Modul praktikum - Minggu 12 - *JavaScript Standard Library*

Dosen pengampu: **Henokh Lugo Hariyanto**
Asisten mata kuliah: **Jein Ananda - (10221031)**; **Muhammad Aulia Rahman - (10221055)**

Tujuan:

- Mampu memahami beberapa *standard library* dalam JavaScript
- Mampu memahami *regular expression* dalam JavaScript
- Mampu menggunakan *Dates and Times standard library*

> Tips belajar bahasa pemrograman adalah mengetik ulang perintah yang kita temukan di buku atau di internet, lalu kita ubah-ubah untuk menguji pemahaman kita sudah tepat atau belum. Faktor bermain-main dan eksplorasi sangat diperlukan untuk memahami setiap perintah bahasa pemrograman yang kita pelajari. Setiap potongan kode di bawah dapat ditulis dalam berkas `.js` lalu dapat di-*running* dengan Node.js.

Di minggu ini kita akan membahas beberapa *standard library* atau bisa dibilang *function* dan *class* yang tersedia secara *default* dalam bahasa pemrograman JavaScript

Ada beberapa *library* yang akan kita bahas.

- Sets and Maps
- Regular Expression
- Dates and Times
- Timer

Library yang disebutkan diatas hanyalah segelintir dari berbagai macam *standar library* dan *built-in object* dalam JavaScript. Lebih lengkap dapat melihat dokumentasi di Mozilla Developer Network (MDN) Web Docs tentang Web APIs dan Standard built-in objects

Di sesi ini lebih ditekankan pada penggunaan *regular expression* yang memiliki kegunaan untuk melakukan pencarian string secara efisien. *Library* yang lain hanya diulas sedikit.

Sets and Maps

Sets merupakan dasar di JavaScript yang digunakan untuk mewakili objek matematis himpunan (ingat kembali di mata kuliah matematika diskrit terkait himpunan). Secara singkat yang menjadi ciri khas objek *Sets* ini adalah seluruh elementnya idak memperboleh ada yang duplikat. Bisa dikatakan *Sets* merupakan bentuk khusus dari *Array* yang mengharuskan elementnya tidak ada yang duplikat/ganda.

Empat hal yang akan kita jelajahi dalam objek *Sets* ini adalah

- membuat objek *Sets*
  create-set.js

```
let s = new Set();         // Membuat set s tanpa element/anggota
let t = new Set([1, s])    // Membuat set t dengan dua anggota

// Membuat set u dari string
let unique_str = new Set("Balikpapan");   // huruf a dan p akan dihitung
sekali
```

- memanipulasi objek *Sets*
  manipulate-set.js

```
let s = new Set();     // Membuat set s tanpa element
console.log(s);

s.add(1);              // Menambahkan element 1
console.log(s);

s.add(1);              // Menambahkan element yang sama tidak mengubah apapun
console.log(s);

s.add(true);           // Menambahkan element selain 1, yaitu true
console.log(s);

s.add([1, 2, 3]);      // Menambahkan element berupa array [1,. 2, 3]
console.log(s);

s.delete(1);           // Menghapus element 1
console.log(s);

s.delete("test")       // Menghapus element yang tidak ada, tidak mengubah
apapun
console.log(s);

s.delete(true);        // Menghapus element true
console.log(s)

s.delete([1, 2, 3])    // Menghapus array [1, 2, 3] gagal karena array yang
dicari dan yang ada di set s memiliki referensi yang berebeda
console.log(s);

s.clear();              // Menghapus semua element set s
```

- menguji keanggotaan suatu element dalam *Sets*
  membership-set.js

```
let result;
let oneDigitPrimes = new Set([2, 3, 5, 7]);

result = oneDigitPrimes.has(2);   // 2 ada di dalam set oneDigitPrimes
```

```
console.log(result);

result = oneDigitPrimes.has(4);   // 4 tidak ada di dalam set oneDigitPrimes
console.log(result);
```

