

## Temat 11

### Temat: Obiekty2

#### Funkcja konstruująca obiekt - konstruktor

Na poprzedniej lekcji tworzyliśmy obiekty z różnymi własnościami. Gdybyśmy jednak chcieli stworzyć np. kilka obiektów dla różnych osób, musielibyśmy w znacznym stopniu powielać kod. Takie działanie zwiększyłoby niepotrzebnie jego długość. Aby usprawnić tworzenie wielu zbliżonych obiektów możemy wykorzystać pewien szablon, **funkcję konstruującą (konstruktor)** obiekt i wywołać ją następnie wielokrotnie. Stanowi ona pewną formę do tworzenia obiektów. W konstruktorze wszystkie tworzone atrybuty i metody są poprzedzone słowem kluczowym **this**.

Aby teraz utworzyć nowe obiekty na bazie takiego szablonu korzystamy ze słowa kluczowego **new**, po którym następuje nazwa konstruktora z ewentualną listą argumentów w nawiasach okrągłych.

```
function Konstruktor(argumenty)
{
  this.nazwa_wlasnosci1 = (argument1);
  this.nazwa_wlasnosci2 = (argument2);
  ...
}
```

Warto zwrócić uwagę na to, że nazwy atrybutów i metod obiektu są w konstruktorze poprzedzone słowem kluczowym **this** i zamiast dwukropka jest znak równości (w porównaniu do metody tworzenia obiektu opisanej w poprzedniej lekcji).

#### Przykład 1. funkcja konstruktora

```
<script>
// funkcja konstrukcyjna
function myCar(a,b,c,d){
  this.Car = a;
  this.Year = b;
  this.Model = c;
  this.Cost = d;
}
const myCar1 = new myCar("Toyota",2015,"Corolla",40000);
console.log(myCar1);
const myCar2 = new myCar("Honda",2013,"Civic",20000);
console.log(myCar2);
const myCar3 = new myCar("Ford",2010,"Mustang",50000);
console.log(myCar3);
</script>
```

## Przykład 2. funkcja konstruktora

```
<p id="output"></p>
<script>
  const out = document.getElementById("output");
  function Point(x, y)
  {
    this.x = x;
    this.y = y;
  }
  const point = new Point(1,2);
  document.write("Współrzędne punktu:<br />");
  document.write("x = " + point.x + "<br />");
  document.write("y = " + point.y);
  out.innerHTML = `Współrzędne punktu: (${point.x}, ${point.y})`;
</script>
```

Powstała funkcja **Point** przyjmująca argumenty *x* i *y*. Ich wartości są przypisywane obiektowi dostępnemu przez wskazanie *this*. Będzie to obiekt utworzony dzięki operatorowi *new*. A zatem użycie konstrukcji `new Point(1,2)` spowoduje utworzenie nowego obiektu, przypisanie właściwościom *x* i *y* wartości 1 oraz 2 (jest to równoznaczne z utworzeniem tych właściwości) i podstawienie tego obiektu w miejscu użycia operatora *new*. Tym samym w skrypcie obiekt zostanie przypisany zmiennej *punkt*, a jego właściwości zostaną następnie odczytane i wyświetlone.

## Przykład 3. funkcja konstruktora

```
<p id="output"></p>
<script>
  const out = document.getElementById("output");
  function szpital(lekarz, specjalizacja, pacjent)
  {
    this.lekarz = lekarz;
    this.specjalizacja = specjalizacja;
    this.pacjent = pacjent;
  }
  function osoba(imie, nazwisko)
  {
    this.imie = imie;
    this.nazwisko = nazwisko;
  }

  const pacjent = new osoba("Jan", "Kowalski");
  const oddzial = new szpital("P.Nowak", "neurolog", pacjent);
  /* zwróć uwagę na to, że obiekt pacjent został wykorzystany
  przy tworzeniu obiektu oddzial */
  document.write("<b>lekarz: </b>" + oddzial.lekarz + "<br>");
  document.write("<b>specjalizacja: </b>" + oddzial.specjalizacja +
    "<br>");
  document.write("<b>pacjent: </b>" + oddzial.pacjent.imie + " " +
    oddzial.pacjent.nazwisko + "<br>");

  out.innerHTML = `Lekarz ${oddzial.specjalizacja} ${oddzial.lekarz}
    opiekuje się pacjentem ${oddzial.pacjent.nazwisko}`;
</script>
```

