|  |  |
| --- | --- |
| **Tipe Numerik**  DARA SRI CANDRA (L2001400080)  DAFTAR TIPE DATA PADA JAVASCRIPT | Pada dasarnya JavaScript hanya mengenal dua macam tipe numerik, yaitu bilangan bulat (integer) dan bilangan pecahan(real/float).  Untuk bilangan bulat, kita dapat merepresentasikan dengan basis desimal, oktal atau heksadesimal.  Contoh :  var A = 100;  var B = 0x2F;  untuk pendeklarasian tipe bilangan real, dapat menggunakan tanda titik atau notasi ilmiah (notasi E).  Contoh :  var a = 3.14533567;  var b = 1.23456E+3; |
| **Tipe String** | Untuk mendeklarasikan tipe string dapat dilakukan dengan cara menuliskan string diantara tanda petik tunggal (’) atau tanda petik ganda (”)  Contoh :  var  str  =’Contoh deklarasi string’;  var str1 = ”cara ini juga bisa untuk menulis string”; |
| **Tipe Boolean** | Tipe boolean hanya mempunyai nilai True atau False. Tipe ini biasanya digunakan untuk mengecek suatu kondisi atau keadaan.  Contoh :  var  X  = (Y > 90);  contoh diatas menunjukkan bahwa jika Y lebih besar dari 90 maka X akan bernilai True. |
| **Tipe Null** | Tipe Null digunakan untuk merepresentasikan variabel yang tidak diberi nilai awal (inisialisasi). |

Nama fungsi & Pembuatan Fungsi pada Javascript

Sebuah fungsi pada Javascript dibuat dengan cara seperti berikut:

|  |  |
| --- | --- |
| 1  2  3  4 | **function** tambah(a, b) {  hasil = a + b;  **return** hasil;  } |

Cara penulisan fungsi seperti ini dikenal dengan nama *function declaration*, atau deklarasi fungsi. Terdapat empat komponen yang membangun fungsi yang baru kita definisikan di atas, yaitu:

1. Kata kunci function, yang memberitahu Javascript bahwa kita akan membuat fungsi.
2. Nama fungsi, dalam contoh di atas adalah tambah. Dengan memberikan sebuah fungsi nama maka kita dapat merujuk ke fungsi tersebut dengan nama yang diberikan. Harus diingat bawa nama fungsi bersifat *opsional*, yang berarti **fungsi pada Javascript tidak harus diberi nama**. Kita akan membahas tentang hal ini lebih dalam nanti.
3. Daftar parameter fungsi, yaitu a, b pada contoh di atas. Daftar parameter ini selalu dikelilingi oleh tanda kurung (()). Parameter boleh kosong, tetapi tanda kurung wajib tetap dituliskan. Parameter fungsi akan secara otomatis didefinisikan menjadi variabel yang hanya bisa dipakai di dalam fungsi. Variabel pada parameter ini diisi dengan nilai yang dikirimkan kepada fungsi secara otomatis.
4. Sekumpulan perintah yang ada di dalam kurung kurawal ({}). Perintah-perintah ini dikenal dengan nama badan fungsi. Badan fungsi dieksekusi secara berurut ketika fungsi dijalankan.

Penulisan deklarasi fungsi (*function declaration*) seperti di atas merupakan cara penulisan fungsi yang umumnya kita gunakan pada bahasa pemrograman imperatif dan berorientasi objek. Tetapi selain deklarasi fungsi Javascript juga mendukung cara penulisan fungsi lain, yaitu dengan memanfaatkan ekspresi fungsi (*function expression*). Ekspresi fungsi merupakan cara pembuatan fungsi yang memperbolehkan kita melewatkan nama fungsi. Fungsi yang dibuat tanpa nama dikenal dengan sebutan fungsi anonim atau fungsi lambda. Berikut adalah cara membuat fungsi dengan ekspresi fungsi:

|  |  |
| --- | --- |
| 1  2  3  4 | **var** tambah = **function** (a, b) {  hasil = a + b;  **return** hasil;  }; |

Terdapat hanya sedikit perbedaan antara ekspresi fungsi dan deklarasi fungsi:

1. Penamaan fungsi. Pada deklarasi fungsi, kita langsung memberikan nama fungsi sesuai dengan sintaks yang disediakan Javascript. Menggunakan ekspresi fungsi kita pada dasarnya menyimpan sebuah fungsi anonim ke dalam variabel, dan nama fungsi adalah nama variabel yang kita buat. Perlu diingat juga bahwa pada dasarnya ekspresi fungsi *adalah* fungsi anonim. Penyimpanan ke dalam variabel hanya diperlukan karena kita akan memanggil fungsi nantinya.
2. Ekspresi fungsi dapat dipandang sebagai sebuah ekspresi atau perintah standar bagi Javascript, sama seperti ketika kita menuliskan kode var i = 0;. Deklarasi fungsi merupakan konstruksi khusus untuk membuat fungsi. Hal ini berarti pada akhir dari ekspresi fungsi kita harus menambahkan ;, sementara pada deklarasi fungsi hal tersbut tidak penting.

Karena pada Javascript sebuah fungsi juga adalah sekaligus sebuah objek, maka pada buku ini kita akan hampir selalu menggunakan ekspresi fungsi. Setiap kali kita menciptakan fungsi, pada dasarnya kita membuat sebuah objek Functionbaru, dengan nama yang kita berikan. Karenanya, secara eksplisit menuliskan bahwa kita membuat objek baru dan memperlakukan objek tersebut sama seperti perintah-perintah lain dalam program akan menyederhanakan kode program kita, yang pada akhirnya akan mempermudah kita mengerti kode kita nantinya.

Aturan pembuatan fungsi, baik ekspresi fungsi maupun deklarasi fungsi, sama dengan aturan penulisan ekspresi. Di mana kita dapat menuliskan ekspresi, kita dapat mendefinisikan fungsi juga. Karena aturan ini, maka kita juga dapat mendefinisikan fungsi di dalam fungsi lainnya. Fungsi yang berada di dalam fungsi lainnya memiliki akses terhadap semua variabel yang ada pada fungsi penampungnya. Keterhubungan fungsi di dalam fungsi ini dikenal dengan nama *closure*. Kita akan membahas tentang *closure* dan melihat bagaimana *closure* memberikan kemampuan ekspresi yang sangat besar kepada pengembang pada bagian berikutnya.

**Note**

Terdapat satu lagi cara membuat fungsi pada Javascript, yaitu dengan menggunakan objek Function. Tetapi kita tidak akan membahas cara ini, karena cara yang ketiga akan sangat jarang digunakan.

Fungsi sebagai Objek

Sebelum melihat bagaimana fungsi dapat dipanggil, kita akan melihat keterhubungan antara fungsi dengan objek terlebih dahulu. Kita perlu mengerti hubungan antara fungsi dan objek karena terdapat empat cara pemanggilan fungsi pada Javascript, dan dua dari empat cara tersebut melibatkan konsep fungsi sebagai objek.

Fungsi pada javascript adalah sebuah objek. Sebagai sebuah objek, semua fungsi dalam Javascript merupakan turunan dari Function.prototype. Function.prototype juga adalah merupakan turunan dari Object.prototype, sama seperti semua objek-objek lain dalam Javascript. Perbedaan utama fungsi dengan objek lain pada umumnya adalah fungsi dapat dipanggil, dan memiliki dua buah properti khusus, yaitu konteks pemanggilan fungsi dan kode pada isi badan fungsi. Kegunaan dari dua buah properti khusus ini akan kita lihat pada bagian selanjutnya.

Sebagai sebuah objek, fungsi juga dapat kita perlakukan sama dengan objek lainnya. Pada bagian sebelumnya kita telah melihat bahwa fungsi dapat disimpan di dalam variabel. Fungsi juga dapat kita simpan di dalam array atau objek lain, dikirimkan sebagai argumen dari fungsi lain, atau dikembalikan dari sebuah fungsi. Sama seperti objek, kita juga dapat mengaitkan fungsi (*method*) kepada fungsi.

Pemanggilan Fungsi

Sebuah fungsi dapat dipanggil untuk menjalankan seluruh kode yang ada di dalam fungsi tersebut, sesuai dengan parameter yang kita berikan. Pemanggilan fungsi dilakukan dengan cara menuliskan nama fungsi tersebut, kemudian mengisikan argumen yang ada di dalam tanda kurung.

Misalkan fungsi tambah yang kita buat pada bagian sebelumnya:

|  |  |
| --- | --- |
| 1  2  3  4 | **var** tambah = **function** (a, b) {  **var** hasil = a + b;  **return** hasil;  }; |

dapat dipanggil seperti berikut:

|  |  |
| --- | --- |
| 1 | tambah(3, 5); |

Yang terjadi pada kode di atas adalah kita menggantikan a dan b masing-masing dengan 3 dan 5. Seperti yang dapat dilihat, hal ini berarti pengisian argumen pada saat pemanggilan fungsi harus berurut, sesuai dengan deklarasi fungsi.

Sama seperti sebuah variabel, fungsi juga mengembalikan nilai ketika dipanggil. Dalam kasus di atas, tambah(3, 5) akan mengembalikan nilai 8. Nilai ini tentunya dapat disimpan ke dalam variabel baru, atau bahkan dikirimkan sebagai sebuah argumen ke fungsi lain lagi:

|  |  |
| --- | --- |
| 1  2  3  4  5 | **var** simpan = tambah(3, 5); *// simpan === 8*  tambah(simpan, 2); *// mengembalikan 10*  tambah(tambah(3, 5), 2) *// juga mengembalikan 10*  tambah(tambah(2, 3), 4) *// mengembalikan 9* |

