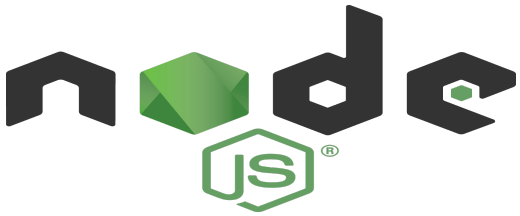


Crash Course

Dr. Bambang Purnomosidi D. P.

PT Wabi Teknologi Indonesia

2018

PT Wabi Teknologi Indonesia
Jl. Raya Janti Karang Jambe no 143
Yogyakarta 55198
Indonesia
Phone: 0274 486664 ext 3431
WhatsApp/Telegram (9:00-16:00): +62 813-3593-7700
General inquiries: info@kamiwabi.id
Engineering inquiries: engineering@kamiwabi.id
Training inquiries: education@kamiwabi.id

00. Tentang Buku Ini

Buku ini merupakan buku yang dirancang untuk keperluan memberikan pengetahuan mendasar tentang JavaScript, khususnya dengan menggunakan *interpreter* Node.js. Pada buku ini akan dibahas dasar-dasar pemrograman menggunakan Node.js. Node.js merupakan software di sisi server yang dikembangkan dari *engine* JavaScript V8 dari Google serta libuv (https://github.com/joyent/libuv). Versi sebelum 0.9.0 menggunakan *libev* dari Mark Lechmann.

Jika selama ini kebanyakan orang mengenal JavaScript hanya di sisi klien (browser), dengan Node.js ini, pemrogram bisa menggunakan JavaScript di sisi server. Meskipun ini bukan hal baru, tetapi paradigma pemrograman yang dibawa oleh Node.js dengan *evented - asynchronous I/O* menarik dalam pengembangan aplikasi *networking*/Web. Kondisi ini memungkinkan para programmer untuk menggunakan 1 bahasa yang sama di sisi server maupun di sisi klien (bahkan beberapa software DBMS juga menggunakan JavaScript untuk *query language*, misalnya MongoDB).

Untuk mengikuti materi yang ada pada buku ini, pembaca diharapkan menyiapkan peranti komputer dengan beberapa software berikut terpasang:

1. Sistem operasi (apa saja sepanjang mendukung Node.js).
2. IDE / Editor Teks. Bisa menggunakan Vim (http://www.vim.org) atau lainnya (Emacs, Visual Studio Code, dan lain-lain).
3. Software utama untuk keperluan workshop ini yaitu Node.js. Versi yang digunakan adalah versi *current*.

01. Instalasi Node.js

Node.js tersedia untuk Linux, Windows, Mac OS X, serta SunOS. Untuk versi Linux, kebanyakan distro sudah menyertakan paket Node.js, hanya saja ada banyak versi dari Node.js dan jika kita menggunakan manajemen paket dari distro Linux, kita hanya bisa menginstall 1 versi saja. Sebagai contoh, di Arch Linux, paket Node.js bisa diinstall dengan perintah ``pacman -S nodejs'' tetapi hanya pada versi resmi di repo Arch Linux (versi 11.4.0-1 pada tanggal 13 Desember 2018).

Langkah instalasi berikut ini adalah langkah untuk instalasi tanpa manajemen paket dari distro Linux.

- ➢ Ambil paket *binary executable* dari http://nodejs/download atau langsung ke http://nodejs.org/dist/. Versi yang digunakan disini adalah *current* yang berisi perkembangan terakhir dari Node.js. Jika ingin versi yang lebih stabil, pilih versi LTS (*Long Term Support*). Download file tersebut, kemudian simpan di direktori tertentu (lokasi bebas, di buku ini diletakkan di $HOME/master/nodejs).

```
» ls -la  ~/master/nodejs/
total 24300
```

```
drwxr-xr-x  3 bpdp bpdp     4096 Dec 13 10:13 ./
drwxr-xr-x 12 bpdp bpdp     4096 Apr 27  2017 ../
...
...
-rw-r--r--         1     bpdp     bpdp     12554872     Dec     11     21:11
node-v11.4.0-linux-x64.tar.xz
...
»
```

➢ Ekstrak ke direktori yang diinginkan. Node.js akan diinstall di direktori $HOME/software:

```
» cd
$ cd software
$ tar -xvf ~/master/nodejs/node-v11.4.0-linux-x64.tar.xz
$ ln -s node-v11.4.0-linux-x64 nodejs
....
....
»
```

➢ Konfigurasi variabel lingkungan. Sebaiknya disimpan pada suatu file (pada buku ini, konfigurasi akan disimpan di $HOME/environment/nodejs):

```
NODEJS_HOME=/home/bpdp/software/nodejs

PATH=$PATH:$NODEJS_HOME/bin
MANPATH=$MANPATH:$NODEJS_HOME/share/man
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$NODEJS_HOME/lib
C_INCLUDE_PATH=$C_INCLUDE_PATH:$NODEJS_HOME/include
CPLUS_INCLUDE_PATH=$CPLUS_INCLUDE_PATH:$NODEJS_HOME/include

export PATH
export MANPATH
export LD_LIBRARY_PATH
export C_INCLUDE_PATH
export CPLUS_INCLUDE_PATH
```

➢ Setiap akan menggunakan Node.js, yang diperlukan adalah men-source file konfigurasi tersebut:

source ~/environment/nodejs

Untuk memeriksa apakah Node.js sudah terinstall:

```
» node -v
nv11.4.0
» npm -v
6.4.1
»
```

02. REPL dan Dasar-dasar JavaScript di Node.js

REPL adalah lingkungan pemrograman interaktif, tempat developer bisa mengetikkan program per baris dan langsung mengeksekusi hasilnya. Biasanya ini digunakan untuk menguji perintah-perintah yang cukup dijalankan pada satu baris atau satu blok segmen kode sumber saja. Karena fungsinya itu, maka istilah yang digunakan adalah REPL (*read-eval-print-loop*), yaitu *loop* atau perulangan baca perintah - evaluasi perintah - tampilkan hasil. REPL sering juga disebut sebagai *interactive top level* atau *language shell*. Tradisi ini sudah dimulai sejak jaman LISP di mesin UNIX di era awal pengembangan *development tools*. Saat ini hampir semua *interpreter/compiler* mempunyai REPL, misalnya Python, Ruby, Scala, PHP, berbagai interpreter/compiler LISP, dan tidak ketinggalan Node.js.

Mengaktifkan REPL

Untuk mengaktifkan REPL dari Node.js, *executable command line program*-nya adalah **node**. Jika **node** dipanggil dengan argumen nama file JavaScript, maka file JavaScript tersebut akan dieksekusi, sementara jika tanpa argumen, akan masuk ke REPL:

```
» node
> .help
.break    Sometimes you get stuck, this gets you out
.clear    Alias for .break
.editor   Enter editor mode
.exit     Exit the repl
.help     Print this help message
.load     Load JS from a file into the REPL session
.save     Save all evaluated commands in this REPL session to a file
> ⏎
```

```
» node -v
nv11.4.0
» npm -v
6.4.1
»
```

Tanda **>** adalah tanda bahwa REPL Node.js siap untuk menerima perintah. Untuk melihat perintah-perintah REPL, bisa digunakan **.help**

Perintah-perintah REPL

Pada sesi REPL, kita bisa memberikan perintah internal REPL maupun perintah-perintah lain yang sesuai dan dikenali sebagai perintah JavaScript. Perintah internal REPL Node.js terdiri atas:

- **.break**: keluar dan melepaskan diri dari "keruwetan" baris perintah di REPL.
- **.clear**: alias untuk .break
- **.editor**: memasuki mode editor, jika ingin menuliskan lebih dari 1 baris.
- **.exit**: keluar dari sesi REPL (bisa juga dengan menggunakan Ctrl-D)
- **.help**: menampilkan pertolong perintah internal REPL
- **.load**: membaca dan mengeksekusi perintah-perintah JavaScript yang terdapat pada suatu file.
- **.save**: menyimpan sesi REPL ke dalam suatu file.

Contoh **.load** untuk mengambil dan menjalankan file JavaScript. Contoh file **simple-http-server.js** - tidak perlu memperhatikan artinya, nanti akan dipelajari.

```
var http = require('http');
http.createServer(function (req, res) {
     res.writeHead(200, {'Content-Type': 'text/plain'});
     res.end('Hello World\n');
}).listen(1337, '127.0.0.1');

console.log('Server running at http://127.0.0.1:1337/');
```

File tersebut akan di-*load* dan dijalankan REPL:

```
» node
> .load simple-http-server.js
var http = require('http');
http.createServer(function (req, res) {
```

```
        res.writeHead(200, {'Content-Type': 'text/plain'});
      res.end('Hello World\n');
}).listen(1337, '127.0.0.1');
console.log('Server running at http://127.0.0.1:1337/');

Server running at http://127.0.0.1:1337/
undefined
>
```

Setelah keluar dari sesi REPL, maka port akan ditutup dan hasil eksekusi di atas akan dibatalkan.

Untuk menyimpan hasil sesi REPL bisa digunakan **.save**, jika tanpa menyebutkan direktori, maka akan disimpan di direktori aktif saat itu.

Dasar-dasar JavaScript di Node.js

Node.js merupakan sistem peranti lunak yang merupakan implementasi dari bahasa pemrograman JavaScript. Spesifikasi JavaScript yang diimplementasikan merupakan spesifikasi resmi dari *ECMAScript* serta *CommonJS* (http://commonjs.org). Dengan demikian, jika sudah pernah mempelajari JavaScript sebelumnya, tata bahasa dari perintah yang dipahami oleh Node.js masih tetap sama dengan JavaScript.

Membaca Masukan dari Stream / Masukan Standar (stdin)

Untuk lebih memahami dasar-dasar JavaScript serta penerapannya di Node.js, seringkali kita perlu melakukan simulasi pertanyaan - proses - keluaran jawaban. Proses akan kita pelajari seiring dengan materi-materi berikutnya, sementara untuk keluaran, kita bisa menggunakan *console.log*. Bagian ini akan menjelaskan sedikit tentang masukan.

Perintah untuk memberi masukan di Node.js sudah tersedia pada pustaka API *Readline* (lengkapnya bisa diakses di http://nodejs.org/api/readline.html). Pola dari masukan ini adalah sebagai berikut:

1. *Require* pustaka Readline
2. membuat *interface* untuk masukan dan keluaran
3. .. gunakan interface ..
4. .. gunakan interface ..
5. .. gunakan interface ..
6. .. gunakan interface ..
7. ..
8. ..
9. tutup interface

Implementasi dari pola diatas bisa dilihat pada kode sumber berikut ini (diambil dari manual Node.js):

```
// readline.js
var readline = require('readline');

var rl = readline.createInterface({
    input: process.stdin,
      output: process.stdout
});

rl.question("What do you think of node.js? ", function(answer) {
  console.log("Thank you for your valuable feedback:", answer);
  rl.close();
});

// hasil:
// $ node readline.js
// What do you think of node.js? awesome!
// Thank you for your valuable feedback: awesome!
// $
```

Catatan: *function(answer)* pada listing di atas merupakan *anonymous function* atau fungsi anonimus. Posisi fungsi pada listing tersebut disebut dengan fungsi *callback*. Untuk keperluan pembahasan saat ini, untuk sementara yang perlu dipahami adalah hasil input akan dimasukkan ke *answer* untuk diproses lebih lanjut. Fungsi dan *callback* akan dibahas lebih lanjut pada pembahasan berikutnya.

Nilai/Value dan Tipe Data

Program dalam JavaScript akan berhubungan dengan data atau nilai. Setiap nilai mempunyai tipe tertentu. JavaScript mengenali berbagai tipe berikut ini:

- Angka: bulat (misalnya 4) atau pecahan (misalnya 3.75)
- Boolean: nilai benar (true) dan salah (false)
- String: diapit oleh tanda petik ganda ("contoh string") atau tunggal ('contoh string')
- null
- undefined

JavaScript adalah bahasa pemrograman yang mengijinkan pemrogram untuk tidak mendefinisikan tipe data pada saat deklarasi dan tipe data bisa berubah-ubah tergantung pada

isi dari data yang ada pada variabel. Jenis ini sering disebut juga dengan *dynamically typed language.*

```
// dynamic.js
var jumlahMahasiswa = 30
console.log('Jumlah mahasiswa dalam satu kelas = ' + jumlahMahasiswa);
// Jumlah mahasiswa dalam satu kelas = 30
```

Pada contoh di atas, kita bisa melihat bahwa data akan dikonversi secara otomatis pada saat program dieksekusi. Khusus untuk operator "+", JavaScript akan melakukan penggabungan string (*string concatenation*), tetapi untuk operator lain, akan dilakukan operasi matematis sesuai operator tersebut (-,/,*). Konversi string ke tipe numerik bisa dilakukan dengan *parseInt(string)* (jika bilangan bulat) dan *parseFloat(string)* (jika bilangan pecahan).

Variabel

Variabel adalah suatu nama yang didefinisikan untuk menampung suatu nilai. Nama ini akan digunakan sebagai referensi yang akan menunjukkan ke nilai yang ditampungnya. Variabel merupakan bagian dari Identifier. Ada beberapa syarat pemberian nama identifier di JavaScript:

- ➢ Dimulai dengan huruf, \textit{underscore} (_), atau tanda dollar (\$).
- ➢ Karakter berikutnya bisa berupa angka, selain ketentuan pertama di atas.
- ➢ Membedakan huruf besar - kecil.

Konvensi yang digunakan oleh pemrogram JavaScript terkait dengan penamaan ini adalah variasi dari metode camel case, yaitu *camelBack*. Contoh: jumlahMahasiswa, linkMenu, status.

Konstanta

Konstanta mirip dengan variabel, hanya saja sifatnya *read-only*, tidak bisa diubah-ubah setelah ditetapkan. Untuk menetapkan konstanta di JavaScript, digunakan kata kunci *const*. Contoh:

```
// const.js
const MENU = "Home";

console.log("Posisi menu = " + MENU);

// mencoba mengisi MENU. berhasil?

MENU = "About";
```

```
console.log("Posisi menu = " + MENU);

// Posisi menu = Home
// Posisi menu = Home
```

Konvensi penamaan konstanta adalah menggunakan huruf besar semua. Bagian ini (sampai saat buku ini ditulis) hanya berlaku di Firefox dan Google Chrome - V8 (artinya berlaku juga untuk Node.js).

Literal

Literal digunakan untuk merepresentasikan nilai dalam JavaScript. Ada beberapa tipe literal.

Literal Array

Array atau variabel berindeks adalah penampung untuk obyek yang menyerupai *list* atau daftar. Obyek array juga menyediakan berbagai fungsi dan metode untuk mengolah anggota yang terdapat dalam daftar tersebut (terutama untuk operasi *traversal* dan permutasi. Listing berikut menunjukkan beberapa operasi untuk literal array.

```
// array.js
var arrMembers = ['one','two',,'three',];
// sengaja ada koma di bagian akhir
console.log(arrMembers[0]);
// hasil: one
console.log(arrMembers[2]);
// hasil: undefined
console.log(arrMembers[3]);
// hasil: three
console.log(arrMembers[4]);
// hasil: undefined - karena tidak ada
console.log(arrMembers.length);
// hasil: 4
var multiArray = [
     ['0-0','0-1','0-2'],
     ['1-0','1-1','1-2'],
     ['2-0','2-1','2-2']];
console.log(multiArray[0][2]);
// hasil: 0-2
console.log(multiArray[1][2]);
// hasil: 1-2
```

Literal Boolean

Literal boolean menunjukkan nilai benar (true) atau salah (false).

Literal Integer

Literal integer digunakan untuk mengekspresikan nilai bilangan bulat. Nilai bilangan bulat dalam JavaScript bisa dalam bentuk:

1. decimal (basis 10): digit tanpa awalan nol.
2. octal (basis 8): digit diawali dengan 1 angka nol. Pada ECMA-262, bilangan octal ini sudah tidak digunakan lagi.
3. hexadecimal (basis 16): digit diawali dengan 0x.

Literal Floating-point

Literal ini digunakan untuk mengekspresikan nilai bilangan pecahan, misalnya 0.4343 atau bisa juga menggunakan E/e (nilai eksponensial), misalnya
-3.1E12.

Literal Obyek

Literal ini akan dibahas di bab yang menjelaskan tentang paradigma pemrograman berorientasi obyek di JavaScript.

Literal String

Literal string mengekspresikan suatu nilai dalam bentuk sederetan karakter dan berada dalam tanda petik (ganda maupun tunggal). Contoh:

- “Kembali ke halaman utama”
- 'Lisensi'
- “Hari ini, Jum'at, tanggal 21 November”
- “1234.543”
- “baris pertama \n baris kedua”

Contoh terakhir di atas menggunakan karakter khusus (\n). Beberapa karakter khusus lainnya adalah:

\b: Backspace
\f: Form feed
\n: New line
\r: Carriage return
\t: Tab

\v: Vertical tab
\': Apostrophe atau single quote
\": Double quote
\\: Backslash (\).
\XXX: Karakter dengan pengkodean Latin-1 dengan tiga digit octal antara 0 and 377. (misal, \251 adalah simbol hak cipta).
\xXX: seperti di atas, tetapi hexadecimal (2 digit).
\uXXXX: Karakter Unicode dengan 3 digit karakter hexadecimal.

Backslash sendiri sering digunakan sebagai *escape character*, misalnya “NaN sering disebut juga sebagai \”Not a Number\””

Aliran Kendali

Alur program dikendalikan melalui pernyataan-pernyataan untuk aliran kendali. Ada beberapa pernyataan aliran kendali yang akan dibahas.

Pernyataan Kondisi if .. else if .. else

Pernyataan ini digunakan untuk mengerjakan atau tidak mengerjakan suatu bagian atau blok program berdasarkan hasil evaluasi kondisi tertentu.

```
// if.js
var kondisi = false;
if (kondisi) {
     console.log('hanya dikerjakan jika kondisi bernilai benar/true');
};
// hasil: n/a, tidak ada hasilnya
var kondisi = true;
if (kondisi) {
     console.log('hanya dikerjakan jika kondisi bernilai benar/true');
};
// hasil: hanya dikerjakan jika kondisi bernilai benar/true
// Contoh berikut lebih kompleks, melibatkan input

var readline = require('readline');

var rl = readline.createInterface({
    input: process.stdin,
      output: process.stdout
});

rl.question("Masukkan angka nilai: ", function(answer) {
```

```
  if (answer > 80) {
    console.log("Nilai: A");
  } else if (answer > 70) {
    console.log("Nilai: B");
  } else if (answer > 40) {
    console.log("Nilai: C");
  } else if (answer > 30) {
    console.log("Nilai: D");
  } else {
    console.log("Tidak lulus");
  }
  rl.close();
});

// hasil:
// hanya dikerjakan jika kondisi bernilai benar/true
// Masukkan angka nilai: 50
// Nilai: C
```

Pernyataan *switch*

Pernyataan ini digunakan untuk mengevaluasi suatu ekspresi dan membandingkan sama atau tidaknya dengan suatu label tertentu di dalam struktur pernyataan switch, serta mengeksekusi perintah-perintah sesuai dengan label yang cocok.

```
// switch.js
var readline = require('readline');

var rl = readline.createInterface({
    input: process.stdin,
      output: process.stdout
});

console.log("Menu");
console.log("====");
console.log("1. Mengisi data");
console.log("2. Mengedit data");
console.log("3. Menghapus data");
console.log("4. Mencari data");
rl.question("Masukkan angka pilihan anda: ", function(answer) {
  console.log("Pilihan anda: " + answer);
  switch (answer) {
```

```
    case "1":
      console.log("Anda memilih menu pengisian data");
      break;
    case "2":
      console.log("Anda memilih menu pengeditan data");
      break;
    case "3":
      console.log("Anda memilih menu penghapusan data");
      break;
    case "4":
      console.log("Anda memilih menu pencarian data");
      break;
    default:
      console.log("Anda tidak memilih salah satu dari menu di atas");
      break;
  }
  rl.close();
});

// hasil:
// $ node switch.js
// Menu
// ====
// 1. Mengisi data
// 2. Mengedit data
// 3. Menghapus data
// 4. Mencari data
// Masukkan angka pilihan anda: 10
// Pilihan anda: 10
// Anda tidak memilih salah satu dari menu di atas
// $ node switch.js
// Menu
// ====
// 1. Mengisi data
// 2. Mengedit data
// 3. Menghapus data
// 4. Mencari data
// Masukkan angka pilihan anda: 2
// Pilihan anda: 2
// Anda memilih menu pengeditan data
```

Looping

Looping atau sering juga disebut “kalang” adalah konstruksi program yang digunakan untuk melakukan suatu blok perintah secara berulang-ulang. Salah satu pernyataan yang digunakan adalah *for*.

```
// for.js
for (var i = 0; i < 9; i++) {
  console.log(i);
}

// hasil:
// 0
// 1
// 2
// 3
// 4
// 5
// 6
// 7
// 8
```

Pernyataan ``for'' juga bisa digunakan untuk mengakses data yang tersimpan dalam struktur data JavaScript (JSON).

```
// forIn.js
var data = {a:1, b:2, c:3};

for (var iterasi in data) {
    console.log("Nilai dari iterasi " + iterasi + " adalah: " +
data[iterasi]);
}

// hasil:
// Nilai dari iterasi a adalah: 1
// Nilai dari iterasi b adalah: 2
// Nilai dari iterasi c adalah: 3
```

Pernyataan lain yang bisa digunakan untuk *looping* adalah *do .. while*. Pernyataan ini digunakan untuk mengerjakan suatu blok program selama suatu kondisi bernilai benar dengan jumlah minimal pengerjaan sebanyak 1 kali.

```
// doWhile.js
var i = 0;
```

```
do {
  i += 2;
  console.log(i);
} while (i < 20);

// hasil:
// 2
// 4
// 6
// 8
// 10
// 12
// 14
// 16
// 18
// 20
```

Variasi dari *do… while* adalah *while*. Seperti *do .. while*, pernyataan ini digunakan untuk mengerjakan suatu blok program secara berulang-ulang selama kondisi bernilai benar. Meskipun demikian, bisa saja blok program tersebut tidak pernah dikerjakan jika pada saat awal expresi dievaluasi sudah bernilai false.

```
// while.js
var n = 0;
var x = 0;

while (n < 5) {
  n ++;
  x += n;
  console.log("Nilai n = " + n);
  console.log("Nilai x = " + x);
}

// hasil:
// Nilai n = 1
// Nilai x = 1
// Nilai n = 2
// Nilai x = 3
// Nilai n = 3
// Nilai x = 6
// Nilai n = 4
// Nilai x = 10
```

```
// Nilai n = 5
// Nilai x = 15
```

Komponen yang mengatur jalannya aliran program alternatif di dalam *looping* adalah *label, break*, dan *continue*. Bagian ini digunakan dalam *looping* dan *switch*.

