# JAVASCRIPT

JavaScript kalboje yra trys skirtingi kintamųjų (arba kitų kintamųjų tipų) deklaravimo raktiniai žodžiai: **var**, **let**, ir **const**. Pasirinkimas tarp var, let ir const priklauso nuo to, kokių savybių jums reikia. const naudojama, kai norima, kad kintamasis būtų nekintamas, o let naudojama, kai planuojate perrašyti kintamojo reikšmę. var yra senesnė deklaravimo forma, ir dažnai geriau naudoti let ir const dėl jų saugesnio ir prognozuojamo elgesio. Kiekvienas iš jų turi savo savybes ir naudojimo taisykles:

|  |  |  |
| --- | --- | --- |
| **var:** | **let:** | **const:** |
| Scope (sritis): Kintamieji, deklaruoti su var, yra **funkcijos apimtyje** arba globaliame lygmenyje. **Negalima** naudoti blokinės (block) srities, pvz., **if, for, while**.  Hoisting: Kintamieji, deklaruoti su var, yra "kėlinami" (hoisted) į funkcijos pradžią arba globalų kontekstą, todėl gali būti naudojami prieš deklaravimą.  function example() {      if (true) {        var x = 10;      }      console.log(x); // Čia x bus matomas ir pasiekiamas, nes var nesilaiko blokinės srities    } | Scope: Kintamieji, deklaruoti su **let**, laikomi blokinėje sričių riboje (**block scope**), tai yra jie yra matomi tik bloke, kuris juos aprėpia (pvz., **if, for, while**).  Hoisting: Kintamieji, deklaruoti su let, taip pat yra "kėlinami", bet prieš pasiekimą jie yra "temporarily dead zone", kur juos naudoti yra klaida.  function example() {      if (true) {        let y = 20;      }      console.log(y); // Klaida! y neegzistuoja šioje sričių riboje    } | Scope: Kaip ir **let**, **const** laikomas **blokinėje** sričių riboje.  Pertvarkymas (Reassignment): Kintamąjį, deklaruotą su const, galima **priskirti tik vieną kartą**. Tai reiškia, kad kintamųjų reikšmės negalima keisti po priskyrimo.  const z = 30;  z = 40; // Klaida! Konstantos negalima perrašyti |

## Baziniai operatoriai:

Galime trumpinti formules.

A black board with white lines and numbers

Description automatically generated

% ženklas nėra procentas ar dalyba jis skaičiuoja liekana. Pvz. 5%2, ats.:1

### Javascript operatoriai

Operandas yra reikšmė arba reiškinys, prie kurio taikomas operatorius. Operatoriai veikia su vienu arba daugiau operandų, atliekant tam tikrą operaciją. Operandai gali būti skaičiai, kintamieji, konstantos, arba kitokių tipų reikšmės, su kuriais atliekama tam tikra veiksmų seka.

**Pavyzdžiui, žiūrint į paprastą matematikos išraišką "2 + 3", čia "2" ir "3" yra operandai, o "+" yra operatorius, kuris atlieka sudėties veiksmą su šiais operandais.**

|  |  |  |  |
| --- | --- | --- | --- |
| Unary plus | **+** | Konvertuoja operandą į skaičių ir grąžina teigiamą skaičių. | let x = 5;  console.log(+x); // 5  console.log(-x); // -5  console.log(++x); // 6  console.log(--x); // 5  console.log(!true); // false  console.log(~5); // -6  console.log(typeof x); // "number"  delete obj.property; // Ištrina savybę iš objekto  void function() {}; // Grąžina undefined |
| Unary negation | **-** | Keičia skaičiaus ženklą į priešingą (teigiamą į neigiamą ir atvirkščiai) |
| Increment | **++** | Prieš operandą prideda vienetą |
| Decrement | **- -** | Nuo operandų atima vienetą. |
| Logical NOT | **!** | Pakeičia operandą į jo priešingą loginę reikšmę (jei buvo **true**, tampa **false** ir atvirkščiai) |
| Bitwise NOT | **~** | Pakeičia operandą į jo priešingą bitų reikšmę. |
| Typeof |  | Grąžina tekstinę eilutę, nurodančią operandų duomenų tipą |
| Delete |  | Naudojamas ištrinti savybę iš objekto arba elementą iš masyvo. |
| Void |  | Grąžina **undefined** reikšmę. |

### Binariniai operatoriai

Binariniai operatoriai yra operatoriai, atliekantys binarines operacijas, kuriose dalyvauja du operandai. Programavimo ir matematikos kontekste šie operatoriai priima du reikšmes (operandus) ir atlieka tam tikrą operaciją, kurios rezultatas yra nauja reikšmė.

**Štai keli dažniausiai naudojami binariniai operatoriai:**

1. **Aritmetiniai operatoriai:**

|  |  |
| --- | --- |
| **Sudėtis (+)**  **Atimtis (-)**  **Daugyba (\*)**  **Dalyba (/)**  **Liekana (%)** | let suma = 5 + 3; // Sudėtis  let skirtumas = 7 - 2; // Atimtis  let sandauga = 4 \* 6; // Daugyba  let dalmuo = 8 / 2; // Dalyba  let liekana = 9 % 4; // Liekana |

*% ženklas nėra procentas ar dalyba jis skaičiuoja liekana. Pvz. 5%2, ats.:1*

1. **Palyginimo operatoriai:**

|  |  |
| --- | --- |
| **Lygu (==)**  **Nelygu (!=)**  **Griežtai lygu (===)**  **Griežtai nelygu (!==)**  **Daugiau (>)**  **Mažiau (<)**  **Daugiau arba lygu (>=)**  **Mažiau arba lygu (<=)** | let arLygu = (5 == 5); // Lygu  let arNelygu = (6 != 3); // Nelygu  let arGrieztaiLygu = (3 === '3'); // Griežtai lygu  let arGrieztaiNelygu = (4 !== '4'); // Griežtai nelygu  let arDaugiau = (10 > 8); // Daugiau  let arMaziau = (7 < 9); // Mažiau  let arDaugiauArbaLygu = (5 >= 5); // Daugiau arba lygu  let arMaziauArbaLygu = (6 <= 6); // Mažiau arba lygu |

1. **Loginiai operatoriai:**

|  |  |
| --- | --- |
| **Loginis IR (&&)**  **Loginis ARBA (||)** | let loginisIr = (true && false); // Loginis IR  let loginisArba = (true || false); // Loginis ARBA |