- Melakukan iterasi element *Sets*
  Sepertinya halnya *array*, iterasi juga dapat dilakukan ke masing-masing element dari *Sets*
  iterate-set.js

```
let result;

// Membuat set yang tersusun dari 4 buah bilangan prima berdigit tunggal
let oneDigitPrimes = new Set([2, 3, 5, 7]);

// Menghitung jumlahan keseluruhan element oneDigitPrimes
let sum = 0;
for (let p of oneDigitPrimes) {
  sum += p;
}
console.log(sum)

// Spread operator dapat juga diterapkan untuk menjadikan
// set sebagai array
result = [...oneDigitPrimes];
console.log(result instanceof Array);
```

Objek berikutnya adalah *Map*. Secara garis besar, *Map* merupakan perluasan/perumuman dari objek *array*. Objek *Map* ini sangat mirip dengan objek *dict* pada Python. Di dalam objek *Map* ini tersusun atas serangkaian pasangan *key* dan *value*. Bentuk pasangan *key* dan *value* ini hampir mirip seperti *property name* dan *property value* pada JavaScript object. Namun perbedaan mendasar dari *Map* dan *object* adalah, *map* digunakan untuk bentuk data yang lebih spesifik untuk menyimpan nilai-nilai yang akan kita akses dengan *key*. Sedangkan *object* merupakan struktur data lebih luas daripada *Map*, dan *object* dapat memuat fungsi untuk melakukan manipulasi property yang dimilikinya. Meskipun di dalam *Map* kita juga dapat menempatkan fungsi, namun dalam prakteknya kita tidak menggunakan *Map* seperti *object*.

Sepertinya halnya *Set*, berikut akan diberikan beberapa hal dasar yang dapat dilakukan dengan *Map*

- membuat object *Map*
  create-map.js

```
let m1 = new Map();      // membuat Map kosong
console.log(m1);

let m2 = new Map([       // membuat Map dengan array N x 2
  ["one", 1] ,
  ["two", 2] ]);
console.log(m2);
```

```
// Membuat Map dari Map lain (menggandakan)
let m3 = new Map(m2);
console.log(m3);

// Membuat Map dari object
let obj = {x: 1, y: 2};
let m4 = new Map(Object.entries(obj));
console.log(m4);
```

- memanipulasi object *Map* dan *method* yang terdapat di dalam *map*

manipulate-map.js

```
// Membuat map kosong yang akan kita lakukan manipualsi
let m = new Map();
console.log(m) ;

m.set("one", 1);    // mengisi map dengan pasangan key, val = "one", 1
m.set("two", 2);    // mengisi map dengan pasangan key, val = "two", 2
m.set("three", 3).set("four", 4);   // pengisian map dapat dilakukan dengan
bentuk rantai .set berulang-ulang
console.log(m);

// Melihat ukuran map (banyaknya pasangan key dan val)
console.log(m.size);

// Mendapatkan value dari key tertentu
console.log(m.get("two"));

// Jika key tidak ada di dalam Map, maka akan dihasilkan undefined
console.log(m.get("five"));

// Kita dapat menghapus pasangan key, val dengan method .delete
m.delete("one");
console.log(m);

// Untuk menghapus seluruh pasangan key, val gunakan .clear
m.clear();
console.log(m);
```

- Menguji ada tidaknya *key* dalam object *Map*

membership-map.js

```
let m = new Map([
  ["one", 1],
  ["two", 2],
  ["three", 3],
  ["four", 4] ]);
```

```
console.log(m.has("three"));     // menghasilkan true, karena map m memuat
key "three"
console.log(m.has("five"));      // menghasilkan false, karena map m tidak
emmuat key "five"
```

- Melakukan iterasi pasangan *key*, *val* di dalam *Map*
  iterate-map.js

```
// Membuat map berisi 4 pasangan key, val
let m = new Map([
  ["x", 1],
  ["y", 2],
  ["z", 3],
  ["w", 4] ]);
console.log(m);

// Menyatakan Map sebagai array dengan dimensi 4 x 2
console.log([...m]);

// Mengambil key saja dan mengubah menjadi array
console.log([...m.keys()]);

// Mengambil value saja dan mengubah menjadi array
console.log([...m.values()]);

// Mengambil pasangan key, val (sama seperti [...m])
console.log([...m.entries()])

// Iterasi dapat dilakukan menggunakan for ... of ...
for (let [key, val] of m) {
  console.log(`[key, val]: [${key}, ${val}]`);
}
```

Pattern Matching with Regular Expression

Gagasan *pattern matching* merupakan mekanisme untuk mencari suatu substring menggunakan pola tertentu. Pola tertentu ini dapat dibuat dengan struktur literal yang dikenal *regular expression*.

Di dalam JavaScript, struktur regular expression dapat dibuat dengan mengapit *pattern* dengan `/` dan `/`

```
let pattern = /s$/i
```

Di dalam potongan *regular expression* di atas, *pattern* diberikan oleh `s$`. Huruf `i` setelah `/` adalah penanda bagaimana *pattern* tersebut digunakan saat melakukan *pattern matching*. Kita akan memahami di bagian selanjutnya bahwa huruf `s` adalah karakter dasar (*literal character*), `$` adalah *special character* yang menunjukan bahwa proses *pattern matching* dilakukan di akhir string sebelum karakter *newline* (`\n`).

Bagian terakhir setelah tanda garis miring kedua, yaitu huruf `i` merupakan penanda bahwa proses *pattern matching* dilakukan untuk semua karakter huruf kapital atau kecil (*uppercase* atau *lowercase*). `i` disini merupakan singkatan dari `ignore case`.

Sehingga apabila digabung kita memiliki keseluruhan *regular expression* `/s$/i` untuk mencari string apapun yang berakhiran huruf `s` atau `S`. Contoh: `"pass"`, `"special characters"`, `"SaaS"`. `KT 2023 PAS`.

Dikarenakan luasnya topik *regular expression*, maka disini kita hanya membahas hal-hal yang sangat mendasar dan penting dan dapat digunakan untuk penerapan pengembangan web.

Di bagian ini akan kita uraikan dua bagian besar untuk memahami *regular expression*. Beberapa hal dapat dilompati misal dirasa terlalu panjang selama sesi praktikum.

Bagian pertama akan membahas 7 subbagian terkait *regular expression* yaitu:

1. Pola dasar *regular expression*
   Disini tidak akan didaftarkan satu-satu semua pola dasar namun hanya beberapa saja sehingga cukup memberikan gambaran.

   - `.` (karakter titik): semua jenis karakter kecuali `\newline`.
     Pattern ini digunakan untuk melakukan pencarian semua jenis karakter kecuali `\newline` (`\newline` adalah karakter yang kita input ketika menekan tombol `Enter`). Contoh penggunaannya `/./`

   - karakter satu huruf (contoh: `a`):
     Pattern ini akan mencari karakter yang dispesifikasi (misal `a`) dalam suatu string. Contoh penggunaannya `/a/`

   - karakter lebih dari satu huruf (contoh: `ab`):
     Pattern ini akan mencari karakter yang dispesifikasi (misal `ab`)
     dalam suatu string. Contoh penggunaannya `/ab/`

   - conditional character dengan `|` (contoh: `a|b`):
     Pattern ini akan mencari karakter yang dispesifikasi sebelum `|` atau sesudah `|`. Contoh misal kita menuliskan *pattern* `a|b` maka akan dicari karakter `a` atau `b` dalam suatu string. Contoh penggunaannya `a|b`.

2. Quantifiers
   Quantifier digunakan untuk menyatakan berapa kali karakter yang ingin kita cari muncul.

   - `*` (tanda bintang, *asterisk*)
     Quantifier ini digunakan untuk menyatakan bahwa karakter yang kita cari memiliki nol atau lebih perulangan karakter. Misal kita memiliki *pattern* `/a*/`, maka akan dicari *strings* berikut `""`, `"a"`, `"aa"`, `"aaa"`, `"aaaa"`, dan seterusnya.

   - `+` (tanda positif, *plus sign*)
     Quantifier ini digunakan untuk menyatakan bahwa karakter yang kita cari memiliki satu atau lebih perulangan karakter. Misal kita memiliki *pattern* `/a+/`, maka akan dicari *strings* berikut `"a"`, `"aa"`, `"aaa"`, `"aaaa"`, dan seterusnya.

- `?` (tanda tanya, *question mark*) *Quantifier* ini digunakan untuk menyatakan bahwa karakter yang kita cari memiliki perulangan nol atau satu karakter. Misal kita memiliki *pattern* `/a?/` maka akan dicari dua string berikut `""` dan `"a"`

- `{n}`: `n` perulangan karakter
  Quantifier ini digunakan untuk menyatakan bahwa karakter yang kita cari memiliki perulangan tepat `n` kali. Misal kita memiliki `pattern /a{2}/` maka akan dicari satu string `"aa"`.

- `{n,m}`: perulangan karakter dari `n` kali hinga `m` kali.
  Quantifier ini digunakan untuk menyatakan bahwa karakter yang kita cari memiliki perulangan antara `n` kali perulangan dan `m` kali perulangan. Misal kita memiliki *pattern* `/a{2,5}/` maka akan dicari *string* berikut: `"aa"`, `"aaa"`, `"aaaa"`, `"aaaaa"`.

- `{n,}`: perulangan karakter minimal `n` kali
  Quantifier ini digunakan untuk menyatakan bahwa karakter yang kita cari memiliki perulangan minimal `n` kali. Misal kita memiliki *pattern* `/a{2,}` maka akan dicari *string* berikut: `"aa"`, `"aaa"`, `"aaaa"`, `"aaaaa"`, dan seterusnya.

3. Groups
   Groups digunakan untuk mengelompokkan *pattern*

   - `(...)` (tanda kurung)
     Group ini digunakan untuk mengumpulkan beberapa *patterns* menjadi satu unit *pattern*. Sangat berguna ketika digunakan bersamaan dengan *quantifier*. Perlu diingat *group* ini dapat direferensi dengan `\#` (`#` menyatakan urutan *group*). Misal kita memiliki *pattern* `/(a|b)+/` maka akan dicari *string* yang terdiri dari kombinasi satu atau lebih karakter `a` atau `b`. Contohnya: `"a"`, `"b"`, `"ab"`, `"ba"`, `"aaa"`, `"aab"`, `"aba"`, `"baa"`, `"abb"`, `"bab"`, `"bba"`, `"bbb"`, dan seterusnya.

   - `(?:...)` (tanda kurung dan `?:`)
     Sama seperti *Group* `(...)` namun tidak mengingat referensi.

   - `\#` (garis miring balik dan urutan `#`-th *group*)
     Group ini digunakan untuk mereferensi *group* `()` yang sudah digunakan sebelumnya. Sehingga tidak perlu menuliskan dua kali. Misal kita memiliki *pattern* `/(a|b)c(d|e)(a|b)/` dapat disingkat menjadi `/(a|b)c(d|e)\1/`.

4. Flags
   Digunakan untuk mengubah proses *pattern matching*. Sebelumnya telah dibahas *flag* `i` yang menyatakan *pattern matching* mengabaikan huruf besar kecil dari *pattern* yang digunakan. Berikut ini beberapa *flag* yang umum digunakan.

   - `g` (*global match*)
     Flag ini digunakan untuk mencari semua *pattern* yang muncul dalam suatu string. Sehingga tidak hanya kemunculan pertama saja yang dihasilkan. Misalkan kita memiliki *pattern* '/JavaScript/g' maka semua kata `JavaScript` akan dimunculkan sebagai hasil t tidak hanya kata pertama yang ditemukan.

   - `i` (*ignore case*)
     Flag ini digunakan untuk mengabaikan huruf besar kecil dari karakter alfabet. Misal kita memiliki

pattern: '/js/i', maka semua kombinasi besar kecil `j` dan `s` akan dicari oleh *pattern* tersebut, yaitu: `"js"`, `"Js"`, `"jS"`, dan `"JS"`.

- `m` (*multiline mode*)
  Dengan *flag* ini suatu string yang terdiri dari beberapa baris (tiap baris dipisahkan oleh *newline character*) tidak lagi dilihat sebagai satu *string* panjang, melainkan terdiri beberapa *substrings* yang dipisahkan oleh *newline character*. Sehingga apabila digunakan pencarian bersamaan dengan *assertion* `^` atau `$` (lihat penjelasan di bagian *assertion*), akan dilakukan pencarian ditiap baris.

  multiline-mode-regex.js

```
let text = "JavaScript script\nscript";   // dua baris string
let p = /^(script)/;
console.log(text.replace(p, "a")); // => "Javascript script\nscript"
                                   //    (tidak mengubah apapun)

p = /^(script)/m;
console.log(text.replace(p, "a")); // => "Javascript script\na"

p = /(script)$/;
console.log(text.replace(p, "a")); // => "Javascript script\na"
                                   // `text` dianggap string panjang

p = /(script)$/m;
console.log(text.replace(p, "a")); // => "Javascript a\nscript"
                                   // `text` terdiri dari substring
```

5. Character classes
   Merupakan gabungan dari karakter individu yang membentuk kelas karakter. Sangat berguna untuk mendaftar semua karakter menjadi lebih ringkas. Seperti contohnya jika ingin melakukan pencarian *pattern* alfabet dari `a` sampai `z`, kita tidak perlu menuliskan `/abcdefghijklmnopqrstuvwxyz/` tetapi cukup dengan `/[a-z]/` Berikut adalah *character class* yang ada di JavaScript:

   - `[...]`: semua karakter yang ada dalam kurung siku.
     Misal kita memiliki contoh *pattern* `/[ab-d]/` maka *string* yang dicari adalah karakter `a`, `b`, `c`, `d`.

   - `[^...]`: semua karakter yang yang bukan di dalam kurung siku.
     Misal kita memiliki contoh pattern `/[^ab-d]/` maka *string* yang dicari adalah karakter selain empat karakter `a`, `b`, `c`, `d`.

   - `[\b]` (*backspace character*)
     Character class ini akan mencari *backspace character*. Di dalam kurung siku hanya boleh ada `\b`. Jika disisipkan huruf lain maka *character class* ini tidak lagi menunjukkan *backspace*. *Character class* ini sangat jarang digunakan dan merupakan warisan dari pengembangan Unicode.

     Contoh penggunaannya

     backspace-regex.js

```
// \b (`backspace character` memindahkan kursor ke kiri)
// namun tidak menghapus karakter selama pergerakan kursor ke kiri
let text = "JavaScript\b\b\b\b\b\b s";
console.log(text);    // => Java sript;
let p = /[\b]+ s/;
console.log(text.replace(p, ""));   // => JavaScript
```

- `\d` (*digit*)
  Character class ini menyatakan *pattern* untuk angka. Setara dengan *pattern* `[0-9]`.
- `\D` (*non-digit*)
  Character class ini menyatakan *pattern* selain angka. Setara dengan *pattern* `[^0-9]`
- `\s` (*whitespace character*)
  Character class ini menyatakan *pattern* untuk Unicode *whitespace character*. Contohnya: spasi, *Enter*, dan *Tab*.
- `\S` (*non-whitespace character*)
  Character class ini menyatakan *pattern* selain *whitespace character*.
- `\w` (*word character*)
  Character class ini menyatakan semua ASCII *word character*. *Word character* adalah karakter alfabet (huruf besar dan kecil), angka dan *underscore*. Setara dengan *pattern* `[a-zA-Z0-9_]`.
- `\W` (*non-word character*)
  Character class ini menyatakan semua karakter selain ASCII *word character*

6. Special character
   Di bagian ini merupakan kumpulan *pattern* untuk *special character* seperti karakter baris baru (diinput ketika menekan tombol *Enter*) .