- *label* digunakan untuk memberi pengenal pada suatu lokasi program sehingga bisa direferensi oleh *break* maupun *continue* (jika dikehendaki).
- *break* digunakan untuk menghentikan eksekusi dan meneruskan alur program ke pernyataan setelah *looping* atau *switch*.
- *continue* digunakan untuk meneruskan eksekusi ke iterasi atau ke kondisi switch berikutnya.

```
// breakContinue.js
var n = 0;
var x = 0;

while (n < 5) {
  n ++;
  x += n;

  if (x%2 == 0) {
    continue;
  };

  if (x>10) {
    break;
  };

  console.log("Nilai n = " + n);
  console.log("Nilai x = " + x);

};
// hasil:
//Nilai n = 1
//Nilai x = 1
//Nilai n = 2
//Nilai x = 3
```

Contoh lain:

```
// breakWithLabel.js
topLabel:
```

```
  for(var k = 0; k < 10; k++){
    for(var m = 0; m < 20; m++){
      if(m == 5){
        console.log("Nilai k = " + k);
        console.log("Nilai m = " + m);
        break topLabel;
      }
    }
  }
// hasil:
//Nilai k = 0
//Nilai m = 5
```

Fungsi

Fungsi merupakan subprogram atau suatu bagian dari keseluruhan program yang ditujukan untuk mengerjakan suatu pekerjaan tertentu dan (biasanya) menghasilkan suatu nilai kembalian. Subprogram ini relatif independen terhadap bagian-bagian lain sehingga memenuhi kaidah "bisa-digunakan-kembali" atau *reusable* pada beberapa program yang memerlukan fungsionalitasnya. Fungsi dalam ilmu komputer sering kali juga disebut dengan i, *routine*, atau *method*. Definisi fungsi dari JavaScript di Node.js bisa dilakukan dengan sintaksis berikut ini:

```
function namaFungsi(argumen1, argumen2, ... , argumentN) {
  ..
  JavaScript code ..
  JavaScript code ..
  JavaScript code ..
  JavaScript code ..
  ..
}
```

Setelah dideklarasikan, fungsi tersebut bisa dipanggil dengan cara sebagai berikut:

```
..
..
  namaFungsi(argumen1, argumen2, ..., argumenN);
..
..
```

Contoh dalam program serta pemanggilannya adalah sebagai berikut:

```
$ node
> function addX(angka) {
... console.log(angka + 10);
... }
undefined
> addX(20);
30
undefined
>
> function add2Numbers(angka1, angka2) {
... return angka1 + angka2;
... }
undefined
> console.log("232 + 432 = " + add2Numbers(232, 432));
232 + 432 = 664
undefined
>
```

Fungsi Anonim

Fungsi anonim adalah fungsi tanpa nama, pemrogram tidak perlu memberikan nama ke fungsi. Biasanya fungsi anonim ini hanya digunakan untuk fungsi yang dikerjakan pada suatu bagian program saja dan tidak dengan maksud untuk dijadikan komponen yang bisa dipakai di bagian lain dari program (biasanya untuk menangani *event* atau *callback*). Untuk mendeklarasikan fungsi ini, digunakan literal *function*.

```
// fungsiAnonim.js
var pangkat = function(angka) {return angka * angka};
console.log(pangkat(10));
// output: 100
```

Fungsi Rekursif

Fungsi rekursif adalah fungsi yang memanggil dirinya sendiri. Contoh dari aplikasi fungsi rekursif adalah pada penghitungan faktorial berikut:

```
function factorial(n) {

  if ((n == 0) || (n == 1))
    return 1;
  else
    return (n * factorial(n - 1));
```

```
}

console.log("factorial(6) = " + factorial(6));

// hasil:
// factorial(6) = 720
```

Fungsi di dalam Fungsi / *Nested Functions*

Saat mendefinisikan fungsi, di dalam fungsi tersebut, pemrogram bisa mendefinisikan fungsi lainnya. Meskipun demikian, fungsi yang terletak dalam suatu definisi fungsi tidak bisa diakses dari luar fungsi tersebut dan hanya tersedia untuk fungsi yang didefinisikan.

```
// nested.js
function induk() {

  var awal = 0;
  function tambahkan() {
    awal++;
  }

  tambahkan();
  tambahkan();

  console.log('Nilai = ' + awal);

}

induk();
tambahkan();

// hasil:
// Nilai = 2
//
// src/bab-02/nested.js:12
// tambahkan();
// ^
// ReferenceError: tambahkan is not defined
//    at Object.<anonymous> (src/bab-02/nested.js:12:1)
//    at Module._compile (module.js:456:26)
//    at Object.Module._extensions..js (module.js:474:10)
```

```
//    at Module.load (module.js:356:32)
//    at Function.Module._load (module.js:312:12)
//    at Function.Module.runMain (module.js:497:10)
//    at startup (node.js:119:16)
//    at node.js:901:3
```

Struktur Data dan Representasi JSON

JSON (*JavaScript Object Notation*) adalah subset dari JavaScript dan merupakan struktur data *native* di JavaScript. Bentuk dari representasi struktur data JSON adalah sebagai berikut (diambil dari http://en.wikipedia.org/wiki/JSON dengan sedikit perubahan:

```
// json.js
var data = {
      "firstName": "John",
      "lastName": "Smith",
      "age": 25,
      "address": {
            "streetAddress": "21 2nd Street",
            "city": "New York",
            "state": "NY",
            "postalCode": "10021"
      },
      "phoneNumber":
      {
            "home": "212 555-1234",
            "fax": "646 555-4567"
      }
}

console.log(data.firstName + " " + data.lastName +
           " has this phone number = "
           + data.phoneNumber.home );

// hasil:
// John Smith has this phone number = 212 555-1234
```

Dari representasi di atas, kita bisa membaca:

- ➢ Nilai data “firstname” adalah “John”
- ➢ Data “address” terdiri atas sub data “streetAddress”, “city”, “state”, dan “postalCode” yang masing-masing mempunyai nilai data sendiri-sendiri.
- ➢ dan seterusnya

Penanganan Error

JavaScript mendukung pernyataan *try .. catch .. finally* serta *throw* untuk menangani error. Meskipun demikian, banyak hal yang tidak sesuai dengan konstruksi ini karena sifat JavaScript yang *asynchronous*. Untuk kasus asynchronous, pemrogram lebih disarankan menggunakan *function callback*.

```
// try.js
try {
  gakAdaFungsiIni();
} catch (e) {
  console.log ("Error: " + e.message);
} finally {
   console.log ("Bagian 'pembersihan', akan dikerjakan, apapun yang terjadi");
};

// hasil:
// Error: gakAdaFungsiIni is not defined
// Bagian 'pembersihan', akan dikerjakan, apapun yang terjadi
```

Jika diperlukan, kita bisa mendefinisikan sendiri error dengan menggunakan pernyataan *throw.*

```
// throw.js
try {
  var a = 1/0;
  throw "Pembagian oleh angka 0";
} catch (e) {
  console.log ("Error: " + e);
};

// hasil:
// Error: Pembagian oleh angka 0
```

03. Paradigma Pemrograman di JavaScript

Pemrograman Fungsional

Pemrograman fungsional, atau sering disebut *functional programming*, selama ini lebih sering dibicarakan di level para akademisi. Meskipun demikian, saat ini terdapat kecenderungan paradigma ini semakin banyak digunakan di industri. Contoh nyata dari implementasi paradigma ini di industri antara lain adalah Scala (http://www.scala-lang.org), OCaml (http://www.ocaml.org), Haskell (http://www.haskell.org), Microsoft F# (http://fsharp.org), dan lain-lain. Dalam konteks paradigma pemrograman, peranti lunak yang dibangun menggunakan pendekatan paradigma ini akan terdiri atas berbagai fungsi yang mirip dengan fungsi matematis. Fungsi matematis tersebut di-evaluasi dengan penekanan pada penghindaran *state* serta *mutable data*. Bandingkan dengan paradigma pemrograman prosedural yang menekankan pada *immutable data* dan definisi berbagai prosedur dan fungsi untuk mengubah *state* serta data.