**Matematiniai objetai (formulės)**

|  |  |
| --- | --- |
| 1. **Math.abs(x) - Modulis:**   Grąžina skaičiaus absoliutinę vertę (modulį). | let absValue = Math.abs(-5); // absValue bus 5  Math.sqrt(x) - Šaknis: |
| 1. **Math.sqrt(x) - Šaknis:**   Grąžina skaičiaus kvadratinę šaknį. | let squareRoot = Math.sqrt(16); // squareRoot bus 4 |
| 1. **Math.pow(x, y) - Kėlimas laipsniu:**   Grąžina skaičių, pakeltas nurodytu laipsniu. | Math.pow(x, y) - Kėlimas laipsniu:  let powerResult = Math.pow(2, 3); // powerResult bus 8 |
| 1. **Math.floor(x) - Apvalinimas žemyn:**   Grąžina mažiausią sveikąjį skaičių, kuris yra mažesnis arba lygus duotam skaičiui. | Math.floor(x) - Apvalinimas žemyn:  let floorValue = Math.floor(3.8); // floorValue bus 3 |
| 1. **Math.ceil(x) - Apvalinimas aukštyn:**   Grąžina mažiausią sveikąjį skaičių, kuris yra didesnis arba lygus duotam skaičiui. | Math.ceil(x) - Apvalinimas aukštyn:  let ceilValue = Math.ceil(3.2); // ceilValue bus 4 |
| 1. **Math.round(x) - Apvalinimas:**   Apvalina skaičių į artimiausią sveikąjį skaičių. | Math.round(x) - Apvalinimas:  let roundValue = Math.round(4.5); // roundValue bus 5 |
| 1. **Math.random() - Atsitiktinis skaičius:**   Grąžina atsitiktinį slankiojo kablelio skaičių intervale [0, 1). | Math.random() - Atsitiktinis skaičius:  let randomValue = Math.random(); // randomValue bus atsitiktinis skaičius tarp 0 ir 1 (neįskaitant 1) |
| 1. **Math.PI - Skaičius π (pi):**   Grąžina matematinę konstantą π. | let piValue = Math.PI; // piValue bus 3.141592653589793 |

[Math - JavaScript | MDN (mozilla.org)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math)

## Duomenų tipai

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Number:** | **String:** | **Boolean:** | **null:** | **undefined:** |
| Šis tipas naudojamas aprašyti skaičius. Jis apima sveikuosius skaičius ir slankiojo kablelio skaičius.  **NaN** ir **Infitity** taip pat number.  let integerNumber = 42;  let floatNumber = 3.14; | Tai yra teksto tipas, naudojamas aprašyti simbolių eilutes. Simbolių eilutės yra apibrėžiamos viengubose (**''**) arba dvigubose (**""**) kabutėse.  let text = "Labas, pasauli!"; | Šis tipas turi tik dvi galimos reikšmės - **true** arba **false**. Jis dažnai naudojamas loginėse operacijose ir sąlygose.  let isTrue = true;  let isFalse = false; | Šis tipas turi tik vieną reikšmę - **null.** Ji nurodo, kad kintamasis neturi reikšmės.  let emptyValue = null; | Kintamojo reikšmė, kai ji dar nepriskirta jokia reikšmė.  let notDefined;  let age;  //rodys "undefined" nes age buvo deklaruota bet nepriskirta reikšmė  alert(age) |

### Typeof operatorius

JavaScript kalboje **typeof** operatorius grąžina argumento tipą. Jis palaiko dvi sintaksės formas: kaip operatorių (**typeof x**) ir kaip funkciją (**typeof(x)).**

Šis operatorius yra naudingas, kai norima gauti informaciją apie kintamojo tipą programos metu. Priklausomai nuo argumento, **typeof** gali grąžinti reikšmes, tokią kaip "number", "string", "boolean", "object", "function", "undefined", arba "symbol".

|  |  |
| --- | --- |
| 1. **Sintaksė kaip operatorius:**   **typeof x:** Grąžina argumento tipo pavadinimą kaip eilutės reikšmę. | 1. **Sintaksė kaip funkcija:**   **typeof(x):** Taip pat grąžina argumento tipo pavadinimą kaip eilutės reikšmę. |
| let example = 42;  console.log(typeof example); // Rezultatas: "number" | let example = "Hello, World!";  console.log(typeof(example)); // Rezultatas: "string" |

## Ciklas (While)

### Increment/Decrement

"Increment" ir "decrement" yra operacijos, kurios atliekamos su skaičiais ir skirtos padidinti arba sumažinti jų reikšmę vienetu. Šios operacijos dažnai naudojamos programavime, ypač cikluose ar kituose kontekstuose, kai reikia keisti kintamojo reikšmę vienetu.

|  |  |
| --- | --- |
| **Increment (Padidinimas):** | **Decrement (Sumažinimas):** |
| "Increment" operacija padidina skaičių vienetu. Šią operaciją galima atlikti naudojant **++** operatorių.  let x = 5;  x++; // Padidina x reikšmę vienetu  console.log(x); // Rezultatas: 6  alternatyva:  let y = 8;  y = y + 1; // Padidina y reikšmę vienetu  console.log(y); // Rezultatas: 9 | "Decrement" operacija sumažina skaičių vienetu. Šią operaciją galima atlikti naudojant **--** operatorių.  let a = 10;  a--; // Sumažina a reikšmę vienetu  console.log(a); // Rezultatas: 9  alternatyva  let b = 7;  b = b - 1; // Sumažina b reikšmę vienetu  console.log(b); // Rezultatas: 6 |

Galima naudoti tiek **++a** (prieš padidinant) tiek **--b** (prieš sumažinant) užrašus. Šie užrašai taip pat vykdo "increment" ir "decrement" operacijas, tačiau su šiais skirtumais:

|  |  |
| --- | --- |
| **++a (prieš padidinant):** | **--b (prieš sumažinant):** |
| Pirmiausiai padidina reikšmę vienetu ir grąžina jau padidintą reikšmę.  Tai vadinama "pre-increment" operacija.  let x = 5;  let result = ++x;  console.log(x); // x dabar yra 6  console.log(result); // Rezultatas: 6 | Pirmiausiai sumažina reikšmę vienetu ir grąžina jau sumažintą reikšmę.  Tai vadinama "pre-decrement" operacija.  let y = 8;  let result = --y;  console.log(y); // y dabar yra 7  console.log(result); // Rezultatas: 7 |

A close-up of a person

Description automatically generated

A pair of red and white lines

Description automatically generated

### Nežinomo kartojimų skaičiaus ciklas WHILE

Jis vykdo tam tikrą bloką kode, kol sąlyga išlieka tiesa (true). Čia yra paprastas while ciklo pavyzdys:

|  |  |
| --- | --- |
| let count = 0;  while (count < 5) {    console.log("Count: " + count);    count++; // Padidina count reikšmę vienetu} | REZULTATAS  Count: 0  Count: 1  Count: 2  Count: 3  Count: 4 |

Šis pavyzdys sukurs ciklą, kuris vykdys tam tikrą bloką kode, tol kol count bus mažiau nei 5. Kiekvieno ciklo iteracijos metu išvestas bus "Count: " kartu su esama count reikšme. Po kiekvieno ciklo count padidinamas vienetu (count++).

Svarbu užtikrinti, kad ciklo sąlyga taptų netiesa (false) tam tikru metu, kitaip ciklas gali tęstis begalinį kartų. Taip pat būtina atkreipti dėmesį į tai, kad **while** cikle blokas vykdomas tik tada, kai sąlyga yra tiesa, todėl jei sąlyga iš pradžių yra netiesa, blokas gali būti praleistas.