   - `\n` (*newline*)
     Special character untuk mencari karakter pemisah antar baris. Karakter ini dapat digunakan dalam *regular expression* untuk memisahkan tiap baris dalam suatu string.
   - `\r` (*carriage return*)
     Special character untuk membawa kursor ke posisi awal *string*.
   - `\t` (*tab*)
     Special character yang mewakili tombol *Tab*. Ukuran tab dari tanpa spasi hingga umumnya 4 spasi berdasarkan posisi ke kelipatan terdekat.

     tab-example-regex.js

     ```
     let p = /\t/;
     let text = "abcd\tefgh";   // \t disini setara dengan 4 spasi
     console.log(text);
     console.log(text.replace(p, "+"));
     ```

```
text = "abcdefg\tefgh";    // \t disini setara dengan 1 spasi
console.log(text);
console.log(text.replace(p, "+"));
```

- `\nnn` (*octal character*)
  Unicode *character* dengan menggunakan *octal digit*. Misal huruf kapital `E` dapat dinyatakan dalam `\105`.

- `\xhh` (*latin character* dengan dua digit heksadesimal)
  Dua digit heksadesimal yang digunakan untuk menyatakan karakter latin (huruf A-Z). Misal huruf kapital `E` dapat dinyatakan dalam `\x45`.

- `\uhhhh` (Unicode *character* dengan empat digit heksadesimal)
  Empat digit heksadesimal yang digunakan untuk menyatakan Unicode *character*.
  Misal *parallel world* dalam bahasa Jepang 異世界 (*isekai*) dapat dinyatakan dalam `\u7570\u4e16\u754c`.

7. Assertion
   Assertion merupakan karakter yang digunakan untuk menyatakan bahwa *pattern matching* mengacu (*anchoring*) ke posisi tertentu dalam suatu *string*. Berikut karakter yang digunakan:

   - `^` (pencarian dimulai dari karakter awal *string*)
     Assertion ini melakukan pencarian karakter dimulai dari awal karakter suatu string. Jika dikombinasikan dengan *flag* `m` maka akan dilakukan pencarian juga di awal baris. (lihat contoh sebelumnya di bagian *Flag*)

   - `$` (pencarian dimulai dari karakter akhir *string*)
     Assertion ini melakukan pencarian karakter dimulai dari akhir karakter suatu string. Jika dikombinasikan dengan *flag* `m` maka akan dilakukan pencarian juga di akhir baris. (lihat contoh sebelumnya di bagian *Flag*)

   - `\b` (pencarian dilakukan di antara *word boundary*)
     Assertion ini akan melakukan pencarian diantara `\w` dan `\W` (lihat bagian *character classes*). *Boundary* disini adalah bagian diantara dua karakter. Jika umumnya *anchoring* dilakukan pada karakter (seperti `^` dan `$`), `\b` melakukan *anchoring* diantara dua karakter. Misal kita melakukan pencarian kata "JavaScript" yang berdiri sendiri (kanan kiri dipisahkan spasi). Kita dapat menggunakan *pattern* `/\sJavaScript\s/g` (`\s` disini adalah karakter spasi dan *flag* `g` untuk mode *global*). Namun cara ini tidak dapat diterapkan apabila kata "JavaScript" muncul di awal string (sebelum "JavaScript" tidak dimulai dengan spasi) atau di akhir string (setelah "JavaScript" tidak diakhiri oleh spasi). Maka dengan *word boundary* kita dapat mengatasi masalah tersebut dan mengubahnya menjadi `/\bJavaScript\b/g`

   - `\B` (pencarian dilakukan di antara *non-word boundary*)
     Sama seperti `\b` hanya saja melakukan pencarian *pattern* yang bukan *word boundary*. Sebagai contoh *pattern* `/\B[Ss]cript/` dapat ditemukan didalam *string* "JavaScript" dan "postscript", namun tidak untuk "script" atau "Scripting* yang merupakan kata yang berdiri sendiri (bukan bagian dari suatu kata).

