

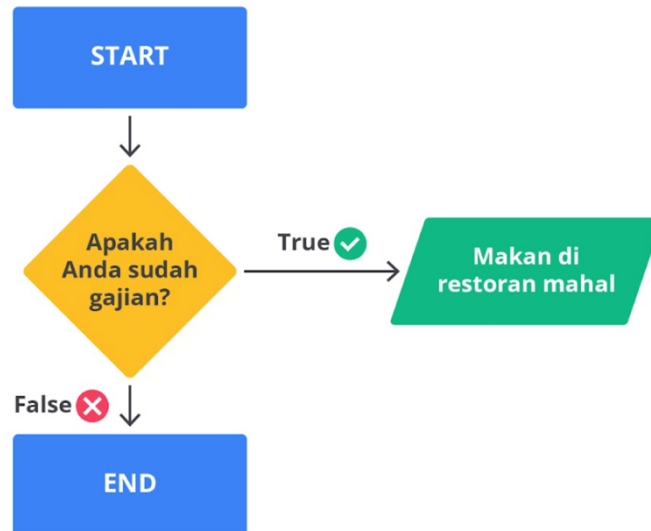


JavaScript

A Game Changer of
Complex Applications

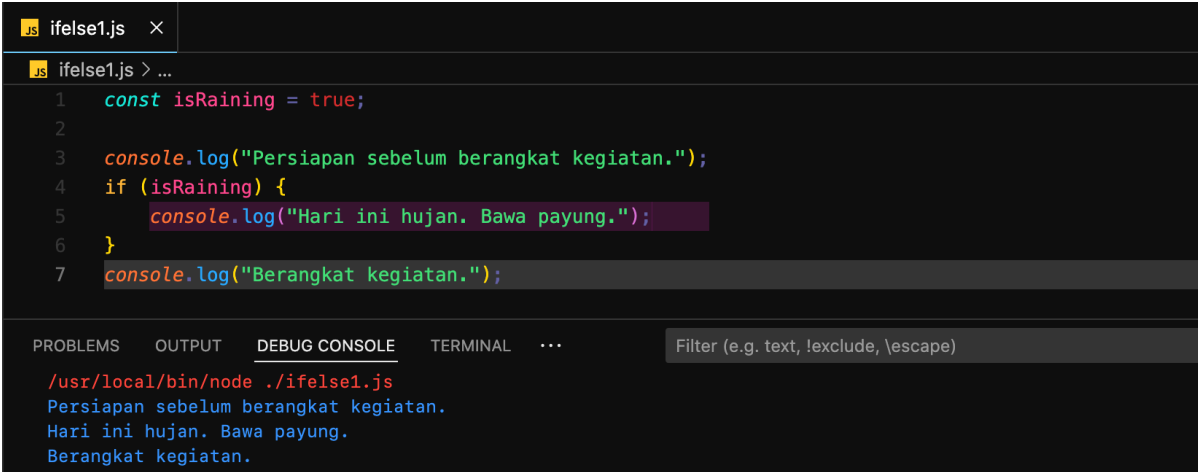
Conditional

Conditional flow adalah cara untuk menentukan apakah kode dieksekusi atau dilewatkan. Jika suatu kondisi terpenuhi, kode akan dieksekusi dan kode yang lainnya akan diabaikan. Kondisi ini ditentukan dari inputan yang diberikan oleh pengguna.



If Statement

If statement merupakan fundamental statement yang memungkinkan JavaScript untuk membuat keputusan apakah mengeksekusi program atau tidak.