### Break

„Begalinis ciklas + Break pagal poreikį“ yra puikus sprendimas situacijoms, kai ciklo sąlygos turi būti patikrinamos ne ciklo pradžioje ar pabaigoje, bet viduryje ar net keliuose jo kūno vietose.“

Kai programa pasiekia "break" komandą, ji išeina iš einamojo ciklo arba iš kito kontrolinio sakinio bloko, neprasukinėdamalikusios kodo dalies.

let suma = 0;

for (let i = 1; i <= 10; i++) {

    suma += i;

    if (suma > 15) {

        break; // Nutraukia ciklą, jei suma tampa didesnė nei 15

    }

}

console.log(suma); // Output: 21

### Continue

Kai programa pasiekia "continue" komandą, ji nevykdo likusios kodo dalies ciklo, bet tęsia kitą ciklo iteraciją.

Tai naudinga, kai norima praleisti likusią kodo dalį dabartinės ciklo iteracijos sąlygos tenkinimo atveju, bet norima tęsti su kitu iteracijos žingsniu.

Tad esminė skirtumas yra tas, kad "break" nutraukia ciklą arba kodo bloką, o "continue" **praleidžia** likusią ciklo dalį ir tęsia su kitu iteracijos žingsniu.

for (let i = 0; i < 5; i++) {

    if (i === 2) {

        continue; // Praleidžia šią iteraciją, kai i tampa lygu 2

    }

    console.log(i);

}

// Output: 0, 1, 3, 4

## JS ciklas FOR

|  |  |
| --- | --- |
| "**For**" ciklas yra vienas iš populiarių programavimo ciklų, leidžiantis įgyvendinti pasikartojantį kodą tam tikrą skaičių kartų.  Pagrindinė "**for**" ciklo sintaksė yra tokia:  for (pradinė\_sąlyga; sąlyga\_tęsimo; žingsnis) {      // Čia įgyvendinamas pasikartojantis kodas  }   * **Pradinė sąlyga**: Iš pradžių nustatoma pradinė būsena (dažniausiai inicializuojama kintamasis). * **Sąlyga tęsimo**: Ciklas tęsiamas, kol ši sąlyga yra tiesa (true). * **Žingsnis**: Kiekvieno ciklo žingsnio metu atliekama ši operacija. |  |

Pavyzdys naudojant "**for**" ciklą:

|  |  |
| --- | --- |
| for (let i = 0; i < 5; i++) {      console.log(i);  }  // Output: 0, 1, 2, 3, 4 | Šis ciklas pradeda **nuo nulio**, vykdomas tol, kol i yra mažesnis nei 5, ir po kiekvieno žingsnio padidina i vienetu. Ciklo kūne esantis kodas bus įvykdomas penkis kartus. |

### Algoritmai:sumos, kiekio, vidurkio

|  |  |
| --- | --- |
| 1. **Sumos algoritmas**   let sum = 0  for (let i =0; i<6; i++){      sum = sum +i;  }  console.log(sum);  //15 | 1. Sum = 0+1=1 2. Sum = 1+2=3 3. Sum = 3+3=6 4. Sum = 6+4=10 5. Sum = 10+5=15 |

|  |  |
| --- | --- |
| 1. **Kiekio skaičiavimo algoritmas**   let count = 0  for (let i =1; i<10; i++){      if (i % 3 == 0) {          count = count +1;      }  }  console.log(count);  //3 | 1. 1%3 ==0 ? False 2. 2%3 ==0 ? False 3. 3%3 ==0 ? True, count 0+1 =1 4. 4%3 ==0 ? False 5. 5%3 ==0 ? False 6. 6%3 ==0 ? True, count 1+1 =2 7. 7%3 ==0 ? False 8. 8%3 ==0 ? False 9. 9%3 ==0 ? True, count 2+1 =3 |

### Ciklas cikle

"**Nested loop**" (lietuviškai - "išorinis ciklas") yra ciklas, kuris yra įdėtas (arba "pridėtas") į kitą ciklą. Tai leidžia vykdyti vieną ciklą kiekvieno kito ciklo vykdymo metu. Šis principas naudojamas, kai turime dvi ar daugiau matmenų struktūrą, pavyzdžiui, matricą.

|  |  |
| --- | --- |
| Šis kodas sukurs išorinį ciklą, kuris vyks tris kartus, ir kiekvieną kartą vykdys vidinį ciklą, kuris taip pat vyks tris kartus. Taip gausime devynias kombinacijas (**i** nuo 1 iki 3 ir **j** nuo 1 iki 3).  // "Nested loop" su "for" ciklais  for (let i = 1; i <= 3; i++) {      for (let j = 1; j <= 3; j++) {          console.log(`Kai i: ${i}, tai j: ${j}`);      }  } | **Rezultatas:**  i: 1, j: 1  i: 1, j: 2  i: 1, j: 3  i: 2, j: 1  i: 2, j: 2  i: 2, j: 3  i: 3, j: 1  i: 3, j: 2  i: 3, j: 3 |

## JS Funkcijos

"Funkcija" programavime yra pernaudojamas kodo blokas, kuris atlieka konkrečias operacijas. Funkcijos leidžia organizuoti kodą, jį pernaudoti ir padaryti kodą lengviau suprantamą bei prižiūrimą.

**Pagrindinės funkcijos ypatybės:**

* **Pavadinimas:** Funkcijos pavadinimas identifikuoja ją ir leidžia ją iškviesti.
* **Parametrai:** Funkcija gali priimti įvairius parametrus (kintamuosius), kurie bus naudojami funkcijos viduje.
* **Vykdymo blokas:** Tai kodas, kuris vykdomas, kai funkcija iškviesta. Šiame bloke yra operacijos, kurias funkcija atlieka.
* **Grąžinimo reikšmė:** Funkcija gali grąžinti rezultatą po vykdymo.
* // Funkcija be parametrų ir grąžinimo reikšmės
* function sveikinimas() {
* console.log("Labas, pasauli!");
* }
* // Iškvietimas
* sveikinimas(); // Output: Labas, pasauli!
* // Funkcija su parametrais
* function pasisveikinimas(vardas) {
* console.log("Labas, " + vardas + "!");
* }
* // Iškvietimas
* pasisveikinimas("Jonai"); // Output: Labas, Jonai!
* // Funkcija su grąžinimo reikšme
* function sudaugink(a, b) {
* return a \* b;
* }
* // Iškvietimas
* let rezultatas = sudaugink(5, 3);
* console.log(rezultatas); // Output: 15

|  |  |
| --- | --- |
| **Parametrai:** | **Argumentai:** |
| * Parametrai yra kintamieji, kurie yra išvardyti funkcijos deklaracijoje. * Jie veikia kaip laikinosios vietos funkcijoje, atstovaujantys reikšmėms, kurias funkcija gaus, kai ji bus iškviesta. * Parametrai yra esminiai įvesties kintamieji, kuriuos funkcija tikisi gauti.   function sveikintis(vardas) {      console.log("Labas, " + vardas + "!");  }  Šiame pavyzdyje **vardas** yra funkcijos **sveikintis** parametras. | * Argumentai yra faktinės reikšmės, kurios perduodamos funkcijai, kai ji yra iškviečiama. * Jie atitinka funkcijoje deklaruotus parametrus. * Iškvietus funkciją, pateikiate argumentus, kurie atitinka funkcijos deklaracijoje nurodytą tvarką ir skaičių.   sveikintis("Jonai");  Šiame pavyzdyje "Jonai" yra argumentas, perduotas funkcijai **sveikintis**. Funkcijos **vardas** parametras šio kvietimo metu gaus reikšmę "Jonai". |