JavaScript bukan merupakan bahasa pemrograman fungsional yang murni, tetapi ada banyak fitur dari pemrograman fungsional yang terdapat dalam JavaScript. Dalam hal ini, JavaScript banyak dipengaruhi oleh bahasa pemrograman Scheme (http://www.schemers.org). Bab ini akan membahas beberapa fitur pemrograman fungsional di JavaScript. Pembahasan ini didasari pembahasan di bab sebelumnya tentang Fungsi di JavaScript.

Ekspresi Lambda

Ekspresi lambda (*lambda expression*) merupakan hasil karya dari ALonzo Church sekitar tahun 1930-an. Aplikasi dari konsep ini di dalam pemrograman adalah penggunaan fungsi sebagai parameter untuk suatu fungsi. Dalam pemrograman, *lambda function* sering juga disebut dikaitkan dengan fungsi anonimus (fungsi yang dipanggil/dieksekusi tanpa ditautkan (*bound*) ke suatu *identifier*). Berikut adalah implementasi dari konsep ini di JavaSCript:

```
// lambda.js
// Diambil dari
//
http://stackoverflow.com/questions/3865335/what-is-a-lambda-language
// dengan beberapa perubahan

function applyOperation(a, b, operation) {
  return operation(a, b);
}

function add(a, b) {
  return a+b;
}

function subtract(a, b) {
  return a-b;
}
```

```
console.log('1,2, add: ' + applyOperation(1,2, add));
console.log('43,21, subtract: ' + applyOperation(43,21, subtract));

console.log('4^3:  '  +  applyOperation(4,  3,  function(a,b)  {return
Math.pow(a, b)}))

// hasil:
// 1,2, add: 3
// 43,21, subtract: 22
// 4^3: 64
```

Higher-order Function

Higher-order function (sering disebut juga sebagai *functor* adalah suatu fungsi yang setidak-tidaknya menggunakan satu atau lebih fungsi lain sebagai parameter dari fungsi, atau menghasilkan fungsi sebagai nilai kembalian.

```
// hof.js
function forEach(array, action) {
  for (var i = 0; i < array.length; i++ )
    action(array[i]);
}

function print(word) {
  console.log(word);
}

function makeUpperCase(word) {
  console.log(word.toUpperCase());
}

forEach(["satu", "dua", "tiga"], print);
forEach(["satu", "dua", "tiga"], makeUpperCase);

// hasil:
//satu
//dua
//tiga
//SATU
//DUA
//TIGA
```

Closure

Suatu *closure* merupakan definisi suatu fungsi bersama-sama dengan lingkungannya. Lingkungan tersebut terdiri atas fungsi internal serta berbagai variabel lokal yang masih tetap tersedia saat fungsi utama / closure tersebut selesai dieksekusi.

```
// closure.js
// Diambil dengan sedikit perubahan dari:
// https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Closures
function makeAdder(x) {
  return function(y) {
    return x + y;
  };
}

var add5 = makeAdder(5);
var add10 = makeAdder(10);

console.log(add5(2));  // 7
console.log(add10(2)); // 12
```

Currying

Currying memungkinkan pemrogram untuk membuat suatu fungsi dengan cara menggunakan fungsi yang sudah tersedia secara parsial, artinya tidak perlu menggunakan semua argumen dari fungsi yang sudah tersedia tersebut.

```
// currying.js
// Diambil dari:
// http://javascriptweblog.wordpress.com/2010/04/05/
//         curry-cooking-up-tastier-functions/
// dengan sedikit perubahan

function toArray(fromEnum) {
    return Array.prototype.slice.call(fromEnum);
}

Function.prototype.curry = function() {
    if (arguments.length<1) {
        return this; //nothing to curry with - return function
    }
    var __method = this;
```

```
    var args = toArray(arguments);
    return function() {
        return __method.apply(this, args.concat(toArray(arguments)));
    }
}

var add = function(a,b) {
    return a + b;
}

//create function that returns 10 + argument
var addTen = add.curry(10);
console.log(addTen(20)); //30
```

Pemrograman Berorientasi Obyek

Pemrograman Berorientasi Obyek (selanjutnya akan disingkat PBO) adalah suatu paradigma pemrograman yang memandang bahwa pemecahan masalah pemrograman akan dilakukan melalui definisi berbagai kelas kemudian membuat berbagai obyek berdasarkan kelas yang dibuat tersebut dan setelah itu mendefinisikan interaksi antar obyek tersebut dalam memecahkan masalah pemrograman. Obyek bisa saling berinteraksi karena setiap obyek mempunyai properti (sifat / karakteristik) dan *method* untuk mengerjakan suatu pekerjaan tertentu. Jadi, bisa dikatakan bahwa paradigma ini menggunakan cara pandang yang manusiawi dalam penyelesaian masalah.

Dengan demikian, inti dari PBO sebenarnya terletak pada kemampuan untuk mengabstraksikan berbagai obyek ke dalam kelas (yang terdiri atas properti serta method). Paradigma PBO biasanya juga mencakup *inheritance* atau pewarisan (sehingga terbentuk skema yang terdiri atas *superclass* dan *subclass*). Ciri lainnya adalah *polymorphism* dan *encapsulation* / pengkapsulan.

JavaScript adalah bahasa pemrograman yang mendukung PBO dan merupakan implementasi dari ECMAScript. Implementasi PBO di JavaScript adalah *prototype-based programming* yang merupakan salah satu subset dari PBO. Pada *prototype-based programming*, kelas / *class* tidak ada. Pewarisan diimplementasikan melalui *prototype*.

Definisi Obyek

Definisi obyek dilakukan dengan menggunakan definisi *function*, sementara *this* digunakan di dalam definisi untuk menunjukkan ke obyek tersebut. Sementara itu, *Kelas.prototype.namaMethod* digunakan untuk mendefinisikan method dengan nama method *namaMethod* pada kelas Kelas. Perhatikan contoh pada listing berikut.

```
// obyek.js
```

```
var url = require('url');

// Definisi obyek
function Halaman(alamatUrl) {
  this.url = alamatUrl;
  console.log("Mengakses alamat " + alamatUrl);
}

Halaman.prototype.getDomainName = function() {
  return url.parse(this.url, true).host;
}
// sampai disini definisi obyek
// Halaman.prototype.getDomainName => menetapkan method getDomainName
// untuk obyek

var halSatu = new Halaman("http://nodejs.org/api/http.html");
var halDua = new Halaman("http://bpdp.name/login?fromHome");

console.log("Alamat URL yang diakses oleh halSatu = " + halSatu.url);
console.log("Alamat URL yang diakses oleh halDua = " + halDua.url);

console.log("Nama domain halDua = " + halDua.getDomainName());

// hasil:
// Mengakses alamat http://nodejs.org/api/http.html
// Mengakses alamat http://bpdp.name/login?fromHome
//     Alamat     URL     yang     diakses     oleh     halSatu     =
http://nodejs.org/api/http.html
//     Alamat     URL     yang     diakses     oleh     halDua     =
http://bpdp.name/login?fromHome
// Nama domain halDua = bpdp.name
```