## Metody obiektów

Składowymi obiektu mogą być również funkcje, nazywane wówczas **metodami**. Funkcję dodaje się do obiektu w sposób analogiczny do innych właściwości. W przypadku literału wyglądałoby to tak:

```
{
  nazwa_funkcji : function(argumenty)
  {
    // treść funkcji
  }
  // inne składowe obiektu
}
```

W przypadku funkcji konstruującej obiekt schemat będzie podobny:

```
function Konstruktor(argumenty)
{
  this.nazwa_funkcji = function(argumenty)
  {
    // treść funkcji
  }
  // definicja innych składowych obiektu
}
```

Po takich definicjach funkcję można wywołać przy użyciu obiektu, w którym została umieszczona, stosując operator:

```
obiekt.nazwa_funkcji(argumenty);
```

### **Przykład 4. Metody**

```

<script>
const osoba={
  wiek:20,
  imie: "Kasia",
  // wartością może być funkcja, nazywana w obiekcie metodą
  umyjZeby: function(x){
    console.log("Myję zęby "+x+" minut");
  },
  zjedzSniadanie: function(y){
    console.log("Dzisiaj serwujemy "+y);
  },
  rodzina: {
    imie: "Ania",
    relacja: "siostra"
  }
}
console.log(osoba);
console.log(osoba.imie);
//console.log(osoba.ujmyZeby(10));
osoba.ujmyZeby(10);
//console.log(osoba.zjedzSniadanie("tosty"));
osoba.zjedzSniadanie("tosty")
</script>

```

Aby uzyskać dostęp do właściwości obiektu z metody tego samego obiektu, musisz użyć słowa kluczowego **this**.

#### Przykład 5a. Metody

```

<p id="output"></p>
<script>
  const out = document.getElementById("output");
  const osoba = {
    imie: "Anna",
    nazwisko: "Nowak",
    pobierzInformacje: function()
    {
      return `${this.imie} ${this.nazwisko}`;
    }
  };
  out.innerHTML = osoba.pobierzInformacje();

  /*przetestuj działanie funkcji dla całego obiektu
  usuń komentarz w poniższej linijce, zakomentuj poprzednią*/
  /*out.innerHTML = osoba;*/
</script>

```

## Dziedziczenie prototypowe

Programowanie obiektowe charakteryzuje się pewnymi mechanizmami. Jednym z podstawowych jest tak zwane **dziedziczenie**.

Dzieci dziedziczą po rodzicach pewne cechy. Może to kolor włosów, może kolor oczu, a może talent do rysowania. Część takich właściwości sobie pobierają od rodziców, ale i też mają swoje własne. Co ważne w drugą stronę to nie działa. Rodzice nigdy nie dziedziczą po dzieciach.

Na podobnej zasadzie działają obiekty w Javascript. Gdy tworzysz jakiś obiekt jakiegoś typu (np. Array, String, Number), możesz dla niego odpalać różne funkcjonalności, które są dziedziczone.

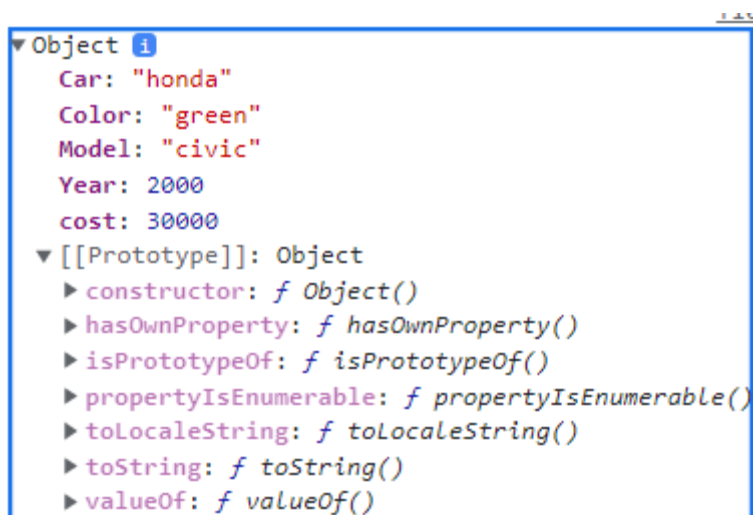
W Javascript występuje tak zwane **dziedziczenie prototypowe**. Oznacza to, że każdy obiekt dziedziczy właściwości i metody z innego obiektu - zwanego tutaj prototypem.