A computer screen with text and symbols

Description automatically generated

Funkcijos gali būti užrašomos įvairiais būdais, priklausomai nuo programavimo kalbos. Štai keletas būdų, kaip galima užrašyti funkcijas:

1. **Deklaratyvus (Declaration) būdas:**

Tai yra tradicinis būdas užrašyti funkcijas, naudojant raktažodį **function**:

function pavadinimas(parametras1, parametras2) {

    // funkcijos kūnas

    return rezultatas; // optional

}

//Pavyzdys:

function suma(a, b) {

    return a + b;

}

let rezultatas = suma(2, 3);

console.log(rezultatas); // Output: 5

1. **Išraiškos (Expression) būdas:**

Tai yra būdas užrašyti funkcijas kaip kintamuosius, naudojant function raktažodį arba anonominius (bevardžius) kintamuosius:

// Naudojant vardą

const pavadinimas = function(parametras1, parametras2) {

    // funkcijos kūnas

    return rezultatas; // optional

};

// Anoniminis (bevardis)

const funkcija = function(parametras1, parametras2) {

    // funkcijos kūnas

    return rezultatas; // optional

};

//pavyzdys:

const suma = function(a, b) {

    return a + b;

};

let rezultatas = suma(2, 3);

console.log(rezultatas); // Output: 5

1. **Sutrumpintas (Arrow Function) būdas (ES6 ir vėliau):**

Tai yra naujesnis būdas naudojant **=>** (streliuką):

const pavadinimas = (parametras1, parametras2) => {

    // funkcijos kūnas

    return rezultatas; // optional

};

Pavyzdys:

const suma = (a, b) => a + b;

let rezultatas = suma(2, 3);

console.log(rezultatas); // Output: 5

|  |  |
| --- | --- |
| Parametrų nurodymas yra procesas, kai apibrėžiate, kokie duomenys arba kintamieji bus naudojami funkcijos vykdymui. Parametrai yra deklaruojami funkcijos apibrėžimo metu, ir jie veikia kaip kintamieji, kurie gauna reikšmes, kai funkcija iškviečiama. | -**()**=>{...} //neturi parametrų  -**x**=>{...} //vienas parametras (kai vienas parametras galima ir nedėti į skliaustus  -(**x,y**)=>{...} //keli parametrai |
| Tai yra funkcijos dalis, kurioje rašote teiginius arba instrukcijas, kurios bus vykdomos, kai funkcija yra iškviečiama.  JavaScript kalboje funkcijos kūnas yra apribotas riestinėmis skliaustais **{}** | -x => {return x\*x} //block  - x => x\*x //ekvivalentiška ankstesniam pavyzdžiui jei kintamasis yra tas pats galime ir nedėti į {} |

### Lokalūs kintamieji

Lokalūs kintamieji (angl. local variables) yra kintamieji, kurie yra apibrėžti tam tikrame bloke kodo ir yra pasiekiami tik šiame bloke ar jo viduje. Lokalūs kintamieji nėra matomi išorės blokuose ar funkcijose, kurių viduje jie nebuvo apibrėžti

. function exampleFunction() {

    // Lokalus kintamasis apibrėžtas funkcijos viduje

    let localVar = "Aš esu lokalus kintamasis";

    console.log(localVar); // Output: Aš esu lokalus kintamasis

}

// Šis blokas neturi prieigos prie localVar, nes jis apibrėžtas funkcijoje

console.log(localVar); // Output: ReferenceError: localVar is not defined

Čia **localVar** yra lokalus kintamasis, apibrėžtas funkcijos **exampleFunction** viduje. Jis yra pasiekiamas tik šioje funkcijoje, todėl išorės blokas neturi prieigos prie jo.

Taip pat, lokalūs kintamieji gali būti apibrėžti ir kituose blokuose, pavyzdžiui, cikluose ar sąlygos sakinio blokuose:

function exampleFunction() {

    if (true) {

        // Lokalus kintamasis apibrėžtas sąlygos sakinio bloke

        let localVarInIf = "Aš esu lokalus kintamasis sąlygos sakinio bloke";

        console.log(localVarInIf); // Output: Aš esu lokalus kintamasis sąlygos sakinio bloke

    }

    // localVarInIf negali būti pasiektas iš čia, nes jis buvo apibrėžtas kitoje blokoje

    console.log(localVarInIf); // Output: ReferenceError: localVarInIf is not defined

}

Šis pavyzdys parodo, kaip lokalūs kintamieji yra susiję su blokų struktūra ir prieigomis

### Išorininiai kintamieji

Išoriniai kintamieji, taip pat žinomi kaip išoriniai arba globalūs kintamieji, yra kintamieji, apibrėžti už tam tikro bloko, funkcijos arba srities ribų. Priešingai nei vietiniai kintamieji, išoriniai kintamieji pasiekiami iš kelių blokų ar funkcijų, priklausomai nuo to, kur jie yra apibrėžti. Jie turi platesnį taikymo lauką, išplečiantis už bloko ar funkcijos ribų, kuriuose jie yra apibrėžti.

// Išorinis kintamasis apibrėžtas už bet kokio bloko ar funkcijos ribų

let isorinisVar = "Aš esu išorinis kintamasis";

function pavyzdineFunkcija() {

    // Prieiga prie išorinio kintamojo funkcijos viduje

    console.log(isorinisVar); // Rezultatas: Aš esu išorinis kintamasis

}

// Prieiga prie išorinio kintamojo funkcijos išorėje

console.log(isorinisVar); // Rezultatas: Aš esu išorinis kintamasis

Šiame pavyzdyje **isorinisVar** yra išorinis kintamasis, apibrėžtas globaliai, todėl jis pasiekiamas tiek funkcijos **pavyzdineFunkcija** viduje, tiek iš jos išorės.

Svarbu paminėti, kad nors globalūs kintamieji suteikia būdą bendrinti duomenis tarp skirtingų kodo dalių, dažnai remtis jais gali sukelti potencialių problemų, tokias kaip kintamųjų pavadinimų konfliktai ir sumažėjusi kodo priežiūra. Dažnai laikoma geru būdu riboti globalių kintamųjų naudojimą ir teikti pirmenybę vietiniams kintamiesiems, kai tai įmanoma.

