Validering och strukturerad felhantering

Kort om reguljära uttryck, kommandoradargument och avsluta med felkod

Hur kontrollera att en egenskaps värde är formaterat på rätt sätt?

```
// address.js
export class Address {
  constructor (addressLine, postalCode, city) {
    this.addressLine = addressLine
    this.postalCode = postalCode
    this.city = city
  }

  toString () {
    return `${this.addressLine}\n${this.postalCode} ${this.city}`
  }
}

// app.js
import { Address } from './address.js'

const address1 = new Address('Storgatan 1', '123 45', 'Storstaden') // OK adress!
console.log(address1.toString())

const address2 = new Address('Lillgatan 2', 'ingen aning', 'Lillstaden') // Felaktigt postnummer!
console.log(address2.toString())
```

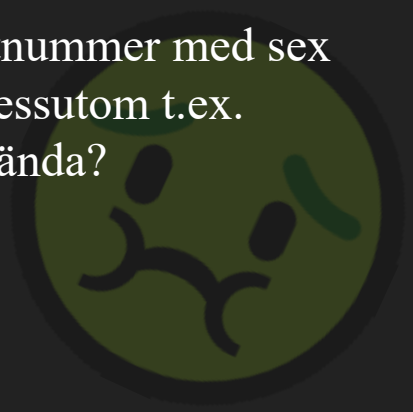
OUTPUT
```
Storgatan 1
123 45 Storstaden
```

OUTPUT
```
Lillgatan 2
ingen aning Lillstaden
```

- ✓ Att kontrollera att till exempel postnumret består av fem siffror med eventuellt mellanslag mellan tredje och fjärde siffran kräver en hel del kod.

En hel del kod…

- ✓ …har skrivits i settern för `postalCode` för att validera värdet innan det tilldelas till den semi-privata egenskapen.
  - Nu kontrolleras att postnumret har rätt format:
    - Det måste vara fem eller sex tecken. Är det sex tecken måste mellanslaget ha rätt position.
    - Det måste vara siffror, men första siffran får inte vara 0.
- ✓ Utmanande kod att skriva och underhålla!
  - Vad behöver vi ändra om även postnummer med sex siffror börjar användas, d.v.s. om dessutom t.ex. 123 456 och 123456 ska vara godkända?



```
// address.js
export class Address {
  constructor (addressLine, postalCode, city) {
    this.addressLine = addressLine
    this.postalCode = postalCode
    this.city = city
  }

  get postalCode () {
    return this._postalCode
  }

  set postalCode (value) {
    let valid = false

    if (value.length === 5 ||
        (value.length === 6 && value.indexOf(' ') === 3)) {
      let i = 0
      for (; i < value.length; i++) {
        if (value[i] === ' ') {
          continue
        }

        if (value[i] < '0' || value[i] > '9' ||
            (i === 0 && value[i] === '0')) {
          break
        }
      }
      valid = i >= 5 && i <= 6
    }

    if (!valid) {
      throw new Error('Invalid postal code.')
    }

    this._postalCode = value
  }

  toString () {
    return `${this.addressLine}\n${this._postalCode} ${this.city}`
  }
}
```

Det måste finnas ett enklare(?) och bättre(!) sätt att göra detta!