Gdy zbadasz kod utworzonego obiektu w konsoli, zobaczysz, że poza ustalonymi właściwościami ma on także specjalną właściwość `[[Prototype]]`.

Jest to referencja dodawana przez JavaScript praktycznie każdemu obiektowi. Wskazuje ona na inny obiekt, który jest prototypem danego obiektu, a z którego nasz obiekt może dziedziczyć funkcjonalności.

Jeżeli używamy jakiejś metody dla danego obiektu, JavaScript początkowo szuka tej funkcjonalności bezpośrednio w danym obiekcie. Jeżeli ją znajdzie - użyje jej. Jeżeli nie - za pomocą referencji przejdzie do prototypu danego obiektu i tam spróbuje użyć danej właściwości.

Prototyp obiektu także jest obiektem, więc także dostał swój prototyp. W razie potrzeby Javascript może więc przejść do kolejnego obiektu i tam poszukać danej funkcjonalności. Sytuacja taka będzie się powtarzać, aż do momentu w którym Javascript odnajdzie daną metodę lub właściwość, lub dojdzie do ostatniego obiektu w hierarchii, który już swojego `[[Prototype]]` już nie ma, a w zasadzie ma ustawione na null.



Możemy zatem skorzystać np. z metody wbudowanej w prototyp **toString**:

#### Przykład 5b. Metody

```
<p id="output"></p>
<script>
  const out = document.getElementById("output");
  const osoba = {
    imie: "Anna",
    nazwisko: "Nowak",
    toString: function()
    /* funkcja wbudowana toString to metoda pozwalająca
    potraktować obiekt jako ciąg znaków
    */
    {
      return `${this.imie} ${this.nazwisko}`;
    }
  };
  out.innerHTML = osoba;
</script>
```

#### Przykład 6. Metody

Skrypt tworzący obiekt `prostokat` posiadający dwie właściwości: `wysokosc` i `szerokosc` oraz dwie metody: `pole` i `obwod`. Właściwości są podczas tworzenia obiektu inicjalizowane wartością 0, a następnie są zmieniane na podstawie danych podanych przez użytkownika w oknie dialogowym. Natomiast metody są wykorzystane do wyświetlenia pola i obwodu prostokąta.

```
<p id="output1"></p>
<p id="output2"></p>
<script>
  const out1 = document.getElementById("output1");
  const out2 = document.getElementById("output2");
  const prostokat =
  {
    wysokosc: 0,
    szerokosc: 0,
    pole: function()
    {
      let wynik1 = this.wysokosc * this.szerokosc;
      out1.innerHTML = `Pole wynosi : ${wynik1}`;
    },
    obwod: function()
    {
      let wynik2 = 2 * this.wysokosc + 2 * this.szerokosc;
      out2.innerHTML = `Obwód wynosi : ${wynik2}`;
    }
  }
  prostokat.wysokosc = parseFloat(prompt('Podaj wysokość: '));
  prostokat.szerokosc = parseFloat(prompt('Podaj szerokość: '));
  prostokat.pole();
  prostokat.obwod();
</script>
```

## Wykorzystanie konstruktorów i metod do tworzenia obiektów

**Uwaga :** Dobrą praktyką jest pisanie pierwszej litery funkcji konstruktora wielką literą.

### Przykład 7. Metody i funkcja konstruktora

```
<p id="output"></p>
<script>
  const out = document.getElementById("output");
  function Person(imie, nazwisko, wiek)
  {
    this.name = imie;
    this.surname = nazwisko;
    this.age = wiek;
    this.toString = function()
    //wykorzystujemy funkcję wbudowaną toString
    {
      return `${this.name}  ${this.surname}`;
    }
  }
  const person1 = new Person("Arek", "Włodarczyk", 15);
  const person2 = new Person("Wiola", "Dowolna", 16);
  const person3 = new Person("Agnieszka", "Pośpieszna", 20);
  console.log(person1);

  out.innerHTML = `${person1}<br> ${person2}<br> ${person3}`;
</script>
```