### Scope chain

Scope chain (lietuviškai - srities grandinė) yra sąvoka, kuri apibūdina, kaip programavimo kalbos, turinčios blokų ar funkcijų sritis (scope), tvarko prieigą prie kintamųjų ir kitų kintamųjų bei funkcijų, esančių skirtinguose blokuose ar funkcijose.

JavaScript yra kalba, kuri naudoja rūšiavimą pagal sritis (lexical scoping). Tai reiškia, kad, kai funkcija yra apibrėžta, ji išlaiko prieigą prie kintamųjų, kurie buvo matomi funkcijos kūrimo metu, net jei funkcija yra vykdoma kitame kontekste.

function exampleFunction() {

    if (true) {

        // Lokalus kintamasis apibrėžtas sąlygos sakinio bloke

        let localVarInIf = "Aš esu lokalus kintamasis sąlygos sakinio bloke";

        console.log(localVarInIf); // Output: Aš esu lokalus kintamasis sąlygos sakinio bloke

    }

    // localVarInIf negali būti pasiektas iš čia, nes jis buvo apibrėžtas kitoje blokoje

    console.log(localVarInIf); // Output: ReferenceError: localVarInIf is not defined

}

function outerFunction() {

    let outerVar = "I am in the outer function";

    function innerFunction() {

        let innerVar = "I am in the inner function";

        // Prieiga prie innerVar ir outerVar

        console.log(innerVar); // Rezultatas: I am in the inner function

        console.log(outerVar); // Rezultatas: I am in the outer function

    }

    // Prieiga prie outerVar

    console.log(outerVar); // Rezultatas: I am in the outer function

    // Iškvietimas innerFunction

    innerFunction();

}

// Iškvietimas outerFunction

outerFunction();

Šiame pavyzdyje **innerFunction** turi prieigą prie savo pačios kintamojo **innerVar** ir taip pat prieigą prie **outerVar**, kuris yra apibrėžtas išorinėje funkcijoje (outerFunction). Tai vyksta dėl to, kad srities grandinė leidžia funkcijai prieiti prie kintamųjų iš išorinių sričių, kuriose ji yra apibrėžta.

|  |  |
| --- | --- |
| let a; //1  function myFunction(){      let a; //3      a = 15; //4      console.loga(a); //5  }  myFunction(); //2  console.log(a); //6 | 1. a yra apibrėžtas globaliame kontekste." 2. Iškviesta funkcija **myFunction()."** 3. a yra apibrėžtas funkcijos srities viduje. 4. Ar a yra apibrėžtas funkcijos srityje? TAIP! Priskirkite reikšmę 15. 5. a turi reikšmę 15. 6. a turi reikšmę 'undefined' globaliame kontekste. |

### Callback funcijos

**Callback** funkcijos yra funkcijos, kurios perduodamos kaip argumentai kitoms funkcijoms ir yra iškviečiamos arba vykdomos po tam tikrų įvykių arba po tam tikro laiko tarpo. Ši funkcijų koncepcija yra plačiai naudojama asinchroniniame programavime ir leidžia programuotojui valdyti, kas turi įvykti po tam tikrų įvykių arba operacijų pabaigos.

// Callback funkcija

function pranesimas(vardas) {

    console.log("Labas, " + vardas + "!");

}

// Funkcija, kuri priima kitą funkciją kaip argumentą (callback)

function pasveikinti(callback) {

    let vardas = "Jonai";

    callback(vardas);

}

// Iškvietimas su callback funkcija

pasveikinti(pranesimas); // Output: Labas, Jonai!

Šiame pavyzdyje **pranesimas** yra callback funkcija, o **pasveikinti** yra funkcija, kuri priima kitą funkciją kaip argumentą ir ją iškviečia. Įvykdžius pasveikinti(pranesimas), pranesimas funkcija bus iškviečiama su nurodytu vardu.

### Numatyti argumentai

Numatytieji argumentai funkcijose leidžia nustatyti numatytąsias reikšmes parametrams. Jeigu funkcijai iškviečiant nėra pateikta reikšmė tam tikram parametrui, bus naudojama numatytoji reikšmė. Ši funkcijų savybė yra naudinga, siekiant padaryti funkcijas lankstesnes ir tvarkyti įvairias situacijas.

// Funkcija su numatytosiomis reikšmėmis

function sveikinti(vardas = "Svečias", sveikinimas = "Labas") {

    console.log(sveikinimas + ", " + vardas + "!");

}

// Funkcijos iškvietimas be pateikiamų argumentų

sveikinti(); // Rezultatas: Labas, Svečias!

// Funkcijos iškvietimas su pateikiamais argumentais

sveikinti("Jonai", "Sveiki"); // Rezultatas: Sveiki, Jonai!

Šiame pavyzdyje sveikinti funkcija turi du parametrus (vardas ir sveikinimas), kuriems nustatytos numatytosios reikšmės. Jeigu funkcijai iškviečiant nebuvo pateikti jokie argumentai, bus naudojamos numatytosios reikšmės ("Svečias" vardui ir "Labas" sveikinimui).

### Rest operatorius

Rest parametrai yra funkcijų programavimo kalbose funkcionalumas, leidžiantis funkcijai priimti neapibrėžtą skaičių argumentų kaip masyvą. Ši funkcija yra naudinga, kai norite dirbti su kintamu skaičiumi parametrų funkcijos viduje.

JavaScript kalboje rest parametrai pažymimi trimis taškais **(...)** prieš parametro pavadinimą. Rest parametras surenka likusius argumentus į masyvą, net jei argumentų nėra.

// Funkcija su rest parametru

function suma(...skaiciai) {

    let suma = 0;

    for (let skaicius of skaiciai) {

        suma += skaicius;

    }

    return suma;

}

// Funkcijos iškvietimas su skirtingais argumentų skaičiais

console.log(suma(1, 2, 3)); // Rezultatas: 6

console.log(suma(4, 5, 6, 7)); // Rezultatas: 22

console.log(suma()); // Rezultatas: 0 (be argumentų)

## JS Masyvai

Masyvas yra duomenų struktūra, kuri leidžia saugoti ir valdyti daugelio elementų sąrašus viename kintamajame. Masyvas leidžia pasiekti kiekvieną elementą pagal jo poziciją (indeksą) ir atlikti įvairias operacijas, pavyzdžiui, pridėti naują elementą, pašalinti esamą elementą arba tiesiog apdoroti visus masyvo elementus.

**Pagrindinės masyvų savybės:**

* **Indeksavimas:** Masyvo elementai yra indeksuojami, pradedant nuo nulio. Tai reiškia, kad **pirmasis elementas** **turi indeksą 0, antrasis - 1**, ir t.t.
* **Dinaminė dydis:** Dauguma programavimo kalbų leidžia keisti masyvo dydį, pridedant naujus elementus arba pašalinant esamus.
* **Skirtingų tipų elementai:** Masyve gali būti saugomi skirtingų tipų duomenys, įskaitant skaičius, tekstą, kitus masyvus arba net objektus.