Fungsi akan mengembalikan nilai ketika kata kunci return ditemukan. Kita dapat mengembalikan fungsi kapanpun, dan fungsi akan segera berhenti ketika kata kunci return ditemukan. Berikut adalah contoh kode yang memberikan gambaran tentang pengembalian nilai fungsi:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **var** naikkan = **function** (n) {  **var** hasil = n + 10;  **return** hasil;  *// kode di bawah tidak dijalankan lagi*  hasil = hasil \* 100;  }  naikkan(10); *// mengembalikan 20*  naikkan(25); *// mengembalikan 35* |

Kita juga dapat langsung memberikan ekspresi kepada return, dan ekspresi tersebut akan dijalankan sebelum nilai dikembalikan. Hal ini berarti fungsi tambah maupun naikkan yang sebelumnya bisa disederhanakan dengan tidak lagi menyimpan nilai di variabel hasil terlebih dahulu:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **var** naikkan = **function** (n) {  **return** n + 10;  }  **var** tambah = **function** (a, b) {  **return** a + b;  }  tambah(4, 4); *// mengembalikan 8*  naikkan(10); *// mengembalikan 20*  tambah(naikkan(5), 7); *// mengembalikan 22* |

Fungsi pada Javascript juga akan selalu mengembalikan fungsi. Ketika tidak menemukan perintah return, Javascript akan mengembalikan undefined pada akhir fungsi.

Pola Pemanggilan Fungsi

Ketika sebuah fungsi dipanggil, secara otomatis Javascript akan memberikan dua nilai tambahan kepada fungsi tersebut. Kedua nilai tambahan ini diberikan bersamaan dengan pemberian nilai argumen fungsi. Adapun kedua nilai yang diberikan adalah this dan arguments.

Nilai arguments merupakan sebuah objek yang mirip dengan array, dan berisi seluruh argumen yang diberikan kepada fungsi. Kita akan membahas penggunaan nilai ini pada bagian selanjutnya.

Nilai this isinya bergantung kepada cara kita memanggil fungsi. Cara pemanggilan fungsi dikenal dengan nama pola pemanggilan (*invocation pattern*) dari fungsi tersebut. Terdapat empat pola pemanggilan fungsi yang ada pada Javascript, yaitu:

1. Method Invocation Pattern,
2. Function Invocation Pattern,
3. Constructor Invocation Pattern, dan
4. Indirect Invocation Pattern.

Mari kita lihat maksud dari masing-masing pola pemanggilan, dan perbedaan nilai this pada setiap pola.

Method Invocation Pattern

Sebuah fungsi yang dijadikan sebagai properti dari objek dikenal dengan istilah *method*. *Method* merupakan salah satu konsep dasar dalam pemrograman berorientasi objek, yang digunakan untuk memberikan sebuah perintah standar bagi sebuah objek. Berikut adalah contoh dari sebuah *method*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **var** papanSkor = {  skor: 0,  tambahSkor: **function** (nilai) {  **this**.skor += (**typeof** nilai === "number")? nilai : 1;  },  ambilSkor: **function** () {  **return** **this**.skor;  }  }; |

Objek papanSkor yang barusan kita buat memiliki satu buah properti, yaitu skor dan dua buah method, yaitu tambahSkor dan ambilSkor.

Method ambilSkor mengembalikan nilai dari properti skor yang sekarang, sementara tambahSkor akan menambahkan nilai skor sesuai dengan parameter yang diberikan oleh pengguna method. Method tambahSkor juga memberikan tambahan sesuai dengan tipe data yang dikirimkan: jika parameter yang diberikan merupakan sebuah angka, maka penambahan dilakukan sesuai dengan jumlah angka yang dikirimkan, jika tidak maka skor akan bertambah satu saja.

Sama seperti properti, pemanggilan method dapat dilakukan dengan dua cara, yaitu dengan menggunakan tanda titik (.) dan kurung siku ([]):

|  |  |
| --- | --- |
| 1  2  3  4  5 | papanSkor.ambilSkor() *// mengembalikan 0*  papanSkor.tambahSkor(1) *// mengembalikan undefined*  papanSkor["ambilSkor"]() *// mengembalikan 1*  papanSkor["tambahSkor"](2) *// mengembalikan undefined*  papanSkor.ambilSkor() *// mengembalikan 3* |

Dari ekspreimen yang kita lakukan pada kode di atas, kita dapat melihat bahwa properti this berisi objek yang menampung dirinya. Dalam hal ini nilai this pada Javascript tidak berbeda dengan nilai this pada bahasa pemrograman berorientasi objek lain pada umumnya. Artinya kita dapat mengakses seluruh properti maupun method dari objek itu sendiri ketika menggunakan this.

