Typy danych w JavaScript

W języku JavaScript dostępnych jest siedem typów wbudowanych: **null, undefined, boolean, number, string, object i symbol.** Mogą one być identyfikowane za pomocą operatora typeof. Zmienne nie mają typów, ale zawarte w nich wartości już tak. Typy te definiują natywne zachowanie wartości.
Wielu projektantów będzie przyjmować, że terminy „niezdefiniowana” (ang. undefined) i „niezadeklarowana” znaczą mniej więcej to samo. W języku JavaScript są to jednak dość odmienne pojęcia.

Wartość typu **undefined** może być przechowywana przez zadeklarowaną zmienną.

Termin „**niezadeklarowana**” oznacza zmienną, która nigdy nie została zadeklarowana.

Niestety, w języku JavaScript w pewnym stopniu te dwa terminy są łączone, nie tylko w komunikatach o błędach (np. „Błąd ReferenceError: zmienna b nie została zdefiniowana”), ale też w wartościach zwracanych operatora typeof, który w przypadku obu terminów przekazuje "undefined". Zabezpieczenie (zapobiegające błędowi) operatora typeof użyte w odniesieniu do niezadeklarowanej zmiennej może być jednak pomocne w niektórych sytuacjach.

Operatory w JavaScript

Operatory przypisania

Czyli operatory, które służą do przypisania do zmiennej jakiejś wartości, pola, obiektu itp.
Poniżej zamieszczamy przykłady dla **x = 10** i **y = 5**:

```
{
    let x = 5;
    x += 3; //równoznaczne z x = x + 3;
    console.log(x);
}
{
    let x = 5;
    x -= 3; //równoznaczne z x = x - 3;
    console.log(x);
}
{
    let x = 5;
    x *= 3; //równoznaczne z x = x * 3;
    console.log(x); //15
}
{
    let x = 12;
    x /= 3; //równoznaczne z x = x / 3;
    console.log(x); //4
}
{
    let x = 11;
    x %= 3; //równoznaczne z x = x % 3;
    console.log(x); //2
}
```

Zwiększenie lub zmniejszenie wartości o 1 możemy wykonać tak jak w powyższych przykładach, ale też możemy skorzystać z operatorów inkrementacji i dekrementacji:

```
let x = 5;
x++; //równoznaczne z x = x + 1;
console.log(x); //6

let y = 5;
y--; //równoznaczne z y = y - 1;
```

Gdy występuje przed zmienną, zwiększenie jej wartości nastąpi w danej instrukcji. Jeżeli występuje po zmiennej, zwiększona wartość wystąpi dopiero w kolejnej instrukcji.

```
let x = 5;
console.log(x++); //5
console.log(x); //6

let y = 5;
if (y-- < 5) { //nie zadziała
    console.log(y); //4
}
```

```
let x = 5;
console.log(++x); //6
console.log(x); //6

let y = 5;
if (--y < 5) {//zadziała
    console.log(y); //4
}
```

Z powodu takiego zachowania, dość często inkrementację/dekrementację wykonuje się w oddzielnej linii - tak by poprawić czytelność kodu:

```
let y = 5;
y++;

if (y > 5) {
    console.log(y);
}
```

Operatory logiczne

Operatory logiczne używane będą przez nas głównie w instrukcjach warunkowych. Służą do sprawdzania czy dane warunki są spełnione zwracając w wyniku true lub false.

Operator	Opis	Przykład	Wynik
&&	and (i)	(x < 10 && y > 1)	Prawda, bo x jest mniejsze od 10 **i** y jest większe od 1
||	or (lub)	(x > 8 || y > 1)	Prawda, bo x nie jest większe od 8, ale y jest większe od 1
^	xor (jeden z, ale nie dwa równocześnie)	(x === 6 ^ y === 3)	Fałsz, bo obydwa są prawdziwe
!	not (negacja)	!(x === y)	Prawda, bo negujemy to, że x === y

```
{
    //&& - operator "i" - wszystkie warunki muszą być spełnione
    let x = 6;
    let y = 3;
    console.log(x > 3 && y > 3); //false bo drugie równanie nie jest prawdą
}

{
    //|| - operator "lub" - przynajmniej jeden warunek musi być spełniony
    let x = 0;
    let y = 3;
    console.log(x > 3 || y > 2); //true bo drugi warunek jest spełniony
}

{
    //^ - operator "xor" - przynajmniej jeden warunek musi być spełniony, ale nie wszystki
    let x = 3;
    let y = 3;
    let z = 5;
    console.log(x > 2 ^ y < z); //0 czyli false, bo wszystkie są spełnione
}

{
    //! - "negacja" czyli odwrócenie true na false i odwrotnie
    let x = 2;
    let y = 0;
    console.log(!true); //false
    console.log(!false); //true
    console.log(x && y); //false bo y === 0
    console.log(!(x && y)) //true
}
```

Operatory logiczne w równaniach

Powyższe operatory możemy też wykorzystać w momencie podstawiania pod zmienną nowej wartości. Dzięki temu możemy skrócić nasz zapis pozbywając się dodatkowych instrukcji if.