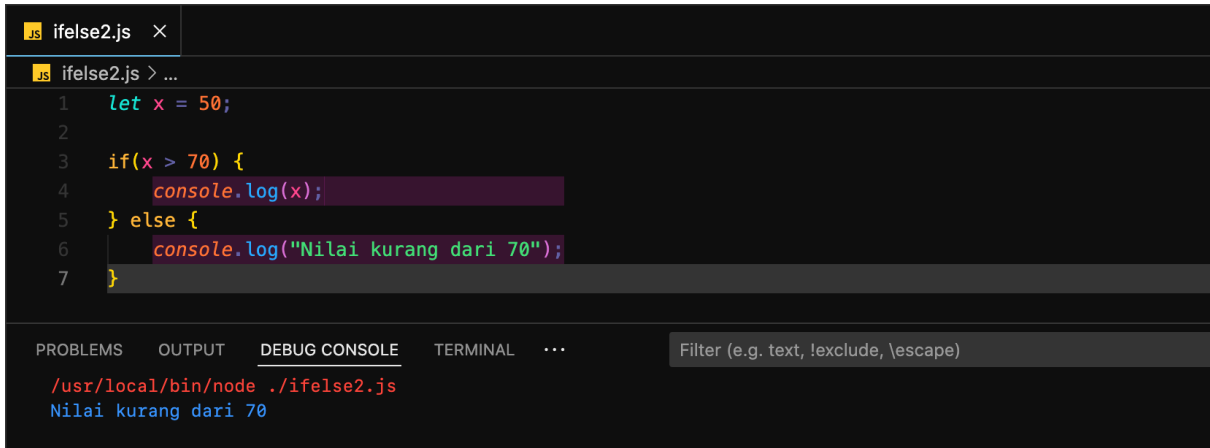
```
ifelse1.js x
ifelse1.js > ...
1  const isRaining = true;
2
3  console.log("Persiapan sebelum berangkat kegiatan.");
4  if (isRaining) {
5      console.log("Hari ini hujan. Bawa payung.");
6  }
7  console.log("Berangkat kegiatan.");
```

PROBLEMS OUTPUT **DEBUG CONSOLE** TERMINAL ... Filter (e.g. text, !exclude, \escape)

```
/usr/local/bin/node ./ifelse1.js
Persiapan sebelum berangkat kegiatan.
Hari ini hujan. Bawa payung.
Berangkat kegiatan.
```

JavaScript

Apakah if statement hanya bisa menangani satu cabang kondisi saja? Jawabannya adalah tidak. Kita bisa menggunakan keyword `else` untuk menambah pengecekan kondisi lainnya. Perhatikan contoh berikut ini.

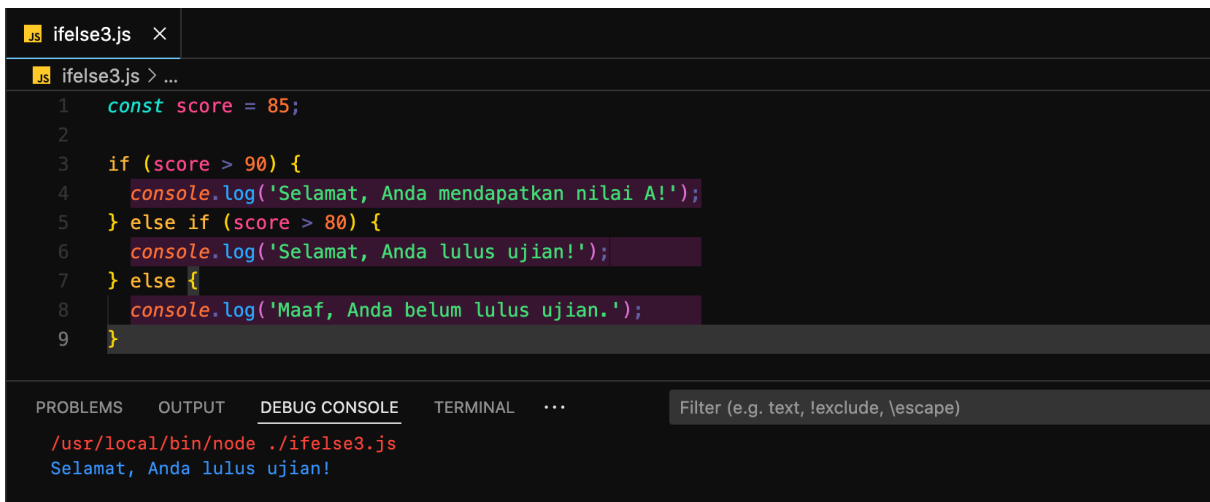


```
1 let x = 50;
2
3 if(x > 70) {
4   console.log(x);
5 } else {
6   console.log("Nilai kurang dari 70");
7 }
```

DEBUG CONSOLE

```
/usr/local/bin/node ./ifelse2.js
Nilai kurang dari 70
```

Lalu, jika memiliki cabang kondisi lebih dari dua, Anda dapat mengecek beberapa kondisi sekaligus dengan menggabungkan `else` dan `if`.



```
1 const score = 85;
2
3 if (score > 90) {
4   console.log('Selamat, Anda mendapatkan nilai A!');
5 } else if (score > 80) {
6   console.log('Selamat, Anda lulus ujian!');
7 } else {
8   console.log('Maaf, Anda belum lulus ujian.');
9 }
```

DEBUG CONSOLE

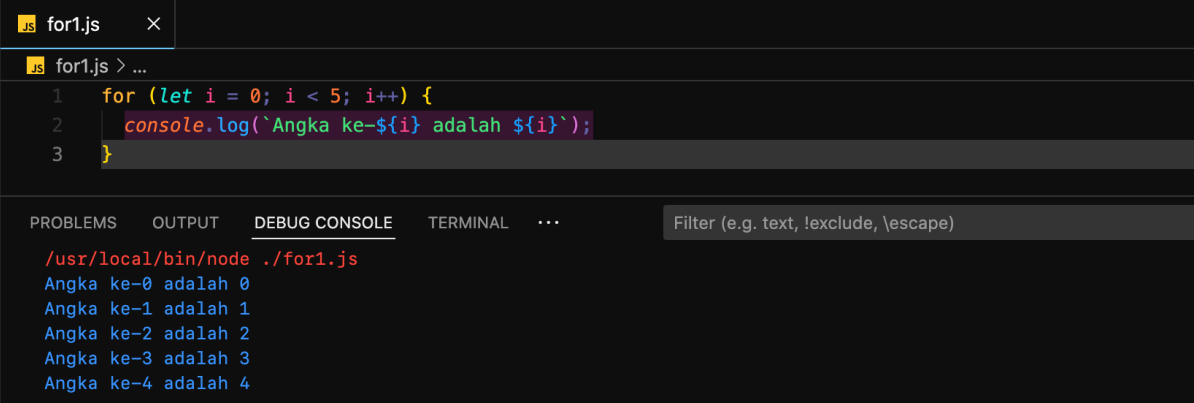
```
/usr/local/bin/node ./ifelse3.js
Selamat, Anda lulus ujian!
```

Looping

Ketika Anda memprogram, ada banyak sekali instruksi yang Anda tulis untuk dieksekusi oleh komputer.

For Loop

Struktur dari for loop tampak seperti berikut ini.

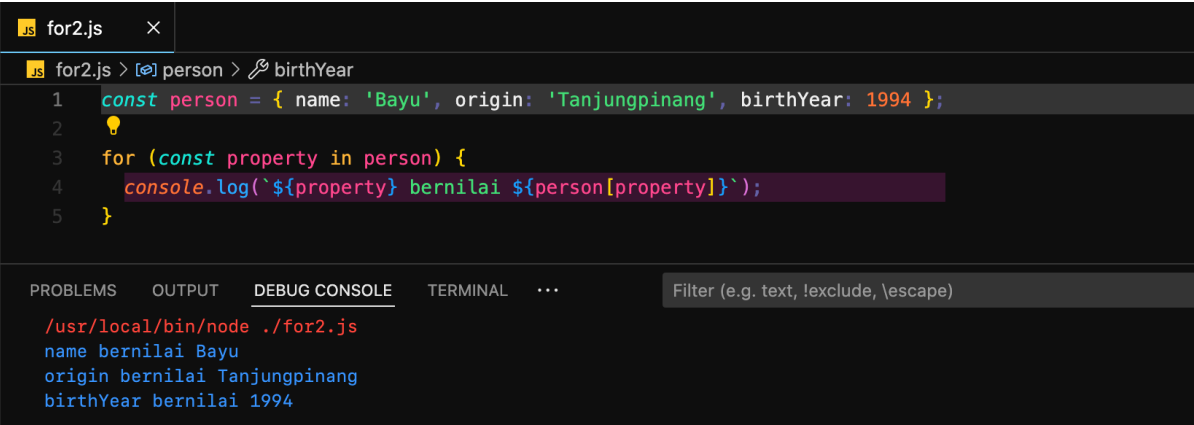


```
JS for1.js x
JS for1.js > ...
1 for (let i = 0; i < 5; i++) {
2   console.log(`Angka ke-${i} adalah ${i}`);
3 }

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL ... Filter (e.g. text, !exclude, \escape)
/usr/local/bin/node ./for1.js
Angka ke-0 adalah 0
Angka ke-1 adalah 1
Angka ke-2 adalah 2
Angka ke-3 adalah 3
Angka ke-4 adalah 4
```

For In

For in banyak digunakan untuk pengulangan pada object karena ia dapat melakukan iterasi ke seluruh data di dalam objek. Bahkan, ia juga dapat melakukan iterasi ke properti *inheritance* dari object seperti *length*. Berikut contoh penggunaan for in.



```
JS for2.js x
JS for2.js > [e] person > birthYear
1 const person = { name: 'Bayu', origin: 'Tanjungpinang', birthYear: 1994 };
2
3 for (const property in person) {
4   console.log(`${property} bernilai ${person[property]}`);
5 }

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL ... Filter (e.g. text, !exclude, \escape)
/usr/local/bin/node ./for2.js
name bernilai Bayu
origin bernilai Tanjungpinang
birthYear bernilai 1994
```

For Of

Kehadiran for of dimulai pada ECMAScript 2015 (ES6). For of berbeda dengan for in. For of lebih sederhana karena kita tidak perlu memikirkan *property* dan *key*. Perhatikan contoh berikut.

JavaScript

```
for3.js x
JS for3.js > ...
1  const names = ['Bebek', 'Ayam', 'Telor', 'Tempe'];
2
3  for (const item of names) {
4      console.log(item);
5  }
```

PROBLEMS OUTPUT **DEBUG CONSOLE** TERMINAL ... Filter (e.g. text, exclude, \escape)

```
/usr/local/bin/node ./for3.js
Bebek
Ayam
Telor
Tempe
```

While

Perulangan di JavaScript tak hanya menggunakan for, tetapi ada cara lainnya yaitu *while statement*.

```
while1.js x
JS while1.js > ...
1  let i = 0;
2
3  while (i < 5) {
4      console.log(`Angka ke-${i} adalah ${i}.`);
5      i++;
6  }
```

PROBLEMS OUTPUT **DEBUG CONSOLE** TERMINAL ... Filter (e.g. text, exclude, \escape)

```
/usr/local/bin/node ./while1.js
Angka ke-0 adalah 0.
Angka ke-1 adalah 1.
Angka ke-2 adalah 2.
Angka ke-3 adalah 3.
Angka ke-4 adalah 4.
```

Control Statement

Ketika melakukan perulangan, ada yang namanya control statement. Control statement berfungsi untuk menghentikan eksekusi kode. Beberapa statement yang masuk ke dalam kategori control statement adalah *break* dan *continue*.

Break

Break statement adalah cara kita untuk memberitahukan interpreter yang sedang mengeksekusi kode untuk berhenti dan langsung berpindah ke akhir dari percabangan atau perulangan.



```
controlstatement1.js
controlstatement1.js > ...
1  for (let i = 0; i < 10; i++) {
2    if (i === 5) {
3      break;
4    }
5
6    console.log(i);
7  }
```

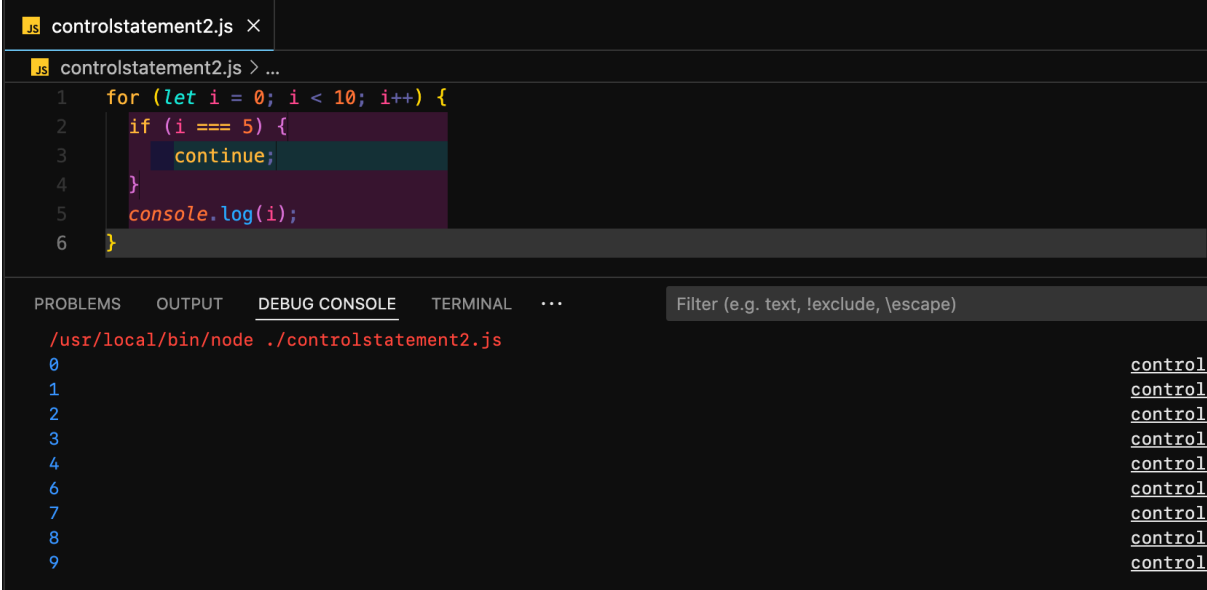
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL ... Filter (e.g. text, !exclude, \escape)

```
/usr/local/bin/node ./controlstatement1.js
0 control
1 control
2 control
3 control
4 control
```