- `(?=p)` (*positive lookahead*)
  Assertion ini digunakan untuk mengatur secara bebas karakter *anchoring* `p`. Sebagai contoh kita ingin mencari kata "JavaScript" lalu diikuti oleh titik dua. Maka *pattern*-nya adalah `/JavaScript(?=\:)/g`

- `(?!p)` (*negative lookahead*)
  Assertion ini adalah negasi dari *positive lookahead* yang mengakibatkan *pattern* `p` tidak dipenuhi dalam proses pencarian. Sebagai contoh kita ingin mencari kata "Java" yang diikuti oleh huruf kapital namun tidak diikuti oleh kata "Script". Maka *pattern*-nya adalah `/JavaScript(?!Script)[A-Z]/g`.

Bagian kedua kita akan menerapkan *regular expression* ke dalam *methods* string. Kita bagi dalam subbagian *string method* berikut

- `search()`

  str-method-search.js

  ```
  let result;

  result = "JavaScript".search(/script/i);
  console.log(result);    // => 4 (indeks S di Script)

  result = "Python".search(/script/i);
  console.log(result);    // => -1 (tidak ditemukan pattern /script/)
  ```

- `replace()`

  str-method-replace.js

  ```
  let result;

  const text = "the following text contains incorrect capitalization of
  jaVaScrIPt";

  // No matter how it is capitalized, replace it with the correct
  capitalization
  result = text.replace(/javascript/gi, "JavaScript");
  result = result.replace(/incorrect/gi, "correct");
  console.log(result);
  ```

- `match()`

  str-method-match.js

  ```
  let result;

  const text = "7 plus 8 equals 15";
  ```

```
result = text.match(/\d+/g);    // => ["7", "8", "15"]
console.log(result);

result = text.match(/\d+/);    // => returns an array
                               //    with complete description of
                               //    how its matched to the text
console.log(result);
```

- `matchAll()`

  str-method-mathc-all.js

```
// One or more Unicode alphabetical characters between word boundaries
const words = /\b\p{Alphabetic}+\b/gu;   // \p is not supported in Firefox
yet
const text = "This is a naïve test of the matchAll() method.";
for (let word of text.matchAll(words)) {
  console.log(`Found '${word[0]}' at index ${word.index}.`);
}
```

- `split()`

  str-method-split.js

```
let result;

result = "123,456,789".split(",");
console.log(result);

result = "1, 2, 3,\n4, 5".split(/\s*,\s*/);
console.log(result);

const htmlTag = /<([^>]+)>/;    // < followed by one or more nan ->,
followed by >
result = "Testing<br/>1,2,3".split(htmlTag);    // ["Testing", "br/",
"1,2,3"]
console.log(result);
```

Beberapa sumber yang berguna untuk mendalami penggunaan praktis, eksplorasi, dan pengujian *regular expression* adalah sebagai berikut:

- Buku yang menjelaskan *regular expression* dalam JavaScript
  - (Agarwal, 2021) - *JavaScrip RegExp - an example based guide*
  - (Johansson, 2019) - *Regex Cheat Sheet*
- Visualisasi *regular expression* JavaScript ke dalam *railroad diagram*: jex.im

- Pengujian *regular expression* di JavaScript dan bahasa lainnya: regex-tester
- Beberapa contoh penggunaan *regular expression* dalam kasus nyata (lihat bagian *Community Pattern*): regexr
- *Cheatsheet* some important regular expression: JavaScript RegEx CheatSheet by CodingTute
- 100 seconds explanation by Fireship.io about regular expression: JavaScript RegEx CheatSheet by Fireship.io
- Useful RegEx API in JavaScript (including other JavaScript built-in command and object): overapi.com