Function Invocation Pattern

Ketika sebuah fungsi bukan merupakan properti dari sebuah objek (*method*), nilai tambahan this akan dihubungkan ke objek global Javascript, yaitu window. Mari kita lihat bagaimana this dikaitkan ke objek global:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | *// Variabel nilai disimpan dalam objek global.*  *// Objek global Javascript secara standar adalah window*  *// sehingga nilai akan disimpan dalam window.nilai*  **var** nilai = 100;  nilai; *// mengembalikan 100*  window.nilai; *// mengembalikan 100*  **var** kurang = **function** (n) {  *// this.nilai merupakan window.nilai (!)*  **this**.nilai = **this**.nilai - n;  };  kurang(10);  nilai; *// mengembalikan 90 (!)*  window.nilai; *// mengembalikan 90 (!)*  *// Hal yang sama berlaku untuk fungsi di dalam fungsi juga*  **var** tambah\_sepuluh = **function** () {  **var** tambah = **function** (n) {  *// this.nilai merupakan window.nilai (!!!)*  **this**.nilai = **this**.nilai + n;  };  tambah(10);  };  nilai; *// mengembalikan 90*  tambah\_sepuluh();  nilai; *// mengembalikan 100 (!!!)* |

Pada kode di atas, kita dapat melihat bagaimana nilai this di dalam fungsi kurang maupun tambah terikat dengan variabel global. Tidak peduli berapa tingkat fungsi di dalam fungsi yang kita buat, variabel this tetap akan mengakses objek global. Hal ini tidak hanya berbeda dengan pada bahasa pemrograman lain, tetapi merupakan salah satu kesalahan besar dalam rancangan Javascript. Pada bahasa yang memiliki fitur fungsional dan objek lainnya, this akan mengikat pada fungsi induk dari fungsi tersebut. Dalam contoh di atas, this pada fungsi tambah akan mencari variabel nilai milih fungsi tambah\_sepuluh.

Kesalahan perancangan Javascript ini sangat fatal, karena teknik memanfaatkan fungsi di dalam fungsi (*inner function*) merupakan salah satu teknik mendasar dan sangat berguna dalam pemrograman fungsional. Tetapi setidaknya untuk *inner function* dari sebuah *method* kita memiliki solusi untuk memperbaiki hal ini, yaitu dengan menyimpan nilai thispada fungsi luar ke dalam sebuah variabel. Fungsi yang berada di dalam fungsi luar kemudian dapat mengakses instan objek dengan menggunakan variabel tersebut, seperti berikut:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | *// kita menggunakan objek papanSkor yang di atas kembali*  papanSkor.hattrick = **function** () {  *// variabel that dibuat agar fungsi tambah\_tiga*  *// dapat mengakses objek papanSkor.*  **var** that = **this**;  **var** tambah\_tiga = **function**() {  *// jika kita menggunakan this di sini,*  *// maka kita akan mengakses variabel global skor,*  *// yang tidak ada.*  that.skor = that.skor + 3;  };  tambah\_tiga();  }; |

Sebagai eksperimen, coba ganti variable that di atas dengan variabel this, dan lihat efeknya!

**Note**

Nama variabel that merupakan perjanjian tak tertulis (konvensi) untuk solusi ini. Meskipun tidak ada aturan untuk wajib menggunakan that, menggunakan nama ini akan mempermudah pengembang lain yang harus merawat kode anda (termasuk anda sendiri di masa depan!).

Constructor Invocation Pattern

Sebuah fungsi yang dipanggil dengan diawali dengan perintah new pada Javascript dikenal dengan istilah *constructor invocation*. Setiap kali sebuah fungsi dipanggil dengan prefiks new, maka fungsi tersebut akan otomatis mengembalikan objek baru pada akhir fungsi, meskipun kita tidak memanggil perintah return. Objek yang dikembalikan ini akan dihubungkan kepada prototype dari fungsi yang dipanggil, dan this diikatkan kepada objek baru ini.

Bingung? Mari kita lihat contoh kode:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | *// Ketika dipanggil dengan new, fungsi Manusia akan*  *// mengembalikan sebuah objek baru yang memiliki*  *// satu properti, yaitu "nama".*  **var** Manusia = **function** (nama) {  **this**.nama = nama;  };  **var** andi = **new** Manusia("Andi");  andi.nama; *// mengembalikan "Andi"*  andi; *// mengembalikan { nama: 'Andi' }*  *// Seperti layaknya sebuah objek, kita dapat menambahkan*  *// method baru kepada seluruh objek Manusia yang telah*  *// dibuat.*  Manusia.prototype.usia = **function** (usia) {  **this**.usia = (**typeof** usia === "number")? usia: 0;  };  andi.usia = 27;  andi.usia; *// mengembalikan 27*  andi; *// mengembalikan { nama: 'Andi', usia: 27 }* |

Perhatikan bagaimana pada fungsi Manusia di atas kita sama sekali tidak memanggil perintah return, dan secara otomatis andi diisikan dengan sebuah objek baru ketika kita memanggil Manusia dengan perintah new. Juga seperti objek pada Javascript, ketika kita menambahkan sebuah *method* baru ke Manusia, andi yang hanyalah salah satu turunan Manusia juga menerima *method* baru tersebut.