// Masyvo apibrėžimas

let skaiciai = [1, 2, 3, 4, 5];

// Prieiga prie masyvo elemento pagal indeksą

console.log(skaiciai[2]); // Rezultatas: 3

// Iteravimas per masyvo elementus

for (let i = 0; i < skaiciai.length; i++) {

    console.log(skaiciai[i]);

}

// Pridėti naują elementą į masyvą

skaiciai.push(6);

// Masyvo ilgis po naujo elemento pridėjimo

console.log(skaiciai.length); // Rezultatas: 6

**toString()** metodas paverčia masyvą į **tekstinę eilutę**, kurioje atskiriama kiekviena masyvo reikšmė kableliu.

Tai yra pagrindinis metodas, kurį masyvas paveldi iš **Object.prototype.**

let masyvas = [1, 2, 3, "tekstas"];

let tekstas = masyvas.toString();

console.log(tekstas); // Rezultatas: "1,2,3,tekstas"

**join (separator)** metodas jungia masyvo elementus į vieną tekstą eilutę, naudojant nurodytą skirtuką (separatorių) tarp elementų.

Jei nenustatote separatoriaus, pagal nutylėjimą bus naudojamas **kablelis**.

let masyvas = [1, 2, 3, "tekstas"];

let tekstas = masyvas.join(" - ");

console.log(tekstas); // Rezultatas: "1 - 2 - 3 - tekstas"

### Masyvo ilgis

### Masyvo metodai

Šie metodai (**unshift, shift, pop, push**) yra JavaScript masyvų metodai, kurie naudojami keisti masyvo turinį:

|  |  |  |  |
| --- | --- | --- | --- |
| **unshift** | **shift** | **pop** | **push** |
| **PRIEKIS** | | **GALAS** | |
| metodas **prideda** vieną arba kelias reikšmes į masyvo **pradžią**.  Grąžina naujo masyvo ilgį.  let masyvas = [2, 3, 4];  masyvas.unshift(1);  console.log(masyvas); // Rezultatas: [1, 2, 3, 4] | metodas **pašalina** **pirmą** masyvo elementą ir grąžina jį.  Masyvo ilgis mažėja.  let masyvas = [1, 2, 3, 4];  let pirmasElementas = masyvas.shift();  console.log(pirmasElementas); // Rezultatas: 1  console.log(masyvas); // Rezultatas: [2, 3, 4] | metodas **pašalina paskutinį** masyvo elementą ir grąžina jį.  Masyvo ilgis mažėja.  let masyvas = [1, 2, 3, 4];  let paskutinisElementas = masyvas.pop();  console.log(paskutinisElementas); // Rezultatas: 4  console.log(masyvas); // Rezultatas: [1, 2, 3] | metodas **prideda vieną** arba **kelias** reikšmes į masyvo **pabaigą.**  Grąžina naujo masyvo ilgį.  let masyvas = [1, 2, 3];  masyvas.push(4);  console.log(masyvas); // Rezultatas: [1, 2, 3, 4] |

**concat():**

metodas, kuris naudojamas sujungti du ar daugiau masyvų į vieną naują masyvą. Šis metodas nekeičia esamų masyvų, o sukuria naują masyvą, kuriame yra visi sujungti elementai.

**Pagrindinės savybės:**

* **Grąžina naują masyvą**: concat() metodas grąžina naują masyvą, kuriame yra visi originalių masyvų elementai.
* **Nepakeičia esamų masyvų**: Originalūs masyvai lieka nepakitę.
* **Gali sujungti daugiau nei du masyvus**: Jūs galite nurodyti bet kokį skaičių masyvų kaip argumentus.

let masyvas1 = [1, 2, 3];

let masyvas2 = [4, 5, 6];

let masyvas3 = [7, 8, 9];

let naujasMasyvas = masyvas1.concat(masyvas2, masyvas3);

console.log(naujasMasyvas);

// Rezultatas: [1, 2, 3, 4, 5, 6, 7, 8, 9]

console.log(masyvas1);

// Originalus masyvas1 nepasikeitė: [1, 2, 3]

Šiame pavyzdyje **concat()** metodas yra naudojamas sujungti tris masyvus į naują masyvą **naujasMasyvas.** Originalūs masyvai **(masyvas1, masyvas2, masyvas3**) lieka nepakitę.

**forEach:**

Norint išspausdinti kiekvieną masyvo elementą, galima naudoti ciklą, pvz., **for** ciklą arba f**orEach** metodą. Štai kaip tai galima padaryti su for ciklu ir forEach metodu.

Visi šie būdai leis jums išspausdinti kiekvieną masyvo elementą atskirai.

|  |  |  |  |
| --- | --- | --- | --- |
| **Naudojant for ciklą:** | **Naudojant forEach metodą:** | **Naudojant for...of ciklą (ES6 ir naujesnės versijos):** |  |
| let masyvas = [1, 2, 3, 4, 5];  for (let i = 0; i < masyvas.length; i++) {      console.log(masyvas[i]);  } | let masyvas = [1, 2, 3, 4, 5];  masyvas.forEach(function(elementas) {      console.log(elementas);  }); | let masyvas = [1, 2, 3, 4, 5];  for (let elementas of masyvas) {      console.log(elementas);  } |  |

**map():**

metodas, kuris leidžia kiekvieną masyvo elementą pakeisti ir grąžinti naują masyvą, kuriame yra modifikuoti elementai. Šis metodas nekeičia originalaus masyvo, bet grąžina naują masyvą su modifikuotais elementais.

let masyvas = [1, 2, 3, 4, 5];

let kvadratuMasyvas = masyvas.map(function(elementas) {

    return elementas \* elementas;

});

console.log(kvadratuMasyvas);

// Rezultatas: [1, 4, 9, 16, 25]

console.log(masyvas);

// Originalus masyvas nepasikeitė: [1, 2, 3, 4, 5]

Šiame pavyzdyje map() metodas yra naudojamas sukurti naują masyvą kvadratuMasyvas, kuriame kiekvienas elementas yra originalaus masyvo kvadratas. Originalus masyvas masyvas lieka nepakitęs.

map() yra naudingas, kai norite sukurti naują masyvą, kurio elementai yra modifikuoti pagal tam tikrą logiką. Jūs nurodote funkciją, kuri apibrėžia, kaip keisti kiekvieną elementą, ir map() sukurs naują masyvą pagal šią logiką.

**filter():**

metodas, kuris leidžia filtruoti masyvo elementus pagal nurodytą sąlygą ir grąžinti naują masyvą, kuriame yra tik tie elementai, kurie tenkina sąlygą. Šis metodas nekeičia originalaus masyvo, bet grąžina naują masyvą su išfiltruotais elementais.

let masyvas = [1, 2, 3, 4, 5];

let lyginiaiElementai = masyvas.filter(function(elementas) {

    return elementas % 2 === 0;

});

console.log(lyginiaiElementai);

// Rezultatas: [2, 4]

console.log(masyvas);

// Originalus masyvas nepasikeitė: [1, 2, 3, 4, 5]

Šiame pavyzdyje filter() metodas yra naudojamas sukurti naują masyvą lyginiaiElementai, kuriame yra tik tie elementai, kurie yra lyginiai. Originalus masyvas masyvas lieka nepakitęs.

filter() yra naudingas, kai norite gauti naują masyvą, kuriame yra tik tie elementai, kurie atitinka tam tikrą sąlygą. Jūs nurodote funkciją, kuri apibrėžia sąlygą, ir filter() sukurs naują masyvą pagal šią sąlygą.