Kode di atas akan mencetak angka 0 hingga sepuluh kali, tetapi akan terhenti ketika nilai variabel *i* sama dengan 5. Hal ini disebabkan oleh adanya statement *break*. Break akan menghentikan proses perulangan.

Continue

Continue statement sama seperti break statement. Namun, alih-alih menghentikan eksekusi program, continue akan melanjutkan iterasi ke iterasi berikutnya. Continue statement hanya dapat digunakan di dalam body looping. Perhatikan contoh berikut ini.



```
1 for (let i = 0; i < 10; i++) {
2   if (i === 5) {
3     continue;
4   }
5   console.log(i);
6 }
```

DEBUG CONSOLE

```
/usr/local/bin/node ./controlstatement2.js
0
1
2
3
4
6
7
8
9
```

control
control
control
control
control
control
control
control
control

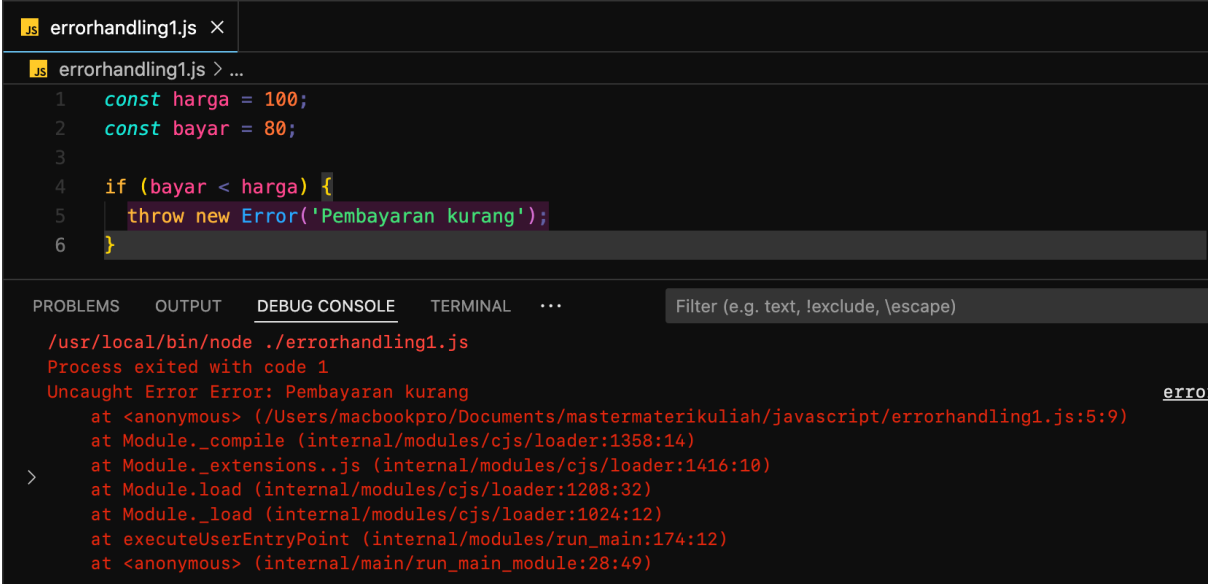
Looping akan berjalan seperti biasa. Namun, ketika nilai variabel `i` sama dengan 5, iterasi akan dihentikan dan lanjut ke iterasi berikutnya sehingga di terminal/console tidak akan menampilkan angka 5.