Fungsi yang dirancang untuk dipanggil bersamaan dengan new seperti Manusia dikenal dengan nama *constructor*. *Constructor* juga secara konvensi ditulis dengan awalan huruf kapital, agar tidak membingungkan pengembang. Hal ini sangat penting karena sebuah fungsi *constructor* yang tidak dipanggil dengan perintah new akan memberikan efek samping yang membahayakan, yaitu memenuhi atau mengubah nilai variabel global. Baca kembali bagian Function Invocation Pattern untuk melihat kenapa hal ini bisa terjadi.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **var** mari = Manusia("Mari");  mari; *// mengembalikan undefined*  nama; *// mengembalikan Mari (!)*  mari.nama; *// mengembalikan TypeError:*  *// Cannot read property 'nama' of undefined* |

Karena memiliki potensi berbahaya seperti ini, sangat tidak disarankan untuk menggunakan fungsi *constructor*. Javascript bukanlah bahasa berorientasi objek *class-based*. Pergunakan Javascript sebagaimana ia dirancang untuk digunakan, yaitu bahasa pemrograman *prototype-based*. Pada bagian selanjutnya nanti kita akan melihat bagaimana kita dapat membuat objek pada Javascript dengan lebih baik, yaitu menggunakan *closure*.

Indirect Invocation Pattern

Karena fungsi pada Javascript juga adalah merupakan sebuah objek, maka fungsi juga dapat memiliki *method*. Terdapat beberapa *method* standar yang selalu ada pada fungsi Javascript, tetapi di bagian ini kita akan melihat dua *method* khusus yang berhubungan dengan nilai this. Kedua *method* khusus ini yaitu call dan apply.

*Method* apply digunakan jika kita ingin mengirimkan argumen ke sebuah fungsi dengan menggunakan array. Terdapat dua parameter yang harus kita kirimkan ke apply, yaitu objek yang ingin kita ikatkan kepada this, dan parameter keduanya adalah sebuah array yang akan digunakan sebagai parameter yang dikirimkan ke fungsi. Dengan begitu, applymemberikan kita fasiliats untuk menentukan nilai this.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | **var** manusia = {  nama: "Adam",  panggil: **function** (sapaan) {  **return** sapaan + " " + **this**.nama + "!";  }  };  manusia.panggil("Halo"); *// mengembalikan "Halo Adam!"*  **var** hawa = {  nama: "Hawa"  };  manusia.panggil.apply(hawa, ["Bonjour"]); *// mengembalikan "Bonjour Hawa!"* |

Kita juga dapat mengirimkan null sebagai parameter pertama dari apply untuk menggunakan objek global sebagai this.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | *// mengembalikan "Guten Tag undefined!" karena tidak ada variabel "nama"*  *// pada konteks global*  manusia.panggil.apply(**null**, ["Guten Tag"]);  *// simpan variabel "nama" dengan isi "Nuh" pada konteks global*  nama = "Nuh";  *// mengembalikan "Guten Tag Nuh!"*  manusia.panggil.apply(**null**, ["Guten Tag"]); |

Hal ini memungkinkan kita untuk menggunakan fungsi global secara efektif. Misalnya jika kita ingin mencari nilai maksimal dari sebuah array, kita dapat langsung menggunakan Math.max daripada dengan perulangan:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | **var** bil = [5, 6, 2, 3, 7];  *// Sama dengan Math.apply(bil[0], bil[1], ...)*  *// berapapun ukuran bil fungsi ini akan tetap berjalan*  **var** max = Math.max.apply(**null**, bil);  max; *// mengembalikan 7*  *// Tanpa menggunakan apply*  *// Kita tidak dapat menggunakan Math.max di sini*  *// karena panjang array tidak akan selalu sama.*  max = -**Infinity**;  **for** (**var** i = 0; i < bil.length; i++) {  **if** (bil[i] > max)  max = bil[i];  }  max; *// mengembalikan 7* |

Satu hal yang perlu diingat ketika menggunakan apply adalah bahwa Javascript memiliki batas jumlah argumen untuk fungsi, yang berbeda-beda pada setiap browser. Jika array yang dikirimkan melebihi batas jumlah argumen maka apa yang terjadi tidak dapat diketahui (tergantung kepada pembuat browser).

*Method* call sendiri berfungsi sama seperti apply, dengan hanya satu perbedaan: call menerima **\*daftar argumen\*** seperti fungsi biasa, sementara apply menerima **\*array argumen\***. Mengambil contoh manusia sebelumnya, kita dapat memanggil call seperti berikut:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | *// kedua fungsi di bawah mengembalikan "Bonjour Hawa!"*  manusia.panggil.apply(hawa, ["Bonjour"]);  manusia.panggil.call(hawa, "Bonjour");  *// kedua fungsi di bawah hasilnya sama*  Math.max.apply(**null**, bil);  Math.max.call(**null**, bil[0], bil[1], ...); |

Argumen Fungsi

Selain this, fungsi pada Javascript juga memiliki satu buah nilai tambahan lagi, yaitu arguments. arguments merupakan nilai yang menampung seluruh argumen yang dikirimkan kepada fungsi, termasuk argumen-argumen yang berlebihan. Jika fungsi hanya meminta dua buah argumen dan pemanggil fungsi mengirimkan empat buah argumen, kita dapat mengakses argumen ketiga dan keempat menggunakan arguments. Hal ini berarti kita dapat membuat fungsi yang bisa menerima jumlah argumen tak tentu, seperti fungsi Math.max yang kita gunakan sebelumnya.