**Primityvūs ir Nuorodos Tipai:**

JavaScript kalba duoda galimybę klasifikuoti kintamuosius į dvi pagrindines kategorijas: **primityvūs tipai** ir **nuorodos tipai**. **Primityvūs** tipai saugo **paprastas reikšmes**, o **nuorodos** tipai rodo į **sudėtingesnius objektus** ar struktūras, kuriose yra laikomi duomenys.

1. **Primityvūs Tipai:**

* Primityvūs tipai yra paprasti duomenų tipai, kurie saugo konkretų duomenų elementą.
* Pagrindiniai primityvūs tipai yra: **number** (skaičius), **string** (tekstas), **boolean** (loginis), **null**, ir **undefined**.
* Primityvūs tipai yra saugomi tiesiogiai kintamojo reikšmėje.

Pavyzdys:

let skaicius = 42; // number

let tekstas = "Labas"; // string

let tiesaArbaNetiesa = true; // boolean

let nulis = null; // null

let neapibreztas; // undefined

1. **Nuorodos Tipai:**

* **Nuorodos tipai** yra **objektai**, kuriuose saugomi daug sudėtingesni duomenys arba struktūros.
* Pagrindiniai nuorodos tipai yra: **object** (objektas), **array** (masyvas), **function** (funkcija), ir kiti specialūs objektai.
* Nuorodos tipai nurodo vieną vietą atmintyje, kur yra laikomi duomenys, o ne patys duomenys tiesiogiai.

Pavyzdys:

let objektas = { vardas: "Jonas", amzius: 30 }; // object

let masyvas = [1, 2, 3]; // array

let funkcija = function() { console.log("Labas!"); }; // function

**spread syntax:**

Ši sintaksė leidžia išskleisti (spread) elementus iš masyvo ar objekto ir naudoti juos naujame masyve ar objekte.

let pirmasMasyvas = [1, 2, 3];

let antrasMasyvas = [4, 5, 6];

let sujungtasMasyvas = [...pirmasMasyvas, ...antrasMasyvas];

console.log(sujungtasMasyvas);

// Rezultatas: [1, 2, 3, 4, 5, 6]

## JS Objektai

Objektai yra pagrindinė duomenų struktūra daugumoje programavimo kalbų, įskaitant JavaScript. Objektai leidžia organizuoti ir saugoti duomenis taip, kad jie būtų lengvai pasiekiami ir valdomi. Objektai gali turėti **savybes** ir **metodus**.

|  |  |
| --- | --- |
| **Objekto Savybės:** | **Objekto Metodai:** |
| Objekto savybės yra poros, kurios susideda iš **raktų** (vadinamų savybių pavadinimais) ir **reikšmių**. Savybės gali būti skaičiai, tekstas, kiti objektai, funkcijos ir kt.  let zmogus = {      vardas: "Jonas",      amzius: 30,      miestas: "Vilnius"  };  Šiame pavyzdyje vardas, amzius ir miestas yra objekto savybės. | Objekto metodai yra funkcijos, kurios yra priskirtos objektui kaip savybės. Metodai gali vykdyti veiksmus su objektu arba grąžinti tam tikrą informaciją.  let automobilis = {      marke: "Toyota",      modelis: "Camry",      metai: 2022,      informacija: function() {          return this.marke + " " + this.modelis + ", " + this.metai + " metai.";      }  };  console.log(automobilis.informacija());  // Rezultatas: "Toyota Camry, 2022 metai."  Šiame pavyzdyje informacija yra objekto metodas. |

Objekto reikšmes galima atspausdinti naudojant kintamųjų pavadinimus (raktus) ir taip pasiekiant atitinkamas reikšmes. Tai galima padaryti keliais būdais: naudojant ciklą, naudojant **for...in** ciklą, arba tiesiog naudojant kintamųjų pavadinimus tiesioginėje **console.log** funkcijoje.

|  |  |  |
| --- | --- | --- |
| **Naudojant Ciklą:** | **Naudojant for...in Ciklą:** | **Tiesioginė console.log:** |
| let zmogus = {      vardas: "Jonas",      amzius: 30,      miestas: "Vilnius"  };  for (let savybe in zmogus) {      console.log(savybe + ": " + zmogus[savybe]);  } | let automobilis = {      marke: "Toyota",      modelis: "Camry",      metai: 2022  };  for (let savybe in automobilis) {      console.log(savybe + ": " + automobilis[savybe]);  } | let gyvunas = {      rusis: "Šuo",      vardas: "Rikis",      amzius: 3  };  console.log("Rūšis: " + gyvunas.rusis);  console.log("Vardas: " + gyvunas.vardas);  console.log("Amžius: " + gyvunas.amzius); |

**Naujos reikšmės pridėjimas:**

Norint pridėti naują reikšmę objektui, galima tiesiog priskirti reikšmę naujam raktui (savybei). Tai galima padaryti taip:

let zmogus = {

    vardas: "Jonas",

    amzius: 30,

    miestas: "Vilnius"

};

// Pridedame naują savybę "lytis" su reikšme "Vyras"

zmogus.lytis = "Vyras";

console.log(zmogus);

Arba galima pridėti kelias naujas savybes vienu metu:

et automobilis = {

    marke: "Toyota",

    modelis: "Camry",

    metai: 2022

};

// Pridedame kelias naujas savybes

automobilis.kuroTipas = "Benzinas";

automobilis.spalva = "Balta";

console.log(automobilis);

Jei jau egzistuoja savybė su tuo pačiu pavadinimu, ji bus pakeista naująja reikšme:

let zmogus = {

    vardas: "Jonas",

    amzius: 30,

    miestas: "Vilnius"

};

// Pakeičiame savybę "miestas" naująja reikšme

zmogus.miestas = "Kaunas";

console.log(zmogus);

**Kaip pašalinti objekto reikšmę:**

Objekto savybę (reikšmę) galima pašalinti naudojant **delete** operatorių. Šis operatorius leidžia pašalinti objekto savybę pagal jos pavadinimą (raktą).

let zmogus = {

    vardas: "Jonas",

    amzius: 30,

    miestas: "Vilnius"

};

// Pašaliname savybę "miestas"

delete zmogus.miestas;

Šiuo atveju, po **delete zmogus**.**miestas**;, objekte nebebus savybės "**miestas"**.

Svarbu atkreipti dėmesį, kad **delete** operatorius pašalina tik savybę, bet nereikšmę ir nekeičia objekto dydžio. Jei bandysite pasiekti pašalintą savybę, gausite **undefined.**

console.log(zmogus.miestas);

// Rezultatas: undefined

Tačiau būkite atsargūs naudojant delete, nes kai kuriais atvejais yra geriau tiesiog priskirti undefined arba null reikšmę savybei, jei norite ją "išvalyti", o ne pašalinti.