Dates and Times

Web API berikutnya adalah pengelolaan variable tanggal dan waktu (*Dates and Times*). Berikut ini adalah contoh penggunaan object `Date` dan beberapa method untuk memanipulasi pencetakan tanggal dan waktu pada *terminal*

date-obj.js

```
function print(s) { console.log(...s); };
let result;

let now = new Date();      // The current time
print(["now: ", now]);

let epoch = new Date(0);  // Midnight, January 1st, 1970, GMT
print(["epoch: ", epoch]);

let century = new Date(2100,          // Year 2100
                       0,             // January
                       1,             // 1st
                       2, 3, 4, 5);   // 02:03:04.005, local time
print(["century: ", century]);

// Midnight in England, January 1, 2100
century = new Date(Date.UTC(2100, 0, 1));
print(["century: ", century]);
print(["century.toString(): ", century.toString()]);
print(["century.toUTCString(): ", century.toUTCString()]);
print(["century.toISOString(): ", century.toISOString()]);

console.log();
century = new Date("2100-01-01T00:00:00Z");   // An ISO format date
print(["century.toISOString(): ", century.toISOString()]);
```

Untuk melakukan aritmatika dengan object *Dates and Time*, kita perlu menggunakan beberapa *method* `get` dan `set` yang sudah disediakan oleh *library Dates and Time*

date-arithmetic.js

```
function print(s) { console.log(...s); };

let d = new Date();
print(["d: ", d]);

d.setMonth(d.getMonth() + 3, d.getDate() + 14);
print(["d: ", d]);

d.setTime(d.getTime() + 30_000);      // add 30 seconds
print(["d: ", d]);
```

Timers

Merupakan fungsi bawaan JavaScript yang berfungsi untuk melakukan jeda sebelum program atau perintah dieksekusi. Berikut contoh penggunaan Timer yang digunakan bersamaan dengan perintah `console.log()`

timer-set-timeout.js

```
setTimeout(() => { console.log("Read..."); }, 1000);
setTimeout(() => { console.log("set..."); }, 2000);
setTimeout(() => { console.log("go!")}, 3000);
```

Berikut ini penggunaan `setTimeout` dan `setInterval` untuk membuat jam digital dengan format waktu lokal.

timer-digital-clock.js

```
// Once a second; clear the console and print the current time
let clock = setInterval(() => {
  console.clear();
  console.log(new Date().toLocaleTimeString());
}, 1000);

// After 10 seconds: stop the repeating code above
setTimeout(() => { clearInterval(clock); }, 10000);


// The above program is an example of asynchronous programming
```

Tugas (Exercise - 08)

> Laporan harus ditulis dan dikumpulkan dalam bentuk berkas *markdown* atau berkas berekstensi `.md`. Apabila laporan memuat lebih dari satu berkas, misal memuat berkas gambar `.png` atau `.jpg`, maka berkas disatukan menjadi berkas `.zip`.
> **PASTIKAN** berkas `md` sudah dilakukan *preview*, sehingga kode *markdown* bisa di-*preview* dengan

> benar.
> Format penamaan file: `NIM_NAMA.md` atau `NIM_NAMA.zip` (boleh nama lengkap atau nama panggilan).
>
> **Contoh format laporan atau jawaban (`NIM_NAMA.md`)**
> Nama: [NAMA LENGKAP]
> NIM: [NIM]
>
> 1. (Jawaban nomor 1)
> 2. (Jawaban nomor 2)

1. [30 poin] Pilih dan telusuri satu Web APIs di daftar yang ada di MDN Web Docs. Berikan salah satu contoh penggunaan yang berhubungan langsung dengan permasalahan organisasi atau manajemen.
2. [70 poin] Berikut ini diberikan data *last access* LMS ITK untuk mahasiswa Sistem Informasi yang mengikuti mata kuliah Pemrograman Terstruktur. Data dapat diunduh di 20230426-1505WITA-last-access.json. Menggunakan *regular expression*, seleksi setiap baris untuk mendapatkan jam dan menit. Menggunakan nama file yang diberikan, tentukan waktu terakhir (tanggal dan jam LMS diakses oleh mahasiswa). Perhitungan waktu bisa menggunakan *library Dates and Times*.