Inheritance / Pewarisan

Pewarisan di JavaScript bisa dicapai menggunakan *prototype*. Listing program berikut memperlihatkan bagaimana pewarisan diimplementasikan di JavaScript.

```
// inheritance.js
// Definisi obyek
function Kelas(param) {
  this.property1 = new String(param);
}
```

```
Kelas.prototype.methodSatu = function() {
  return this.property1;
}

var kelasSatu = new Kelas("ini parameter 1 dari kelas 1");

console.log("Property 1 dari kelasSatu = " + kelasSatu.property1);
console.log("Property 1 dari kelasSatu, diambil dari method = " +
kelasSatu.methodSatu());

// Definisi inheritance:
// SubKelas merupakan anak dari Kelas yang didefinisikan
// di atas.

SubKelas.prototype = new Kelas();
SubKelas.prototype.constructor = SubKelas;

function SubKelas(param) {
  this.property1 = new String(param);
}

// method overriding
SubKelas.prototype.methodSatu = function(keHurufBesar) {
  console.log("Ubah ke huruf besar? = " + keHurufBesar);
  if (keHurufBesar) {
    return this.property1.toUpperCase();
  } else {
    return this.property1.toLowerCase();
  }
}

SubKelas.prototype.methodDua = function() {
  console.log("Berada di method dua dari SubKelas");
}

// mari diuji
var subKelasSatu = new SubKelas("Parameter 1 Dari Sub Kelas 1");

console.log("Property 1 dari sub kelas 1 = " +
subKelasSatu.property1);
console.log("Property 1 dari sub kelas 1, dr method+param = " +
subKelasSatu.methodSatu(true));
```

```
console.log("Property 1 dari sub kelas 1, dr method+param = " +
subKelasSatu.methodSatu(false));

console.log(subKelasSatu.methodDua());
// hasil:
//
//Property 1 dari kelasSatu = ini parameter 1 dari kelas 1
//Property 1 dari kelasSatu, diambil dari method  = ini
//parameter 1 dari kelas 1
//Property 1 dari sub kelas 1 = Parameter 1 Dari Sub Kelas 1
//Ubah ke huruf besar? = true
//Property 1 dari sub kelas 1, dr method+param =
//PARAMETER 1 DARI SUB KELAS 1
//Ubah ke huruf besar? = false
//Property 1 dari sub kelas 1, dr method+param =
//parameter 1 dari sub kelas 1
//Berada di method dua dari SubKelas
```

04. Mengelola Paket Menggunakan npm

Apakah npm Itu?

Node.js memungkinkan developer untuk mengembangkan aplikasi secara modular dengan memisahkan berbagai komponen *reusable code* ke dalam pustaka (*library*). Berbagai pustaka tersebut bisa diperoleh di http://npmjs.org. Node.js menyediakan perintah *npm* untuk mengelola paket pustaka di repositori tersebut. Untuk menggunakan utilitas ini, pemrogram harus terkoneksi dengan Internet.

Menggunakan npm

Saat melakukan instalasi Node.js, secara otomatis *npm* akan disertakan. Dengan perintah *npm* tersebut, seorang pemrogram bisa mengelola pustaka yang tersedia di repositori. Jika pemrogram mempunya pustaka yang bisa digunakan oleh orang lain, maka pemrogram yang bersangkutan juga bisa menyimpan pustaka tersebut ke dalam repositori sehingga memungkinkan untuk diinstall oleh pemrogram-pemrogram lain di seluruh dunia. Sintaksis lengkap dari penggunaan perintah *npm* ini adalah sebagai berikut (beberapa bagian tertulis spesifik lokasi direktori di komputer yang digunakan penulis):

```
» npm --help

Usage: npm <command>
```

```
where <command> is one of:
    access, adduser, audit, bin, bugs, c, cache, ci, cit,
    completion, config, create, ddp, dedupe, deprecate,
    dist-tag, docs, doctor, edit, explore, get, help,
    help-search, hook, i, init, install, install-test, it, link,
    list, ln, login, logout, ls, outdated, owner, pack, ping,
    prefix, profile, prune, publish, rb, rebuild, repo, restart,
    root, run, run-script, s, se, search, set, shrinkwrap, star,
    stars, start, stop, t, team, test, token, tst, un,
    uninstall, unpublish, unstar, up, update, v, version, view,
    whoami

npm <command> -h  quick help on <command>
npm -l            display full usage info
npm help <term>   search for help on <term>
npm help npm      involved overview

Specify configs in the ini-formatted file:
    /home/bpdp/.npmrc
or on the command line via: npm <command> --key value
Config info can be viewed via: npm help config

npm@6.4.1
/opt/software/nodejs-dev-tools/node-v11.4.0-linux-x64/lib/node_modules
/npm
```

Pada bagian berikut, kita akan membahas lebih lanjut penggunaan perintah *npm* tersebut.

Instalasi Paket

npm sebenarnya bukan merupakan singkatan dari *Node Package Manager*, meskipun seringkali orang menterjemahkan dengan singkatan tersebut dan *npm* seharusnya ditulis dalam huruf kecil semua seperti yang dijelaskan pada FAQ (*Frequently Asked Questions* - https://npmjs.org/doc/faq.html). npm merupakan bilah alat berbasis baris perintah, dijalankan melalui shell atau *command prompt*. Sama seperti kebanyakan bilah alat berbasis baris perintah lain, npm memiliki struktur perintah *npm perintah argumen*. Instalasi paket dilakukan dengan perintah berikut :

```
» npm install namapaket
```

Perintah diatas akan memasang versi terakhir dari paket "namapaket". Selain itu *npm* juga dapat memasang paket langsung pada sebuah folder, tarball atau tautan untuk sebuah tarball.

Struktur Instalasi Paket Node.js

Dalam instalasi paket pustaka, berkas-berkas akan terletak dalam folder lokal aplikasi *node_modules*. Pada mode instalasi paket pustaka global (dengan -g atau --global dibelakang baris perintah), paket pustaka akan dipasang pada */usr/lib/node_modules* (dengan lokasi instalasi Node.js standar). Mode global memungkinkan paket pustaka digunakan tanpa memasang paket pustaka pada setiap folder lokal aplikasi. Mode global ini juga membutuhkan hak administrasi lebih (*sudo* atau *root*) dari pengguna agar dapat menulis pada lokasi standar.

Jika berada pada direktori $HOME, maka paket-paket npm tersebut akan terinstall di $HOME/.npm, sedangkan jika kita berada di luar direktori $HOME, maka paket-paket tersebut akan terinstall di *$CWD/node_modules* ($CWD = *Current Working Directory* - direktori aktif saat ini). Daftar paket pustaka yang terpasang dapat dilihat menggunakan perintah berikut:

