**Szkolenie 10**

**JavaScript**

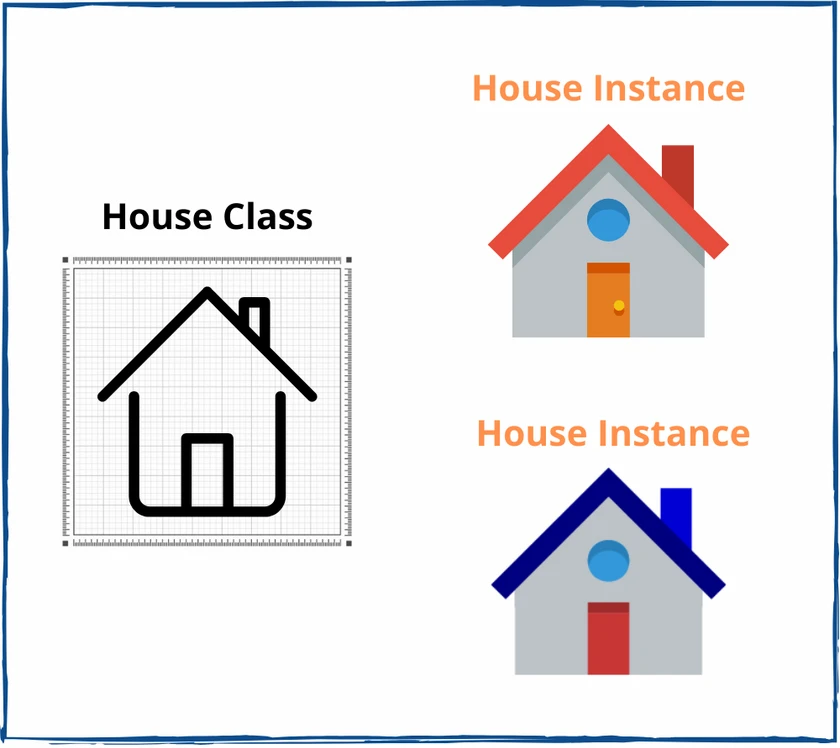
**Object oriented programming (OOP)**

**Wstęp**

Obiektowe podejście do pisania programów służy do tego, aby tworzyć jeszcze łatwiejszy do zrozumienia kod. Używane w takim podejściu klasy oraz obiekty mają przypominać twory występujące w rzeczywistości. Dzięki temu programiści mogą w o wiele łatwiejszy sposób budować wyobrażenie o tworzonym programie i jego logice. Jeżeli znasz inne języki programowania niż Java (np. Python) i rozumiesz ideę OOP, to szkolenie nie powinno Ci sprawić większego problemu. Kwestia tylko do zapoznania się z syntaxem kodu.

**Na czym polega to całe OOP?**

Sekcja ta jest przeznaczona głównie dla tych, którzy mają do czynienia z omawianym zagadnieniem po raz pierwszy. Poniżej omówię Ci pokrótce, jak powinieneś postrzegać programy pisane w podejściu obiektowym. W naszych programach będziemy przede wszystkim tworzyli **klasy**. Są to niejako schematy, na podstawie których będziemy instancjonowali (tworzyli) **obiekty**. Klasę możesz wyobrażać sobie jak plan wytwarzania danego komponentu (obiektu) w fabryce. