Resurse suplimentare
pe care recomand să le parcurgi

Tipuri de erori în JavaScript

JavaScript are câteva tipuri predefinite de erori, fiecare reprezentând diferite categorii de probleme:

1. Error

Clasa de bază pentru toate erorile standard în JavaScript. Poate fi folosită și pentru a crea erori personalizate.

2. SyntaxError

Apare când codul conține o eroare de sintaxă care nu poate fi procesată de interpretor.

```
// Eroare de sintaxă - lipsește ) la finalul funcției
function greet( {
    return "Salut!";
}
```

3. ReferenceError

Apare când încerci să accesezi o variabilă care nu există sau nu a fost încă declarată.

```
// variabila message nu este definită
console.log(message)
```

4. TypeError

Apare când o operație este efectuată pe un tip de date neașteptat.

```
// Încercarea de a apela ca funcție ceva ce nu este o funcție
const number = 42
number() // TypeError: number is not a function
```

5. RangeError

Apare când o valoare numerică este în afara intervalului permis.

```
// Prea multe recursii
function infiniteRecursion() {
  infiniteRecursion()
}
infiniteRecursion() // RangeError: Maximum call stack size exceeded
```

6. URIError

Apare când funcțiile de lucru cu URI (encodeURI, decodeURI, etc.) primesc parametri invalizi.

```
decodeURIComponent('%') // URIError: URI malformed
```

7. EvalError

Istoric, a apărut pentru erori legate de funcția `eval()`. În prezent, este folosit rar.

Gestionarea erorilor cu try...catch

Blocul `try...catch` ne permite să "prindem" erorile și să le gestionăm în mod elegant, fără a opri execuția întregului script.

Structura de bază try...catch

```
try {
  // Codul care ar putea genera o eroare
  const result = 10 / 0
  console.log(result) // Infinity (această operație nu generează eroare în JavaScript)

  // Generăm o eroare artificială
  console.log(nonExistentVariable) // Va genera ReferenceError
} catch (error) {
  // Codul care se execută când apare o eroare
  console.log('A apărut o eroare:', error.message)
} finally {
  // Acest bloc se execută întotdeauna, indiferent dacă a apărut o eroare sau nu
  console.log('Această instrucțiune se execută întotdeauna.')
}
```

Blocul finally

Blocul `finally` este opțional și se va executa întotdeauna, indiferent dacă a apărut o eroare sau nu. Este util pentru curățare (de exemplu, închiderea fișierelor sau conexiunilor).

```
function testeazaEroare() {
  try {
    console.log('Începem operațiunea')
    // Generăm o eroare
    throw new Error('Ups, ceva nu a mers bine!')
    // Acest cod nu se va executa
    console.log('Acest mesaj nu va fi afișat')
  } catch (err) {
    console.log('A apărut o eroare:', err.message)
    return 'din catch' // Încercăm să returnăm din funcție
  } finally {
    console.log('Operațiunea s-a încheiat')
    // Blocul finally se execută chiar și când avem return în try sau catch
  }
}

console.log(testeazaEroare())
// Afișează:
// Începem operațiunea
// A apărut o eroare: Ups, ceva nu a mers bine!
// Operațiunea s-a încheiat
// din catch
```

Aruncarea erorilor cu throw

Keyword-ul `throw` ne permite să generăm manual erori atunci când detectăm condiții invalide.

Structura de bază

```
function divideNumbers(a, b) {
  if (b === 0) {
    throw new Error('Nu se poate împărți la zero!')
  }
  return a / b
}

try {
  console.log(divideNumbers(10, 2)) // 5
  console.log(divideNumbers(10, 0)) // Va genera o eroare
} catch (err) {
  console.log('Eroare:', err.message) // Eroare: Nu se poate împărți la zero!
}
```

Aruncarea diferitelor tipuri de erori

Poți folosi constructor-ul `Error` sau oricare dintre subclasele sale:

```
// Eroare generică
throw new Error('Mesaj de eroare general')

// Eroare de tip
throw new TypeError('Tipul de date nu este potrivit')

// Eroare de valoare în afara intervalului
throw new RangeError('Valoarea este în afara intervalului permis')
```

Crearea erorilor personalizate

Putem extinde clasa `Error` pentru a crea tipuri personalizate de erori adaptate nevoilor specifice ale aplicației noastre.

```
// Definirea unei clase de eroare personalizată
class ValidationError extends Error {
  constructor(message, invalidField) {
    super(message)
    this.name = 'ValidationError'
    this.invalidField = invalidField
  }
}

function validateAge(age) {
  if (isNaN(age)) {
    throw new ValidationError('Vârsta trebuie să fie un număr', 'age')
  }

  if (age < 0 || age > 120) {
    throw new ValidationError('Vârsta trebuie să fie între 0 și 120', 'age')
  }

  return true
}

try {
  validateAge('douazeci')
} catch (err) {
  if (err instanceof ValidationError) {
    console.log(
      `Eroare de validare pentru câmpul ${err.invalidField}: ${err.message}`
    )
  } else {
    console.log('A apărut o eroare:', err.message)
  }
}
```

Gestionarea erorilor asincrone

În codul asincron, gestionarea erorilor are câteva particularități:

Cu callback-uri

```
function operatieAsincrona(callback) {
  setTimeout(() => {
    try {
      // Simulăm o eroare
      throw new Error('Eroare în operația asincronă')
    } catch (err) {
      callback(err, null)
    }
  }, 1000)
}

operatieAsincrona((eroare, rezultat) => {
  if (eroare) {
    console.log('A apărut o eroare:', eroare.message)
  } else {
    console.log('Rezultatul:', rezultat)
  }
})
```

Cu Promises

```
function operatieAsincrona() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      // Simulăm o eroare
      reject(new Error('Eroare în operația asincronă'))
    }, 1000)
  })
}

operatieAsincrona()
  .then((rezultat) => {
    console.log('Rezultatul:', rezultat)
  })
  .catch((eroare) => {
    console.log('A apărut o eroare:', eroare.message)
  })
```

Cu async/await

```
async function executaOperatie() {
  try {
    const rezultat = await operatieAsincrona()
    console.log('Rezultatul:', rezultat)
  } catch (eroare) {
    console.log('A apărut o eroare:', eroare.message)
  }
}

executaOperatie()
```

Strategii de gestionare a erorilor

1. Validarea input-urilor

Verifică datele de intrare înainte de a le procesa pentru a preveni erorile.

```
function addUser(user) {
  if (!user) {
    throw new Error('Obiectul utilizator este obligatoriu')
  }

  if (!user.name || typeof user.name !== 'string') {
    throw new TypeError('Numele utilizatorului trebuie să fie un string valid')
  }

  if (!user.email || !user.email.includes('@')) {
    throw new Error('Email-ul utilizatorului este invalid')
  }

  // Procesează utilizatorul...
  console.log('Utilizator adăugat cu succes:', user.name)
}

try {
  addUser({ name: 'Ana', email: 'ana@exemplu.com' }) // OK
  addUser({ name: 'Ion' }) // Eroare: email invalid
} catch (err) {
  console.log('Nu s-a putut adăuga utilizatorul:', err.message)
}
```

2. Folosirea valorilor implicite

```
function calculateTotal(products = []) {
  // Folosim array gol ca valoare implicită pentru a evita erorile
  return products.reduce((total, product) => total + (product.price || 0), 0)
}

const total1 = calculateTotal([{ price: 10 }, { price: 20 }])
console.log(total1) // 30

const total2 = calculateTotal() // Fără argumente
console.log(total2) // 0 (nu generează eroare)
```

3. Verificarea tipurilor

```
function add(a, b) {
  if (typeof a !== 'number' || typeof b !== 'number') {
    throw new TypeError('Ambii parametri trebuie să fie numere')
  }
  return a + b
}

try {
  console.log(add(5, '10')) // Eroare
} catch (err) {
  console.log(err.message) // Ambii parametri trebuie să fie numere
}
```

4. Înregistrarea erorilor (Logging)