Reguljärt uttryck till undsättning

- ✓ Reguljära uttryck är ett kraftfullt sätt att bearbeta och hitta mönster i texter.
  - Reguljära uttryck finns i de flesta programmeringsspråk så som Python, Java, C#, etc.
- ✓ Vad är ett reguljärt uttryck?
  - Ett reguljärt uttryck är ett mönster som används för att matcha kombinationer av tecken i en sträng.
  - Ett sätt att skapa ett reguljärt uttryck som en literal är att skriva ett mönster mellan snedstreck (/). Det går även att skapa en instans av `RegExp`.
  - Exempel
    - Det reguljära uttrycket `/tre/` kommer att matcha ”tre” någonstans i en sträng.
      - Matchar ”**tre**”, ”**tre**vligt” och ”s**trejka**”.
    - Det reguljära uttrycket `/^tre/` kommer att matcha ”tre” i början av en sträng.
      - Matchar ”**tre**”, ”**tre**vligt”, men inte ”s**trejka**”.

```
// address.js
export class Address {
  constructor (addressLine, postalCode, city) {
    this.addressLine = addressLine
    this.postalCode = postalCode
    this.city = city
  }

  get postalCode () {
    return this._postalCode
  }

  set postalCode (value) {
    if (!/^[1-9]\d{2} ?\d{2}$/.test(value)) {
      throw new Error('Invalid postal code.')
    }

    this._postalCode = value
  }

  toString () {
    return `${this.addressLine}\n${this._postalCode} ${this.city}`
  }
}
```

Enklare(?) och bättre, när du väl kan lite om reguljära uttryck.

Hur ska `/^[1-9]\d{2} ?\d{2}$/` tolkas?

- ✓ Mellan snedstrecken är det reguljära uttrycket där:
  - `^` start av strängen
  - `[1-9]` exakt en förekomst av 1, 2, 3, 4, 5, 6, 7, 8, eller 9
  - `\d{2}` exakt två förekomster av 0 till 9
  - `␣?` ingen eller en förekomst av ett mellanslag

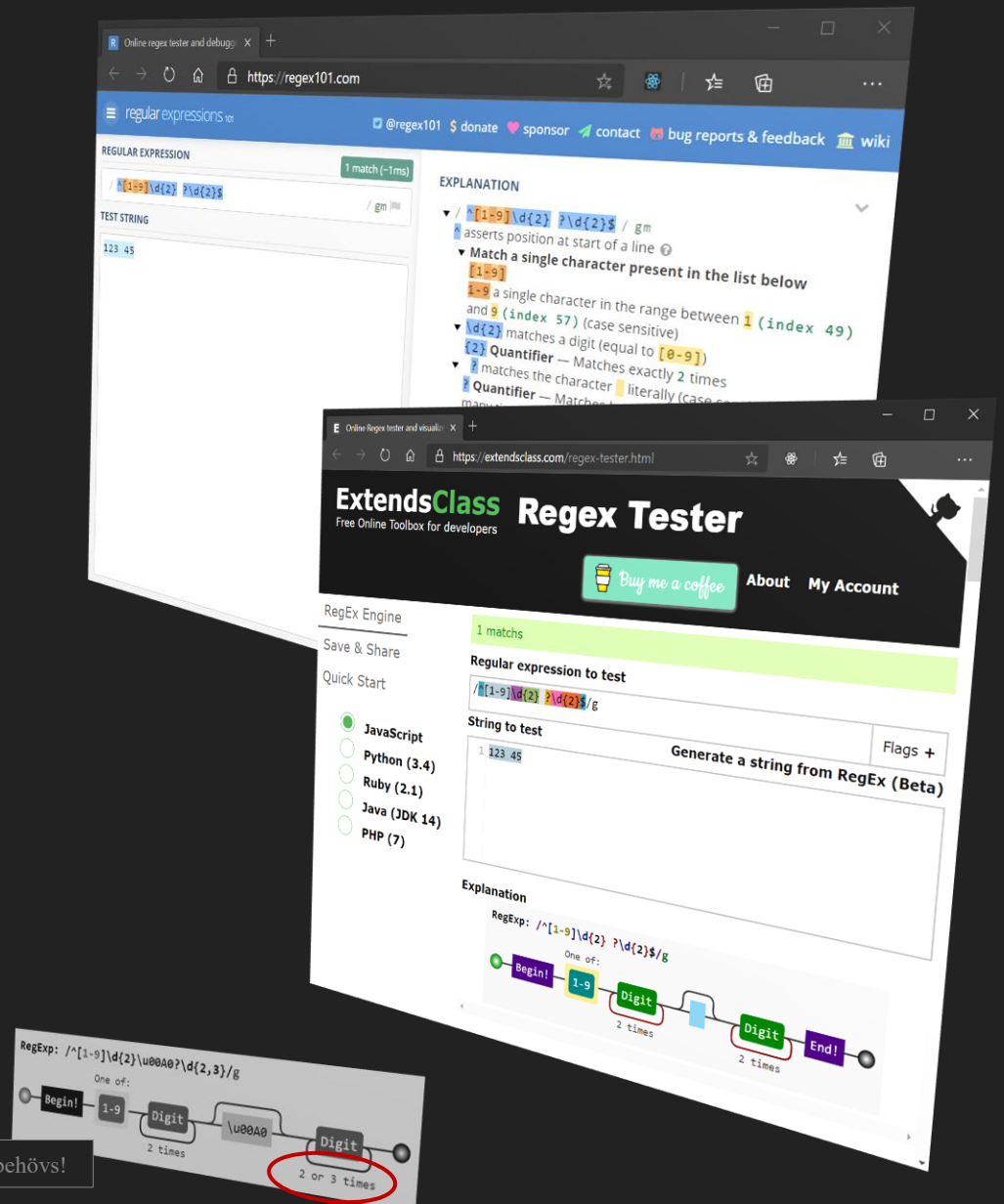
Ett mellanslag!