**Kaip patikrinti ar egzistuoja raktas (key):**

|  |  |  |
| --- | --- | --- |
| **hasOwnProperty metodas:** | **Tikrinimas su undefined:** | **in operatorius:** |
| let zmogus = {      vardas: "Jonas",      amzius: 30,      miestas: "Vilnius"  };  if (zmogus.hasOwnProperty(  "miestas")) {      console.log("Objekte yra savybė 'miestas'.");  } else {      console.log("Objekte nėra savybės 'miestas'.");  } | et zmogus = {      vardas: "Jonas",      amzius: 30  };  if (zmogus.miestas !== undefined) {      console.log("Objekte yra savybė 'miestas'.");  } else {      console.log("Objekte nėra savybės 'miestas'.");  } | let zmogus = {      vardas: "Jonas",      amzius: 30  };  if ("miestas" in zmogus) {      console.log("Objekte yra savybė 'miestas'.");  } else {      console.log("Objekte nėra savybės 'miestas'.");  }  **Galima ir trumpinti:**  **Console.log(“miestas” in “zmogus”); //false** |

**!for of in Objects:**

**for...of** ciklas tiesiogiai negali būti naudojamas su objektais, nes objektai pagal nutylėjimą nėra iterable. Tačiau, jei norite iteruoti per objekto savybes, galite naudoti **Object.keys()**, **Object.values()** arba **Object.entries()** kartu su **for...of**.

Pavyzdys naudojant **Object.entries():**

let zmogus = {

    vardas: "Jonas",

    amzius: 30,

    miestas: "Vilnius"

};

for (let [raktas, reiksme] of Object.entries(zmogus)) {

    console.log(raktas + ": " + reiksme);

}

Object.entries(zmogus) grąžina masyvą, kurio kiekvienas elementas yra pora [raktas, reikšmė], o for...of ciklas leidžia patogiai iteruoti per šį masyvą.

Alternatyviai, galite naudoti Object.keys() arba Object.values():

|  |  |
| --- | --- |
| let zmogus = {      vardas: "Jonas",      amzius: 30,      miestas: "Vilnius"  };  for (let raktas of Object.keys(zmogus)) {      console.log(raktas + ": " + zmogus[raktas]);  }  javascript | let zmogus = {      vardas: "Jonas",      amzius: 30,      miestas: "Vilnius"  };  for (let reiksme of Object.values(zmogus)) {      console.log(reiksme);  } |

**Kaip pridėti funciją į objetą:**

Norėdami pridėti funkciją į objektą, galite tiesiog priskirti ją kaip savybę objektui. Funkciją galite išreikšti kaip funkcijos išraišką arba tiesiogiai kaip funkcijos deklaraciją. Štai keletas pavyzdžių:

|  |  |  |
| --- | --- | --- |
| **Funkcijos Išraiška:** | **Funkcijos Deklaracija:** | **Nauja Funkcijos Išraiška:** |
| let zmogus = {      vardas: "Jonas",      amzius: 30,      miestas: "Vilnius",      pasisveikink: function() {          console.log("Labas, aš esu " + this.vardas + "!");      }  };  zmogus.pasisveikink();  // Rezultatas: "Labas, aš esu Jonas!" | let zmogus = {      vardas: "Jonas",      amzius: 30,      miestas: "Vilnius"  };  zmogus.pasisveikink = function() {      console.log("Labas, aš esu " + this.vardas + "!");  };  zmogus.pasisveikink();  // Rezultatas: "Labas, aš esu Jonas!" | function pasisveikink() {      console.log("Labas, aš esu " + this.vardas + "!");  }  let zmogus = {      vardas: "Jonas",      amzius: 30,      miestas: "Vilnius",      pasisveikink: pasisveikink  };  zmogus.pasisveikink();  // Rezultatas: "Labas, aš esu Jonas!" |

**Kaip objekte įrašyti metodą (funkciją):**

Objekte metodą galima įrašyti kaip funkciją, kuri yra priskiriama objekto savybei. Toks metodas turi turėti prieigą prie objekto savybių ir gali atlikti veiksmus su objektu arba grąžinti informaciją. Štai pavyzdys, kaip įrašyti metodą į objektą:

let zmogus = {

    vardas: "Jonas",

    amzius: 30,

    miestas: "Vilnius",

    pasisveikink: function() {

        console.log("Labas, aš esu " + this.vardas + "!");

    }

};

// Kviečiame objekto metodą

zmogus.pasisveikink();

// Rezultatas: "Labas, aš esu Jonas!"

Objekto metodas čia yra funkcija pasisveikink, kuri yra priskiriama objekto zmogus savybei. Šio metodo viduje this nuoroda į pats objektą, leidžianti pasiekti jo savybes.

Taip pat galite įrašyti metodą kitaip, naudodami naujesnę sintaksę, kurioje nereikia rašyti žodžio function:

let zmogus = {

    vardas: "Jonas",

    amzius: 30,

    miestas: "Vilnius",

    pasisveikink() {

        console.log("Labas, aš esu " + this.vardas + "!");

    }

};

// Kviečiame objekto metodą

zmogus.pasisveikink();

// Rezultatas: "Labas, aš esu Jonas!"

**Kaip paduoti objektą į funciją:**

Objektą galima perduoti funkcijai kaip parametrą. Funkcija tada gali naudotis objekto savybėmis arba atlikti operacijas su juo. Štai kaip tai galima padaryti:

// Funkcija, kuri priima objektą kaip parametrą

function spausdintiInformacija(zmogus) {

    console.log("Vardas: " + zmogus.vardas);

    console.log("Amžius: " + zmogus.amzius);

    console.log("Miestas: " + zmogus.miestas);

}

// Sukuriame objektą

let zmogusObjektas = {

    vardas: "Jonas",

    amzius: 30,

    miestas: "Vilnius"

};

// Iškviečiame funkciją ir perduodame objektą kaip parametrą

spausdintiInformacija(zmogusObjektas);

Šiuo pavyzdžiu sukūrėme funkciją spausdintiInformacija, kuri priima objektą kaip parametrą ir spausdina jo savybes. Tada sukūrėme objektą zmogusObjektas ir iškvietėme funkciją, perduodant objektą kaip argumentą.

Be to, galite perduoti ne tik visą objektą, bet ir tik jo savybes arba reikšmes, kaip parametrus funkcijai:

function spausdintiSavybe(savybe, reiksme) {

    console.log(savybe + ": " + reiksme);

}

let zmogusObjektas = {

    vardas: "Jonas",

    amzius: 30,

    miestas: "Vilnius"

};

// Iškviečiame funkciją, perduodant objekto savybes

spausdintiSavybe("Vardas", zmogusObjektas.vardas);

spausdintiSavybe("Amžius", zmogusObjektas.amzius);

spausdintiSavybe("Miestas", zmogusObjektas.miestas);