### Przykład 8. Metody i funkcja konstruktora

Skrypt tworzący dwa obiekty `prostokat1` i `prostokat2` za pomocą konstruktora `Prostokat()`. Właściwości podczas tworzenia obiektu są inicjalizowane wartością 0, a następnie są zmieniane na podstawie danych podanych przez użytkownika w oknie dialogowym. Natomiast metody są wykorzystane do wyświetlenia pola i obwodu prostokąta.

```
<script>
  function Rectangle()
  {
    this.height=0;
    this.width=0;
    this.area=function()
    {
      let areaP=this.height*this.width;
      return areaP;
    },
    this.circuit=function()
    {
      let circuitP=2*this.height+2*this.width;
      return circuitP;
    }
  }
</script>
```

```
const rectangle1=new Rectangle();
rectangle1.height=parseFloat(prompt('Podaj 1 wysokość: '));
rectangle1.width=parseFloat(prompt('Podaj 1 szerokość: '));
console.log("Pierwszy prostokąt:");
console.log(`Pole prostokąta o bokach ${rectangle1.height} i ${rectangle1.width} wynosi ${rectangle1.area()}`);
console.log(`Obwód tego prostokąta ${rectangle1.circuit()}`);

const rectangle2=new Rectangle();
rectangle2.height=parseFloat(prompt('Podaj 2 wysokość: '));
rectangle2.width=parseFloat(prompt('Podaj 2 szerokość: '));
console.log("Drugi prostokąt:");
console.log(`Pole prostokąta o bokach ${rectangle2.height} i ${rectangle2.width} wynosi ${rectangle2.area()}`);
console.log(`Obwód tego prostokąta ${rectangle2.circuit()}`);
</script>
```

## Klasy w JavaScript

Klasy są jedną z funkcji wprowadzonych w wersji **ES6** JavaScript.

Klasa jest planem obiektu. Możesz stworzyć obiekt z klasy. Klasy ES6 to tylko funkcje specjalne.

Przed ES6 JavaScript nie miał koncepcji klas. Aby naśladować klasę, często używano powyższego wzorca konstruktor/prototyp. W ES6 pojawiły się klasy, które w dużej mierze są tzw. *syntactic sugar* dla konstruktora funkcji z paroma różnicami. Niemniej jest to konstrukcja bardziej nowoczesna i przyjazna dla oka.

Możesz myśleć o klasie jak o szkicu domu. Zawiera wszystkie szczegóły dotyczące podłóg, drzwi, okien itp. Na podstawie tych opisów budujesz dom. Dom jest obiektem.

Ponieważ z tego samego opisu można zrobić wiele domów, możemy stworzyć wiele obiektów z klasy.

### Tworzenie klasy JavaScript

Do konstrukcji klasy używamy słowa kluczowego **class**. Zaraz po nim podajemy nazwę klasy, także według standardów nazwy klas piszemy zawsze wielką literą. Po nazwie klasy od razu otwieramy klamery i tworzymy ciało klasy.

Pierwszym elementem jest **konstruktor**. Jest to metoda specjalna i może wystąpić tylko raz w całej klasie, albo wcale. Konstruktor uruchamia się zawsze na początku tworzenia klasy, jest to pierwsza wywołana metoda przez JavaScript i nie zależy to od nas.

W konstruktorze za pomocą **this** definiujemy pola w klasie. Konstruktor przyjmuje parametry, które są między innymi inicjalizacyjnymi wartościami dla pól klasy.



## Przykład 9. funkcja konstruktora i klasa

```
<script>
  // constructor function
  function Person (name1, age1 ) {
    this.name = name1,
    this.age = age1
  }
  // create an object
  const person1 = new Person("Jan", 32);
  console.log(person1);

  // creating a class
  class Person2 {
    constructor(name,age) {
      this.name = name;
      this.age = age;
    }
  }
  // create an object
  const person2 = new Person2("Gabriela", 18);
  console.log(person2);
</script>
```

**Uwaga:** `constructor()` – metoda wewnątrz klasy, jest wywoływana automatycznie za każdym razem, gdy tworzony jest obiekt.