  - `\d{2}` exakt två förekomster av 0 till 9
  - `$` slut på strängen

- ✓ Du hittar mer om reguljära uttryck på MDN under rubriken Regular expressions.
  - Verktyg som nämns och är värda besök är regular expressions 101 och ExtendsClass Regex Tester.

- ✓ Vad behöver vi ändra om även postnummer med sex siffror börjar användas, d.v.s. om dessutom t.ex. 123 456 och 123456 ska vara godkända? Jo, `/^[1-9]\d{2} ?\d{2,3}$/`!

…och det är allt som behövs!



Vad är kommandoradargument?

- Ofta då du skriver en applikation vill du kunna göra applikationen mer generell genom att göra det möjligt att skicka information, så kallade kommandoradargument, till applikationen då den startar.

```
❯ npm start ett två 3 4
```

```
❯ node src/app.js ett två 3 4
```

kommandoradargument

- Node.js gör det enkelt att få reda på den lista med argument, argumentvektorn, som skickats in. Argumentvektorn är en array du når genom `process.argv`.
  - Arrayen innehåller allt, i form av strängar, som skickas till applikationen, inklusive sökvägen till node.exe samt sökvägen till applikationsfilen som startar applikationen.

```
console.log(process.argv)
```

```
❯ npm start ett två 3 4

> example@1.0.0 start C:\some\path\to\example
> node src/app.js "ett" "två" "3" "4"

 [
  'C:\\Program Files\\nodejs\\node.exe',
  'C:\\some\\path\\to\\example\\src\\app.js',
  'ett',
  'två',
  '3',
  '4'
 ]
```

Ofta är de två första elementen i arrayen ointressanta och kan hoppas över.

Hantera kommandoradargument

- För att plocka ut de argument som är av intresse kan man hoppa över de två första elementen i arrayen genom att lämna tomma platser vid destrukturering.

```
// app.js
const add = function (a, b) {
  return a + b
}

console.log('process.argv:\n', process.argv, '\n')

const [, , first, second] = process.argv.map(Number)

if (process.argv.length !== 4 ||
  !Number.isFinite(first) ||
  !Number.isFinite(second)) {
  throw new Error('Two numbers are required.')
}

const sum = add(first, second)
console.log(sum)
```

Destrukturering av returnerad array. De två inledande kommatecknen gör att de två första elementen i arrayen hoppas över.

`Number.isFinite` upptäcker `NaN` (vilket till exempel `'hej'` ger) och `undefined` (för få argument). Utan `try/catch` kraschar programmet fortfarande okontrollerat.