Contoh lain, kita dapat membuat fungsi yang menghitung total dari seluruh argumen yang dikirimkan kepada fungsi tersebut:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **var** total = **function** () {  **var** i, hasil = 0;  **for** (i = 0; i < arguments.length; i++) {  hasil = hasil + arguments[i];  }  **return** hasil;  };  total(1, 2, 3, 4, 5, 6, 7, 8, 9, 10); *// mengembalikan 55* |

Meskipun pada kode di atas kita menggunakan arguments seperti layaknya array (akses dengan [], properti length), sayangnya arguments **bukan array**. arguments merupakan sebuah objek yang *mirip array*. Selalu ingat hal ini agar tidak menggunakan arguments sebagai array, karena menggunakan arguments sebagai array dapat menyebabkan hal-hal yang tak terbayangkan seperti gangguan pernafasan, serangan jantung, kanker, dan error program tiba-tiba.

Scope dan Function Scope

Dalam konteks bahasa pemrograman, *scope* atau cakupan merupakan sebuah aturan yang mengendalikan waktu hidup (*lifetime*) dan akses dari sebuah nilai dalam program. Nilai yang dikendalikan oleh *scope* umumnya adalah variabel. *Scope* sangat penting dalam pemrograman, terutama untuk memudahkan kita dalam menamakan variabel dan manajemen memori. Dalam Java misalnya, sebuah variabel yang dibuat dalam *scope* lokal akan segera dihapuskan dari memori begitu program keluar dari *scope* tersebut untuk menghemat memori.

Kebanyakan bahasa pemrograman mengimplementasikan konsep *scope* seperti yang ada pada Java. Konsep *scope* ini dikenal dengan nama *block scope*. Pada *block scope*, sebuah variabel yang diciptakan di dalam blok (ditandai dengan {}pada Java dan keluarga C) tidak dapat diakses dari luar blok tersebut. Variabel juga biasanya dihapus ketika program keluar dari blok jika bahasa mendukung manajemen memori otomatis. *Block scope* sangat intuitif dan mudah dipahami, serta hampir selalu digunakan oleh kebanyakan bahasa pemrograman.

Javascript, sayangnya, bukanlah bahasa pemrograman biasa. Meskipun Javascript menggunakan sintaks yang sama dengan keluarga bahasa C, Javascript tidak memiliki *block scope*. Aturan *scoping* yang digunakan oleh Javascript adalah *function scope*. Hal ini mengakibatkan banyak kesalah pahaman tentang aturan *scoping* di Javascript. Sekarang kita akan menjernihkan kesalah pahaman tersebut.

Pada Javascript, sebuah blok **\*tidak\*** membuat *scope* baru. Perhatikan kode berikut:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **var** bil = 1;  console.log(bil); *// mencetak 1*  **if** (bil > 0) {  **var** bil = 2;  console.log(bil); *// mencetak 2*  }  console.log(bil); *// mencetak 2 (!)* |

Pada kode di atas, ketika kita menuliskan var bil = 2; kita **tidak** membuat sebuah variabel lokal bernama bil di dalam blok if. Kita hanya mengisikan kembali bil dengan nilai baru, yaitu 2. Pada bahasa keluarga C, jika kita mencetak bil pada akhir program kita akan mendapatkan nilai 1, karena bil yang ada di dalam if berbeda dengan bil yang ada di luarnya. Hal ini sedikit tidak intuitif, terutama untuk yang telah mempelajari bahasa pemrograman lain, dan jika tidak hati-hati kita dapat secara tidak sadar membuat banyak variabel global.

Aturan *scoping* yang dimiliki Javascript hanyalah *function scope*. *Function scope* berarti semua parameter dan variabel yang dibuat di dalam sebuah fungsi tidak dapat diakses di luar fungsi tersebut. Perhatikan kode berikut:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **var** x = 10;  console.log(x); *// mencetak 10*  **var** tambah = **function** (n) {  **var** x = 3;  console.log(x); *// mencetak 3*  **return** n + x;  }  **var** y = tambah(15);  console.log(y); *// mencetak 18 (15 + 3)*  console.log(x); *// mencetak 10* |

Dalam contoh kode kali ini, kita dapat melihat bagaimana Javascript membedakan x yang ada di luar fungsi tambahdengan x yang ada di dalam fungsi. Begitu keluar dari fungsi tambah, nilai x yang diberikan adalah x yang di luar fungsi. Aturan ini mirip dengan aturan yang ada dalam C. Dengan memanfaatkan pengetahuan *function scope* ini, kita dapat membuat *scope* lokal seperti *block scope* dalam Javascript dengan memanfaatkan fungsi anonim. Mari kita lihat bagaimana hal ini dicapai:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | **var** bil = 1;  console.log(bil); *// mencetak 1*  **if** (bil > 0) {  *// Buat sebuah fungsi anonim dan langsung eksekusi fungsi*  (**function**() {  **var** bil = 2;  console.log(bil); *// mencetak 2*  })();  }  console.log(bil); *// mencetak 1 (!)* |

Pada contoh kode di atas, nilai bil yang ada di luar if berbeda dengan bil yang ada di dalam if. Kita membuat sebuah fungsi anonim dalam if, dengan menggunakan semantik (function() { // isi fungsi }). Tanda kurung (()) yang membungkus fungsi digunakan untuk memastikan fungsi dijalankan sebagai ekspresi, bukan perintah. Sementara tanda kurung yang ada di akhir fungsi bertugas untuk mengeksekusi fungsi anonim yang baru dibuat.