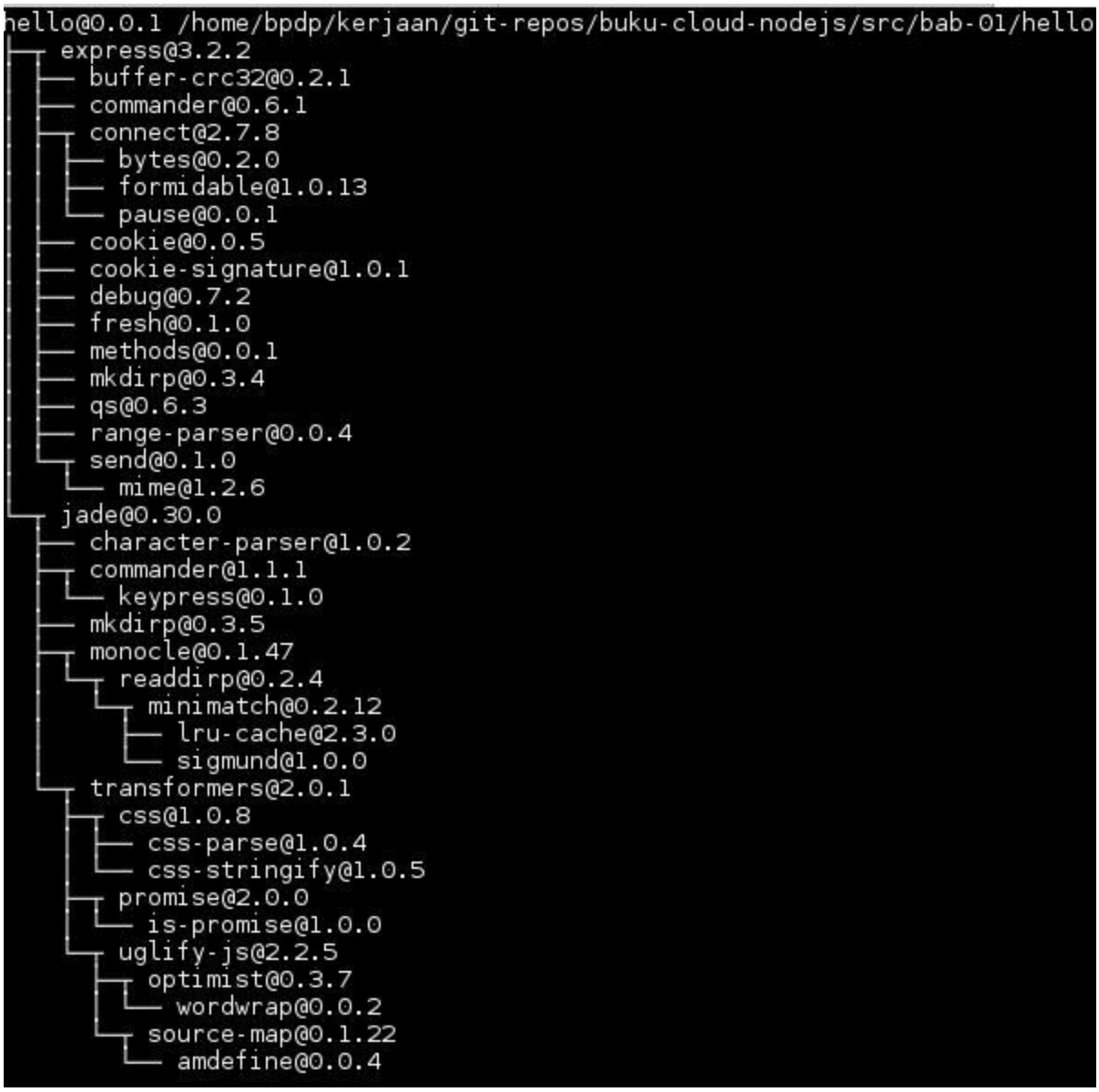
```
» npm ls
--> untuk melihat pada $CWD
    atau
$ npm ls -g
--> untuk melihat pada direktori global
```

Selain melihat daftar paket pustaka yang digunakan dalam aplikasi maupun global, perintah diatas juga akan menampilkan paket dependensi dalam struktur pohon. Jika kita belum menginstall paket-paket yang diperlukan, akan muncul peringatan. Berikut ini adalah contoh peringatan dari paket-paket yang belum terinstall di aplikasi saat mengerjakan perintah *npm ls* di direktori tempat aplikasi tersebut berada:

```
» npm ls
npm WARN package.json hello@0.0.1 No README.md file found!
hello@0.0.1
/home/bpdp/kerjaan/git-repos/buku-cloud-nodejs/src/bab-01/hello
+-- UNMET DEPENDENCY express 3.2.2
+-- UNMET DEPENDENCY jade *

npm ERR! missing: express@3.2.2, required by hello@0.0.1
npm ERR! missing: jade@*, required by hello@0.0.1
npm ERR! not ok code 0
»
```

Jika sudah terinstall, perintah *npm ls* akan menampilkan struktur dari paket yang telah terinstall dalam bentuk struktur pohon seperti pada gambar.



Menghapus Paket / *Uninstall*

Menghapus paket pustaka menggunakan npm pada dasarnya hampir sama dengan saat memasang paket, namun dengan perintah uninstall. Berikut perintah lengkapnya.

```
» npm uninstall namapaket
--> uninstall namapaket di $CWD/node_modules
    atau
» npm uninstall namapaket -g
--> uninstall paket di dir global
»
```

Mencari Paket

Untuk mencari paket, gunakan argumen *search* dan nama atau bagian dari nama paket yang dicari. Contoh berikut ini akan mencari paket dengan kata kunci 'sha512' (tampilan berikut merupakan tampilan yang terpotong):

```
» npm search sha512
NAME      DESCRIPTION                              ...
jshashes  A fast and independent hashing librar...
krypto    High-level crypto library, making the...
passhash  Easily and securely hash passwords wi...
pwhash    Generate password hashes from the com...
...
...
```

Setelah menemukan paketnya, pemrogram bisa menginstall langsung ataupun melihat informasi lebih lanjut tentang pustaka tersebut.

Menampilkan Informasi Paket

Setelah mengetahui nama paket, pemrogram bisa memperoleh informasi lebih lanjut dalam format *human-readable* menggunakan parameter *view*. Contoh dibawah ini menampilkan rincian dari paket *arango.client*:

```
» npm view arango.client

arango.client@0.5.6 | MIT | deps: 1 | versions: 7
ArangoDB javascript client

dist
```

```
.tarball:
https://registry.npmjs.org/arango.client/-/arango.client-0.5.6.tgz
.shasum: 48279e7cf9ea0b4b6766f09671224c46d6e716b0

dependencies:
amdefine: >=0.0.2

maintainers:
- kaerus <anders@kaerus.com>

dist-tags:
latest: 0.5.6

published over a year ago
```

Memperbaharui Paket

Jika terdapat versi baru, kita bisa memperbaharui secara otomatis menggunakan argumen *update* berikut ini:

```
» npm update
--> update paket di $CWD/node_modules
» npm update -g
--> update paket global
```

05. Node.js dan Web: Teknik Pengembangan Aplikasi

Pendahuluan

Pada saat membangun aplikasi Cloud dengan antarmuka web menggunakan Node.js, ada beberapa teknik pemrograman yang bisa digunakan. Bab ini akan membahas berbagai teknik tersebut. Untuk mengerjakan beberapa latihan di bab ini, digunakan suatu file dengan format JSON. File *pegawai.json* berikut ini akan digunakan dalam pembahasan selanjutnya.

```
{
  "pegawai": [
    {
      "id": "1",
      "nama": "Zaky",
      "alamat": "Purwomartani"
```

```
    },
    {
      "id": "2",
      "nama": "Ahmad",
      "alamat": "Kalasan"
    },
    {
      "id": "3",
      "name": "Aditya",
      "alamat": "Sleman"
    }
  ]
}
```

Jika ingin memeriksa validitas dari data berformat JSON, pemrogram bisa menggunakan validator di http://jsonlint.com.

Event-Driven Programming dan *EventEmitter*

Event-Driven Programming (selanjutnya akan disebut EDP) atau sering juga disebut *Event-Based Programming* merupakan teknik pemrograman yang menggunakan *event* atau suatu kejadian tertentu sebagai pemicu munculnya suatu aksi serta aliran program. Contoh event misalnya adalah sebagai berikut:

- Menu dipilih.
- Tombol **Submit** di-klik.
- Server menerima permintaan dari klien.

