford, Frédéric Harper, Ionuț Dan Popa, Jeff Trimble, Rupert Wood, Trey Carrico, Pancho Lopez, Joël kuijten, Tom A Marra, Jeff Jewiss, Jacob Rios, Paolo Di Stefano, Soledad Penades, Chris Gerber, Andrey Dolganov, Wil Moore III, Thomas Marti- neau, Kareem, Ben Thouret, Udi Nir, Morgan Laupies, jory carson- burson, Nathan L Smith, Eric Damon Walters, Derry Lozano- Hoyland, Geoffrey Wiseman, mkeehner, KatieK, Scott MacFarlane, Brian LaShomb, Adrien Mas, christopher ross, Ian Littman, Dan Atkinson, Elliot Jobe, Nick Dozier, Peter Wooley, John Hoover, dan, Martin A. Jackson, Héctor Fernando Hurtado, andy ennamorato, Paul Seltmann, Melissa Gore, Dave Pollard, Jack Smith, Philip Da Silva, Guy Israeli, @megalithic, Damian Crawford, Felix Gliesche, April Carter Grant, Heidi, jim tierney, Andrea Giammarchi, Nico Vignola, Don Jones, Chris Hartjes, Alex Howes, john gibbon, David

J. Groom, BBox, Yu *Dilys* Sun, Nate Steiner, Brandon Satrom, Brian Wyant, Wesley Hales, Ian Pouncey, Timothy Kevin Oxley, George Terezakis, sanjay raj, Jordan Harband, Marko McLion, Wolfgang Kaufmann, Pascal Peuckert, Dave Nugent, Markus Liebelt, Welling Guzman, Nick Cooley, Daniel Mesquita, Robert Syvarth, Chris Coyier, Rémy Bach, Adam Dougal, Alistair Duggin, David Loidolt, Ed Richer, Brian Chenault, GoldFire Studios, Carles Andrés, Carlos Cabo, Yuya Saito, roberto ricardo, Barnett Klane, Mike Moore, Kevin Marx, Justin Love, Joe Taylor, Paul Dijou, Michael Kohler, Rob Cassie, Mike Tierney, Cody Leroy Lindley, tofuji, Shimon Schwartz, Raymond, Luc De Brouwer, David Hayes, Rhys Brett- Bowen, Dmitry, Aziz Khoury, Dean, Scott Tolinski - Level Up, Clement Boirie, Djordje Lukic, Anton Kotenko, Rafael Corral, Phi- lip Hurwitz, Jonathan Pidgeon, Jason Campbell, Joseph C., Swif- tOne, Jan Hohner, Derick Bailey, getify, Daniel Cousineau, Chris Charlton, Eric Turner, David Turner, Joël Galeran, Dharma Vaga- bond, adam, Dirk van Bergen, dave ♥♫★ furf, Vedran Zakanj, Ryan McAllen, Natalie Patrice Tucker, Eric J. Bivona, Adam Spoo- ner, Aaron Cavano, Kelly Packer, Eric J, Martin Drenovac, Emilis, Michael Pelikan, Scott F. Walter, Josh Freeman, Brandon Hudgeons, vijay chennupati, Bill Glennon, Robin R., Troy Forster, otaku_coder, Brad, Scott, Frederick Ostrander, Adam Brill, Seb Flippence, Michael Anderson, Jacob, Adam Randlett, Standard, Joshua Clanton, Sebastian Kouba, Chris Deck, SwordFire, Hannes Papenberg, Richard Woeber, hnzz, Rob Crowther, Jedidiah Broad- bent, Sergey Chernyshev, Jay-Ar Jamon, Ben Combee, luciano bonachela, Mark Tomlinson, Kit Cambridge, Michael Melgares, Jacob Adams, Adrian Bruinhout, Bev Wieber, Scott Puleo, Thomas Herzog, April Leone, Daniel Mizieliński, Kees van Ginkel, Jon Abrams, Erwin Heiser, Avi Laviad, David newell, Jean-Francois Turcot, Niko Roberts, Erik Dana, Charles Neill, Aaron Holmes, Grzegorz Ziółkowski, Nathan Youngman, Timothy, Jacob Mather,

Michael Allan, Mohit Seth, Ryan Ewing, Benjamin Van Treese, Marcelo Santos, Denis Wolf, Phil Keys, Chris Yung, Timo Tijhof, Martin Lekvall, Agendine, Greg Whitworth, Helen Humphrey, Dougal Campbell, Johannes Harth, Bruno Girin, Brian Hough, Darren Newton, Craig McPheat, Olivier Tille, Dennis Roethig, Mathias Bynens, Brendan Stromberger, sundeep, John Meyer, Ron Male, John F Croston III, gigante, Carl Bergenhem, B.J. May, Rebe- kah Tyler, Ted Foxberry, Jordan Reese, Terry Suitor, afeliz, Tom Kiefer, Darragh Duffy, Kevin Vanderbeken, Andy Pearson, Simon Mac Donald, Abid Din, Chris Joel, Tomas Theunissen, David Dick, Paul Grock, Brandon Wood, John Weis, dgrebb, Nick Jenkins, Chuck Lane, Johnny Megahan, marzsman, Tatu Tamminen, Geof- frey Knauth, Alexander Tarmolov, Jeremy Tymes, Chad Auld, Sean Parmelee, Rob Staenke, Dan Bender, Yannick derwa, Joshua Jones, Geert Plaisier, Tom LeZotte, Christen Simpson, Stefan Bruvik, Jus- tin Falcone, Carlos Santana, Michael Weiss, Pablo Villoslada, Peter deHaan, Dimitris Iliopoulos, seyDoggy, Adam Jordens, Noah Kant- rowitz, Amol M, Matthew Winnard, Dirk Ginader, Phinam Bui, David Rapson, Andrew Baxter, Florian Bougel, Michael George, Alban Escalier, Daniel Sellers, Sasha Rudan, John Green, Robert Kowalski, David I. Teixeira (@ditma), Charles Carpenter, Justin Yost, Sam S, Denis Ciccale, Kevin Sheurs, Yannick Croissant, Pau Fracés, Stephen McGowan, Shawn Searcy, Chris Ruppel, Kevin Lamping, Jessica Campbell, Christopher Schmitt, Sablons, Jonathan Reisdorf, Bunni Gek, Teddy Huff, Michael Mullany, Michael Für- stenberg, Carl Henderson, Rick Yoesting, Scott Nichols, Hernán Ciudad, Andrew Maier, Mike Stapp, Jesse Shawl, Sérgio Lopes, jsu- lak, Shawn Price, Joel Clermont, Chris Ridmann, Sean Timm, Jason Finch, Aiden Montgomery, Elijah Manor, Derek Gathright, Jesse Harlin, Dillon Curry, Courtney Myers, Diego Cadenas, Arne de Bree, João Paulo Dubas, James Taylor, Philipp Kraeutli, Mihai Păun, Sam Gharegozlou, joshjs, Matt Murchison, Eric Windham, Timo Behrmann, Andrew Hall, joshua price, Théophile Villard This series is being produced in an open source fashion, including editing and production. We owe GitHub a debt of gratitude for making that sort of thing possible for the community! Thank you again to all the countless folks I didn't name but who I nonetheless owe thanks. May this series be "owned" by all of us and serve to contribute to increasing awareness and understanding of the JavaScript language, to the benefit of all current and future community contributors.

About the Author

Kyle Simpson is an Open Web Evangelist from Austin, TX, who's passionate about all things JavaScript. He's an author, workshop trainer, tech speaker, and OSS contributor/leader.

Colophon

The cover font for *Up & Going* is Interstate. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.