Dengan membuat sebuah fungsi anonim seperti contoh di atas, kita memastikan seluruh variabel yang ada di dalam fungsi anonim tersebut memiliki *scope* yang berbeda dengan variabel lain di luar fungsi anonim. Eksekusi fungsi akan memastikan seluruh kode di dalam fungsi berjalan, karena ingat bahwa kode di dalam fungsi tidak akan dijalankan kalau fungsi tidak dipanggil.

Meskipun kita telah dapat membuat *block scope* dalam Javascript, terkadang kita tetap saja dapat secara tidak sengaja membuat kesalahan program karena asumsi *scope* yang salah. Hal ini terutama terjadi ketika kita berpindah dari bahasa yang digunakan di sisi server ke Javascript. Untuk menghindari kesalahan ini, biasanya praktisi Javascript menuliskan semua deklarasi variabel sedini mungkin, ketika program atau fungsi baru akan dimulai. Hal ini bertentangan dengan bahasa pemrograman lain yang biasanya menyarankan untuk mendeklarasikan variabel hanya ketika akan menggunakan variabel tersebut. Walaupun sedikit berbeda, deklarasi variabel sedini mungkin akan membantu program Javascript kita: jika menemukan var di tengah-tengah fungsi, seorang pengembang Javascript berpengalaman akan mengecek *scope* variabel tersebut, dan memindahkan deklarasi ini ke awal fungsi / program.

Closure

Salah satu kelebihan utama *function scope* adalah fungsi yang ada di dalam fungsi lainnya juga memiliki akses terhadap semua nilai-nilai yang dimiliki fungsi penampungnya. Ketika sebuah fungsi menggunakan dan bergantung kepada nilai yang ada di luar dirinya, fungsi tersebut dikenal dengan nama *closure*. Mari kita lihat contoh sebuah *closure*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **var** jalan = **function** () {  **var** sapaan = "Halo ",  sapa = **function** () {  console.log(sapaan + "pembaca!");  };  sapa();  };  jalan(); *// mencetak "Halo pembaca!"* |

Fungsi jalan memiliki dua buah nilai lokal, yaitu variabel sapaan dan fungsi sapa. Fungsi sapa menggunakan variabel sapaan, dan akan mencetak teks yang berbeda-beda tergantung dengan isi dari variabel sapaan. Variabel sapaandikenal dengan nama variabel bebas karena tidak terikat dengan apapun, dan fungsi sapa merupakan *closure* karena menggunakan variabel bebas yang dibuat di luar dirinya.

Oke, *closure* keren dan kelihatannya sederhana. Pertanyaan selanjutnya tentunya adalah, apa **kegunaan** utama dari *closure*? Pada contoh di atas, kita bisa saja langsung menerima parameter sapaan pada fungsi jalan, dan langsung mencetak sapaan sesuai dengan fungsi sapa kan? Terus kenapa kita harus memanggil fungsi sapa lagi di akhir jalan? Kalau bisa mudah, kenapa harus dibuat sulit seperti sekarang?

Jawaban dari seluruh pertanyaan-pertanyaan di atas adalah potensi kemampuan penyusunan (komposisi) fungsi serta penggunaan kembali kode yang ditawarkan oleh *closure*. Contoh di atas memang tidak memperlihatkan penggunaan *closure* yang optimal. Mari kita tingkatkan fungsi jalan sedikit demi sedikit. Misalkan, kita dapat membuat fungsi ini mengembalikan fungsi lain, sehingga pengguna fungsi dapat membangun sapaannya sendiri:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **var** buatSapaan = **function** (sapaan) {  **return** **function** () {  console.log(sapaan + " pembaca!");  }  };  **var** sapaanJerman = buatSapaan("Guten");  **var** sapaanInggris = buatSapaan("Hello");  sapaanJerman(); *// mencetak "Guten pembaca!"*  sapaanInggris(); *// mencetak "Hello pembaca!"* |

Pada contoh di atas, kita mengirimkan nilai berupa kata sapaan dalam bahasa yang diinginkan kepada buatSapaan, dan buatSapaan akan membuat sebuah fungsi baru kepada kita. Fungsi baru ini bersifat anonim, dan dapat mencetak kata sapaan yang kita berikan sebelumnya. Terdapat dua hal yang menarik dari contoh kode di atas:

1. Variabel (parameter) sapaan adalah milik buatSapaan, tetapi variabel ini tetap dapat digunakan oleh sapaanJermandan sapaanInggris ketika buatSapaan **telah selesai dijalankan**. Biasanya variabel sapaan akan dihapus begitu fungsi buatSapaan selesai dijalankan.
2. Meskipun menggunakan (dan bergantung kepada) sapaan, baik sapaanJerman maupun sapaanInggris **tidak dapat mengakses :code:`sapaan`**. Hal ini berarti kedua fungsi ini tidak dapat mengganti nilai sapaan lagi!