Error Handling

Error yang dibiarkan dan tidak ditangani akan menyebabkan *crash* pada program yang dibangun. JavaScript memiliki cara untuk menangani error tersebut yang disebut dengan error handling. Error handling dapat mencegah *crash* pada program ketika terjadi error yang disebabkan oleh kesalahan syntax atau error lainnya.

Throwing Error

"Throwing error" adalah istilah yang sering digunakan dalam pemrograman untuk merujuk pada proses "melempar" atau memunculkan suatu kesalahan (error) ketika ada masalah yang terdeteksi dalam kode program.



```
errorhandling1.js
1  const harga = 100;
2  const bayar = 80;
3
4  if (bayar < harga) {
5    throw new Error('Pembayaran kurang');
6  }
```

PROBLEMS OUTPUT **DEBUG CONSOLE** TERMINAL ... Filter (e.g. text, !exclude, \escape)

```
/usr/local/bin/node ./errorhandling1.js
Process exited with code 1
Uncaught Error Error: Pembayaran kurang
    at <anonymous> (/Users/macbookpro/Documents/mastermaterikuliah/javascript/errorhandling1.js:5:9)
    at Module._compile (internal/modules/cjs/loader:1358:14)
    at Module._extensions..js (internal/modules/cjs/loader:1416:10)
    at Module.load (internal/modules/cjs/loader:1208:32)
    at Module._load (internal/modules/cjs/loader:1024:12)
    at executeUserEntryPoint (internal/modules/run_main:174:12)
    at <anonymous> (internal/main/run_main_module:28:49)
```

Try-Catch

Try-catch merupakan cara yang dimiliki JavaScript untuk menangani error. Try-catch memiliki dua blok utama yaitu try dan catch. Try merupakan blok kode yang akan menangani error, sedangkan catch merupakan blok kode yang dibangkitkan ketika terjadi error di dalam blok try.

```
JS errorhandling2.js X
JS errorhandling2.js > ...
1  try {
2    // Kode yang mungkin akan menimbulkan error
3    let result = 10 / 0;
4    console.log("Hasil: " + result);
5
6    // Mencoba memanggil variabel yang belum didefinisikan (ini akan menyebabkan error)
7    console.log(undifinedVariable);
8
9  } catch (error) {
10   // Blok ini akan menangkap error yang terjadi di dalam blok try
11   console.log("Terjadi error: " + error.message);
12 }

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  ...  Filter (e.g. text, lexclude, \escape)
/usr/local/bin/node ./errorhandling2.js
Hasil: Infinity
Terjadi error: undifinedVariable is not defined  erro
error
```

Finally

Finally adalah blok kode yang berada di akhir try-catch. Bilamana catch dieksekusi hanya ketika ada error di dalam blok try, blok yang ada di finally akan selalu dieksekusi. Simak contoh di bawah ini.

```
JS errorhandling3.js X
JS errorhandling3.js > ...
1  try {
2    //let nilai = 85;
3    //let nilai = 150;
4    let nilai = 'A';
5
6    if (typeof nilai !== 'number') {
7      // Jika nilai bukan angka, lempar error
8      throw new Error("Nilai harus berupa angka");
9    }
10   if (nilai < 0 || nilai > 100) {
11     // Jika nilai di luar batas yang diizinkan, lempar error
12     throw new Error("Nilai harus antara 0 dan 100");
13   }
14   console.log("Nilai valid: " + nilai);
15 } catch (error) {
16   // Menangkap dan menampilkan error jika ada
17   console.log("Terjadi error: " + error.message);
18 } finally {
19   // Blok ini akan selalu dijalankan, baik ada error maupun tidak
20   console.log("Proses validasi selesai.");
21 }
```

Output :

let nilai = 85;

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  ...  Filter (e.g. text, !exclude, \escape)

/usr/local/bin/node ./controlstatement4.js
Nilai valid: 85
Proses validasi selesai.
```

let nilai = 150;

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  ...  Filter (e.g. text, !exclude, \escape)

/usr/local/bin/node ./controlstatement4.js
Terjadi error: Nilai harus antara 0 dan 100
Proses validasi selesai.
```

let nilai = 'A';

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  ...  Filter (e.g. text, !exclude, \escape)

/usr/local/bin/node ./controlstatement4.js
Terjadi error: Nilai harus berupa angka
Proses validasi selesai.
```