În aplicații reale, este important să înregistrezi erorile pentru analiză și depanare.

```
function logError(error, importanceLevel = 'error') {
  const timeStamp = new Date().toISOString()
  const errorMessage = `[${timeStamp}] [${importanceLevel}] ${error.name}: ${error.messag
e}`

  // În aplicații reale, ai putea:
  // 1. Trimite eroarea către un server de logging
  // 2. Salva în localStorage pentru debugging
  // 3. Afișa în consolă pentru dezvoltare
  console.log(errorMessage)

  // Aici ai putea adăuga logica de trimitere către server
}

try {
  throw new Error('Aceasta este o eroare de test')
} catch (err) {
  logError(err)
}
```

Debugging în JavaScript

Depanarea erorilor este o abilitate esențială. Iată câteva tehnici:

1. Folosirea console.log() strategic

```
function complexFunction(data) {
  console.log('Date de intrare:', data)

  const processingResult = processComplexData(data)
  console.log('După procesare:', processingResult)

  return processingResult
}
```

2. Utilizarea consolei browser-ului

Consola browser-ului oferă diverse metode utile:

```
console.log('Mesaj informativ')
console.error('Mesaj de eroare')
console.warn('Avertisment')
console.info('Informație')

// Gruparea mesajelor
console.group('Grup de mesaje')
console.log('Mesaj în grup 1')
console.log('Mesaj în grup 2')
console.groupEnd()

// Măsurarea timpului
console.time('timer1')
// Cod de cronometrat...
console.timeEnd('timer1') // Afișează timpul scurs

// Tabele
console.table([
  { nume: 'Ana', varsta: 28 },
  { nume: 'Ion', varsta: 35 },
])
```

3. Folosirea debugger-ului

Cuvântul cheie `debugger` oprește execuția codului și deschide debugger-ul, dacă este disponibil.

```
function calculateSum(numbers) {
  let sum = 0
  for (let i = 0; i < numbers.length; i++) {
    debugger // Execuția se va opri aici dacă DevTools este deschis
    sum += numbers[i]
  }
  return sum
}
```

4. Try-catch pentru debugging

```
function debugFunction(fn, ...args) {
  try {
    return fn(...args)
  } catch (err) {
    console.error('DEBUG: Eroare în funcția', fn.name)
    console.error('DEBUG: Argumentele', args)
    console.error('DEBUG: Eroare', err)
    throw err // Re-aruncăm eroarea pentru a nu o înghiți
  }
}

// Utilizare
function divide(a, b) {
  return a / b
}

debugFunction(divide, 10, 0) // Va afișa informații detaliate despre eroarea apărută
```

Cele mai bune practici pentru gestionarea erorilor

1. **Gestionează erorile specific - Prinde erorile cât mai aproape de sursa lor.**

```
function processData() {
  try {
    // Cod specific
  } catch (err) {
    // Gestionare specifică acestei funcții
  }
}
```

Nu înghiți excepțiile fără motiv - Întotdeauna fă ceva cu erorile prinse.

```
// Bine
try {
  riskyOperation()
} catch (err) {
  console.error('Eroare în riskyOperation:', err)
  // Eventual re-aruncă pentru a fi prinsă mai sus în stivă
  throw err
}
```

Folosește blocul finally pentru curățare - Asigură-te că resursele sunt eliberate.

```
let connection = null
try {
  connection = openConnection()
  // Operații cu conexiunea
} catch (err) {
  console.error('Eroare la utilizarea conexiunii:', err)
} finally {
  // Închide conexiunea indiferent dacă a apărut o eroare sau nu
  if (connection) {
    connection.close()
  }
}
```

Creează erori descriptive - Include suficiente informații pentru a identifica problema.

```
if (!user.id) {
  throw new Error(
    'Utilizatorul nu are ID valid. Detalii: ' + JSON.stringify(user)
  )
}
```

Folosește instanceof pentru verificarea tipului de eroare - Permite gestionarea diferitelor tipuri de erori.

```
try {
  // Cod care poate arunca diferite tipuri de erori
} catch (err) {
  if (err instanceof TypeError) {
    // Gestionează erori de tip
  } else if (err instanceof RangeError) {
    // Gestionează erori de interval
  } else {
    // Gestionează alte tipuri de erori
  }
}
```