Fungsi sapaanInggris dan sapaanJerman juga adalah merupakan *closure*. Lengkapnya, sebuah *closure* terdiri dari dua komponen: fungsi dan lingkungan eksekusi fungsi tersebut. Lingkungan dari fungsi berisi variabel maupun fungsi lokal yang ada **ketika fungsi tersebut dibuat**. *Closure* dapat mengakses dan menggunakan nilai dalam lingkungannya, tetapi pengguna *closure* tidak mendapatkan akses tersebut.

Fungsi buatSapaan masih dapat kita kembangkan lebih lanjut lagi, misalkan dengan menambahkan parameter pada *closure* yang dikembalikan:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **var** buatSapaan = **function** (sapaan) {  **return** **function** (target) {  console.log(sapaan + " " + target + "!");  }  };  **var** sapaanJerman = buatSapaan("Guten");  **var** sapaanInggris = buatSapaan("Hello");  sapaanJerman("Soryu"); *// mencetak "Guten Soryu!"*  sapaanInggris("Holmes"); *// mencetak "Hello Holmes!"* |

Pada contoh kali ini, baik sapaanJerman maupun sapaanInggris dapat menerima sebuah parameter: nama yang akan disapa. Kita dapat mengubah dan menambahkan banyak fungsionalitas lain lagi pada buatSapaan, tetapi poin yang paling penting adalah kita **dapat menuliskan fungsi yang membuat fungsi lain secara dinamis**. Para pengembang program berorientasi objek mengenal hal ini dengan nama *factory*.

Salah satu hal penting yang juga ditawarkan oleh *closure* adalah akses nilai lingkungannya. Karena kita dapat mengakses dan mengubah nilai pada lingkungan *closure*, maka kita juga dapat memanfaatkan *closure* untuk membuat sebuah objek yang memiliki properti *private*. Bayangkan kalau kita membuat *closure* yang mengembalikan objek:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | **var** penghitung = **function** () {  *// variabel hitungan di sini menjadi*  *// variabel private yang tidak dapat*  *// diakses dari luar objek.*  **var** hitungan = 0;  **return** {  tambahHitungan: **function** () {  hitungan = hitungan + 1;  },  ambilNilai: **function** () {  **return** hitungan;  }  };  };  **var** jumlahHarimau = penghitung();  jumlahHarimau.hitungan; *// undefined (seolah-olah tidak ada)*  jumlahHarimau.tambahHitungan();  jumlahHarimau.ambilNilai(); *// mengembalikan 1*  jumlahHarimau.tambahHitungan();  jumlahHarimau.tambahHitungan();  jumlahHarimau.ambilNilai(); *// mengembalikan 3* |

Dalam contoh di atas, fungsi penghitung pada dasarnya adalah sebuah *constructor* karena fungsi ini membuat objek untuk kita. Keuntungan tambahan dari membuat objek dengan cara ini yaitu kita bisa membuat properti privat. *Method*tambahHitungan dan ambilNilai sebagai *closure* tetap dapat mengakses hitungan, karena hitungan masih merupakan variabel dalam lingkungan *closure*. Karena objek yang dikembalikan tidak diikatkan dengan hitungan, maka kita tidak dapat mengakses hitungan melalui objek yang dihasilkan ini. Perhatikan bagaimana pada kode di atas ketika kita mencoba mengakses hitungan kita diberikan nilai undefined, seolah-oleh variabel tersebut tidak ada.

Properti privat memiliki sangat banyak kegunaan, terutama kalau kita merancang pustaka atau framework untuk digunakan orang lain. Ketika kita menambahkan fungsionalitas yang masih bersifat eksperimental misalnya, biasanya kita tidak ingin seluruh variabel yang digunakan oleh fungsi tersebut ke pengguna pustaka. Masih ada banyak kegunaan lain dari properti privat, tetapi kita tidak akan membahasnya sekarang.

Terakhir, kita juga dapat membuat sebuah objek secara langsung dengan mengeksekusi *closure* begitu selesai dibuat. Berikut adalah contoh pembuatannya:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | **var** penghitung = (**function** () {  **var** hitungan = 0;  **return** {  tambahHitungan: **function** () {  hitungan = hitungan + 1;  },  ambilNilai: **function** () {  **return** hitungan;  }  };  })();  penghitung.tambahHitungan();  penghitung.ambilNilai(); *// mencetak 1*  penghitung.tambahHitungan();  penghitung.tambahHitungan();  penghitung.tambahHitungan();  penghitung.ambilNilai(); *// mencetak 4* |