Pierwszym z nich jest operator &&. Jeżeli pierwsza wartość nie jest falsy (0, "", null, undefined, NaN, document.all), wtedy podstawiana jest druga wartość. W przeciwnym razie wybierana jest pierwsza wartość:

```
{
    const a = 100 && 300;
    console.log(a); //300 - a jest prawdą, więc weź drugą wartość
}
{
    const a = 200 && 0;
    console.log(a); //0 - a jest prawdą, więc weź drugą wartość
}
{
    const a = 0;
    const b = 200;
    const c = a && b;
    console.log(c); //0 bo a jest falsy, więc zostań na niej
}
```

Kolejny operator - || podstawia pod zmienną wartość pierwszą w przypadku, gdy jest ona inna od falsy. W przeciwnym wypadku podstawiana jest wartość druga.

```
{
    const text = "kot" || "brak"
    console.log(text); //"kot"
}
{
    const text = "" || "pies"
    console.log(text); //"pies"
}
```

```
{
    const a = 0 || 200;
    console.log(a); //200
}
{
    const a = 200;
    const b = 100;
    const c = a || b;
    console.log(c); //200
}
{
    const tab = ["ala", "bala"]; //3 elementu nie ma czyli undefined
    const x = tab[2] || "brak";
    console.log(x); //"brak"
}
```

Wartość false oznacza, że w powyższych testach zmienna a nie może mieć także wartości 0 lub "". W wielu sytuacjach będzie to problematyczne.

W nowym Javascript mamy też operator ??, który działa bardzo podobnie, z tym, że prawa wartość zwracana jest tylko w przypadku, gdy lewa ma wartość nie false, a null (undefined lub null):

```
{
    const x = null || 10;
    const y = null ?? 10;
    console.log(x); //10
    console.log(y); //10
}
{
    const x = "" || 10;
    const y = "" ?? 10;
    console.log(x); //10
    console.log(y); //""
}
{
    const tab = ["ala", "bala"];
    const x = tab[2] || "brak";
    const y = tab[2] ?? "brak";
    console.log(x); //"brak"
    console.log(y); //"brak"
}
```

W EcmaScript 2021 powyższe zapisy możemy jeszcze bardziej uprościć za pomocą operatorów ||=, ??= i &&=.

Operator **||=** podstawi nową wartość tylko wtedy, gdy obecna wartość jest falsy:

```
{
    let a = 0;
    let b = "kot"
    a ||= b
    console.log(a); //"kot"
}
{
    let a = "pies";
    let b = "kot"
    a ||= b
    console.log(a); //"pies"
}
```

Operator **&&=** podstawi pod zmienną nową wartość gdy obecna wartość jest inna niż falsy (jest truthy):

```
let a = 1;
let b = 0;

a &&= 20;
console.log(a); //20

b &&= 20;
console.log(b); //0
```

Operator ??= podstawi nową wartość, gdy obecna wartość jest nullish (null lub undefined):

```
{
    let a = null;
    a ??= 200;
    console.log(a); //200
}
{
    let a = 0;
    a ??= 200;
    console.log(a); //0
}
{
    let a = {
        nr : 100
    }
    a.something ??= 200;
    a.nr ??= 300;
    console.log(a.something); //200
    console.log(a.nr); //100
}
```

Operatory porównania

Możemy je znaleźć między innymi w instrukcjach warunkowych. Służą one do porównywania lewej strony równania do prawej, w wyniku której zawsze zwracana jest prawda albo fałsz (true/false).

```
{
    //== - porównuje obie wartości bez porównania ich typów
    let a = 10;
    console.log(a == 10) //true
    console.log(a == "10") //true
}
{
    //!= - czy wartości są różne, bez sprawdzenia typu
    let a = 10;
    console.log(a != 20) //true
    console.log(a != 10) //false
    console.log(a != "10") //false
}
{
    //=== - porównuje obie wartości i ich typ
    let a = 10;
    console.log(a === 10) //true
    console.log(a === "10") //false
}
{
    //!== - czy wartości lub typy są różne
    let a = 10;
    console.log(a !== 10) //false
    console.log(a !== "10") //true
}
{
    //< i > - mniejsze i większe
    let a = 10;
    let b = 20;
    console.log(a < 20) //true
    console.log(a < b) //true
    console.log(a > b) //false
}
{
    //<= i >= - mniejsze-równe i większe-równe
    let a = 10;
    let b = 20;
    let c = 10;
    console.log(a <= b) //true
    console.log(a <= c) //true
}
```

Falsy

Tworząc warunki, nie musimy porównywać ze sobą dwóch wartości. Wartością false staje się każda z poniższych wartości. Są to tak zwane wartości **falsy**:

```
if (false) { ... }
if (null) { ... }
if (undefined) { ... }
if (0) { ... }
if (NaN) { ... }
if ("") { ... }
if (document.all) { ... }
```