- Giltigt indata.

```
npm start 37 5

> example@1.0.0 start
> node src/app.js 37 5

process.argv:
 [
  'C:\\some\\path\\to\\node\\node.exe',
  'C:\\some\\path\\to\\example\\src\\app.js',
  '37',
  '5'
]

42
```

37 + 5 är 42. Allt fungerar som förväntat!

- Ogiltigt indata – ohanterat, kraschar med avslutningskoden 1.

```
npm start hej 5

> example@1.0.0 start
> node src/app.js hej 5

process.argv:
 [
  'C:\\some\\path\\to\\node\\node.exe',
  'C:\\some\\path\\to\\example\\src\\app.js',
  'hej',
  '5'
]

file:///C:/some/path/to/example/src/app.js:12
  throw new Error('Two numbers are required.')
        ^

Error: Two numbers are required.
    at file:///C:/some/path/to/example/src/app.js:12:9
    at ModuleJob.run (node:internal/modules/esm/module_job:447:25)
    at async node:internal/modules/esm/loader:646:26
    at async asyncRunEntryPointWithESMLoader (node:internal/modules/run_main:101:5)
```

Strängen `'hej'` kan inte tolkas som ett `Number` varför kastat undantag är förväntat.

Att avsluta en Node-applikation

- ✓ Då en applikation avslutas sätter Node automatiskt avslutningskoden till värdet `0`, om allt gått bra.
- ✓ Kastar applikationen ett undantag som inte hanteras avslutas applikationen och avslutningskoden sätts till `1`.
- ✓ Det är möjligt att bestämma avslutningskoden innan applikationen avslutas.
  - ~~`process.exit(42)`~~
    - Metoden tvingar applikationen att avsluta så snart som möjligt. Oftast är det onödigt att anropa denna metod direkt då det är bättre att låta Node.js hantera avslutandet av applikationen på egen hand.
  - `process.exitCode = 42`
    - Sätt `process.exitCode` till ett värde och låt Node.js avsluta applikationen snyggt och prydligt.

```
// my-fake-error.js
export class MyFakeError extends Error {
  constructor () {
    super('This is a fake error.')
    this.name = 'MyFakeError'
    this.errorCode = 42
  }
}

// app.js
import { MyFakeError } from './my-fake-error.js'

try {
  throw new MyFakeError()
} catch (err) {
  console.error(err.message)
  process.exitCode = Number.isInteger(err.errorCode) ? err.errorCode : 1
}
```

En egen klass för fel.

Om ett fel har inträffat och `errorCode` är ett heltal sätts avslutningskoden till det värdet; annars `1`.

```
C:\some\path\node.exe .\src\app.js
This is a fake error.
Process exited with code 42
```

Körs koden i debug-läge ger VSCode avslutningskoden.

Strukturerad felhantering av kommandoradargument

- ✓ Valideringen och avslutningskoden knyts ihop med en egen felklass.

```
// invalid-argument-error.js
export class InvalidArgumentError extends Error {
  constructor(message) {
    super(message)
    this.name = 'InvalidArgumentError'
    this.errorCode = 2
  }
}

// app.js
import { InvalidArgumentError } from './invalid-argument-error.js'

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

try {
  const [, , first, second] = process.argv.map(Number)
  if (process.argv.length !== 4 ||
    !Number.isFinite(first) ||
    !Number.isFinite(second)) {
    throw new InvalidArgumentError('Two numbers are required.')
  }
  console.log(add(first, second))
} catch (err) {
  console.error(err.message)
  process.exitCode = Number.isInteger(err.errorCode) ? err.errorCode : 1
}
```

- ✓ En egen felklass, `InvalidArgumentError`, bär med sig en avslutningskod som en egenskap (`errorCode`).
- ✓ Kommandoradsargumenten valideras – är de inte precis två numeriska värden kastas felet.
- ✓ I `catch`-blocket sätts avslutningskoden till felets `errorCode`, annars till `1` som standardvärde.
- ✓ OBS! Fångar du ett undantag och inte tilldelar `process.exitCode` ett värde kommer dess värde att bli `0`.