## Metody klas JavaScript

Korzystając z funkcji konstruktora, definiujesz metody jako:

```
<script>
  // constructor function
  function Person (name) {
    this.name = name;
    // defining method
    this.greet = function () {
      return `Hello ${this.name}`;
    }
  }
  const person1 = new Person("Jan");
  console.log(person1);
  console.log(person1.greet());
</script>
```

**Zdefiniowanie metody w klasie JavaScript:** Po prostu podajesz nazwę metody, po której następuje (). Na przykład:

```
// creating a class
class Person2 {
  constructor(name) {
    this.name = name;
  }
  // defining method
  greet() {
    return `Hello ${this.name}`;
  }
}

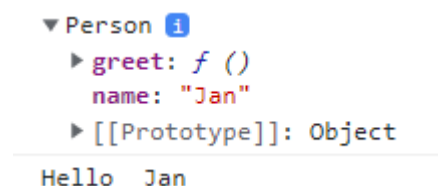
let person2 = new Person2('Anna');
console.log(person2); //zwróć uwagę na funkcję w konsoli
console.log(person2.name);
console.log(person2.greet());
</script>
```

## Różnice między klasą, a funkcją

Pomimo podobieństw między klasą a typem niestandardowym zdefiniowanym za pomocą funkcji konstruktora, istnieje kilka ważnych różnic.

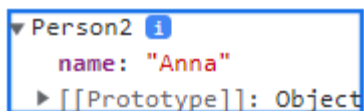
Istnieją pewne różnice między obiektami, które powstają z tych konstrukcji. Wystarczy wypisać te dwa obiekty do konsoli ( patrz powyższy przykład)

W przypadku funkcji konstruktora przez console.log możemy wypisać wszystkie właściwości obiektu, ponieważ są one wyliczalne, głównie chodzi o to, że **widzimy też metody tego obiektu**.



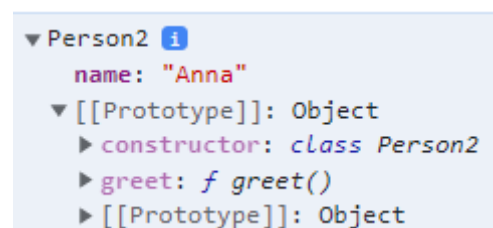
```
▼ Person ⓘ
  ▶ greet: f ()
    name: "Jan"
  ▶ [[Prototype]]: Object
Hello Jan
```

W przypadku klasy widzimy tylko i wyłącznie pole name, natomiast metoda greet i konstruktor klasy nie jest widoczny.



```
▼ Person2 ⓘ
  name: "Anna"
  ▶ [[Prototype]]: Object
```

Zobaczymy je dopiero po rozwinięciu prototypu



```
▼ Person2 ⓘ
  name: "Anna"
  ▼ [[Prototype]]: Object
    ▶ constructor: class Person2
    ▶ greet: f greet()
    ▶ [[Prototype]]: Object
```

**Metody zdefiniowane w klasie nie są wyliczalne.** To oznacza też, że gdy pobieramy klucze obiektu przez `Object.keys` albo wartości przez `Object.values` to metody klasy zostaną pominięte.

Zazwyczaj jest to wygodne, ponieważ to pola klasy przechowują ważne dla nas informacje i nie potrzebujemy iterować po metodach obiektu.

Dodatkowo mamy kilka innych różnic:

- deklaracja klas nie podlega pod hoisting tak jak deklaracja funkcji. Czyli **najpierw musimy zadeklarować klasę**, a potem możemy ją użyć. Jak pamiętamy hoisting przenosi deklarację funkcji na początek kodu, dlatego możliwe jest użycie funkcji przed jej deklaracją.
- kod w klasie jest uruchamiany w trybie ścisłym, czyli razem z poleceniem
- nie możemy wywołać klasy bez słowa `new`
- metod w klasie nie można wywoływać z konstruktorem, czyli ze słówkiem `new`

Poza tymi różnicami obiekty tworzone przez klasy są tymi samymi obiektami, które tworzymy literalnie albo przy pomocy konstruktora funkcji. Możemy na nich używać wszystkich metod pochodzących z `Object.prototype` czy też metod statycznych z `Object`.

Klasę zatem należy zdefiniować przed jej użyciem. W przeciwieństwie do funkcji i innych deklaracji JavaScript, klasa nie jest podnoszona. Dostęp do klasy przed jej zdefiniowaniem powoduje błąd np.:

```
// accessing class
const p = new Person(); // ReferenceError

// defining class
class Person {
  constructor(name) {
    this.name = name;
  }
}
```