Pada dasarnya ada beberapa bagian yang harus disiapkan dari paradigma dan teknik pemrograman ini:

- *main loop* atau suatu konstruksi utama program yang menunggu dan mengirimkan sinyal event.
- definisi dari berbagai *event* yang mungkin muncul
- definisi *event-handler* untuk menangani event yang muncul dan dikirimkan oleh *main loop*

Node.js merupakan peranti pengembangan yang menggunakan teknik pemrograman ini. Pada Node.js, EDP ini semua dikendalikan oleh kelas *events.EventEmitter*. Jika ingin menggunakan kelas ini, gunakan *require('events')*. Dalam terminologi Node.js, jika suatu event terjadi, maka dikatakan sebagai *emits an event*, sehingga kelas yang digunakan untuk menangani itu disebut dengan *events.EventEmitter*. Pada dasarnya banyak event yang digunakan oleh berbagai kelas lain di Node.js. Contoh kecil dari penggunaan itu diantaranya

adalah *net.Server* yang meng-*emit* event “connection”, "listening", "close", dan "error". Untuk memahami mekanisme ini, pahami dua kode sumber berikut:

- server.js: mengaktifkan server http (diambil dari manual Node.js)
- server-on-error.js: mencoba mengaktifkan server pada host dan port yang sama dengan server.js. Aktivasi ini akan menyebabkan Node.js meng-*emit* event 'error' karena host dan port sudah digunakan di server.js.

  File server.js dijalankan lebih dulu, setelah itu baru menjalankan server-on-error.js.

```
// server.js
var http = require('http');

http.createServer(function (req, res) {

  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');

}).listen(1337, '127.0.0.1');

console.log('Server running at http://127.0.0.1:1337/');
```

```
// server-on-error.js
var net = require('net');

var server = net.createServer(function(sock) {

  // Event dan event-handler
  // 'data' => jika ada data yang dikirimkan dari klien
  sock.on('data', function(data) {
    console.log('data ' + sock.remoteAddress + ': ' + data);
  });

  // 'close' => jika koneksi ditutup
  sock.on('close', function(data) {
    console.log('koneksi ditutup');
  });

});

server.listen(1337, function() {
  console.log('Server aktif di 127.0.0.1:1337');
```

```
});

server.on('error', function (e) {

  if (e.code == 'EADDRINUSE') {
    console.log('Error: host dan port sudah digunakan.');
  }

});
```

06. Asynchronous / *Non-blocking* IO

Asynchronous input/output merupakan suatu bentuk pemrosesan masukan/keluaran yang memungkinkan pemrosesan dilanjutkan tanpa menunggu proses tersebut selesai. Saat pemrosesan masukan/keluaran tersebut selesai, hasil akan diberikan ke suatu fungsi. Fungsi yang menangani hasil pemrosesan saat pemrosesan tersebut selesai disebut *callback* (pemanggilan kembali). Jadi, mekanismenya adalah: proses masukan/keluaran - lanjut ke alur berikutnya - panggil kembali fungsi pemroses jika proses masukan/keluaran sudah selesai. Setelah spesifikasi ES6 selesai, *callback* bukan satu-satunya cara untuk *non-blocking IO* ini.

Callback

Perhatikan contoh kode sumber dengan menggunakan teknik *blocking I/O* berikut ini.

```
// synchronous.js
var fs = require('fs');
var sys = require('sys');

sys.puts('Mulai baca file');
data = fs.readFileSync('./pegawai.json', "utf-8");
console.log(data);
sys.puts('Baris setelah membaca file');

// hasil:
//Mulai baca file
//{
//  "pegawai": [
//    {
//      "id": "1",
//      "nama": "Zaky",
//      "alamat": "Purwomartani"
```

```
//    },
//    {
//      "id": "2",
//      "nama": "Ahmad",
//      "alamat": "Kalasan"
//    },
//    {
//      "id": "3",
//      "name": "Aditya",
//      "alamat": "Sleman"
//    }
//  ]
//}
//
//Baris setelah membaca file
```

Perhatikan perbedaan dengan di bawah ini:

```
// asynchronous-callback.js
var fs = require('fs');
var sys = require('sys');

// fs.readFile(file[, options], callback)
// file <string> | <Buffer> | <integer> filename or file descriptor
// options <Object> | <string>:
//      encoding <string> | <null> default = null
//      flag <string> default = 'r'
// callback <Function>

sys.puts('Mulai baca file');
fs.readFile('./pegawai.json', "utf-8",  function(err, data) {
  if (err) throw err;
  console.log(data);
})
sys.puts('Baris setelah membaca file');

// hasil:
//Mulai baca file
//Baris setelah membaca file
//{
//  "pegawai": [
//    {
```

```
//      "id": "1",
//      "nama": "Zaky",
//      "alamat": "Purwomartani"
//    },
//    {
//      "id": "2",
//      "nama": "Ahmad",
//      "alamat": "Kalasan"
//    },
//    {
//      "id": "3",
//      "name": "Aditya",
//      "alamat": "Sleman"
//    }
//  ]
//}
```

Kode sumber yang ke dua adalah kode sumber dengan menggunakan teknik *callback*. Teknik ini masih digunakan meskipun disarankan untuk tidak menggunakan *callback* jika proyek yang dikerjakan adalah proyek baru. Hal ini disebabkan karena *callback* membuat kode sumber susah dipahami dan susah di-*maintain*, sehingga pada spesifikasi ES6 dan ES7 muncul *promises* dan *async/await*.

Promise

Dengan *promise*, asynchronous I/O dikerjakan dalam fungsi dan *Promise* dikembalikan oleh fungsi tersebut. Kata kunci *.then* akan digunakan untuk memeriksa hasil, jika ada *promise* (janji) yang "tidak ditepati" maka hal tersebut akan ditangkap di *.catch*. Contoh berikut ini akan membaca semua file dengan ekstensi .txt menjadi .txtp.

```
// promise.js
var fs = require('fs');

if (process.argv.length <= 2) {
    console.log("Usage: " + __filename + " path/to/directory");
    process.exit(-1);
}

var path = process.argv[2];

function readDirContents() {
    return new Promise(
```

```
        function(resolve, reject) {
            fs.readdir(path, function(err, list) {
                if (err) {
                    reject(err);
                } else {
                    resolve(list);
                }
            })
        }
    )
}

readDirContents()
    .then(list => {
        for (var i=0; i<list.length; i++) {

            var fullName = path + '/' + list[i];
            var newFullName = fullName + '.txtp';

            fs.rename(fullName, newFullName, (err) => {
                if (err) throw err;
            });
        }
    })
    .catch(err => { console.log(err) });
```

Async/Await

Async/await digunakan untuk membuat kode sumber lebih terbaca. Untuk keperluan itu, *Async/Await* digabungkan dengan *Promise*.

```
// async-await.js
var fs = require('fs');

if (process.argv.length <= 2) {
    console.log("Usage: " + __filename + " path/to/directory");
    process.exit(-1);
}

var path = process.argv[2];

function readDirContents(thePath) {
```

```
    promise = new Promise(function(resolve, reject) {
        fs.readdir(path, function(err, list) {
            if (err) {
                reject(err);
            } else {
                resolve(list);
            }
        });
    });

    return promise

}

var a = main();

async function main() {

    var rdir = await readDirContents(path)
        .then(list => {
            for (var i=0; i<list.length; i++) {
                console.log(list[i]);
            }
        })
        .catch(err => { console.log(err) });

    return rdir;

};
```

Generators

Generator merupakan suatu obyek yang digunakan untuk merepresentasikan *sequences*. Obyek tersebut dihasilkan oleh *generator function*. Suatu *generator function* merupakan suatu fungsi dengan tanda *asterisk* di karakter terakhir dari awal nama fungsi.

```
function *genWithParam(x) {
    x++;
    yield x
    yield x/2;
}
```

```
var gwp = genWithParam(5);
console.log(gwp.next());
console.log(gwp.next());
console.log(gwp.next());

function *genNoParam() {
    var a = [1,2,3]
    yield a[0]
    yield a[1]
    yield a[2]
}

var gnp = genNoParam();
console.log(gnp.next('a'));
console.log(gnp.next('b'));
console.log(gnp.next());
console.log(gnp.next());

// results:
// { value: 6, done: false }
// { value: 3, done: false }
// { value: undefined, done: true }
// { value: 1, done: false }
// { value: 2, done: false }
// { value: 3, done: false }
// { value: undefined, done: true }
```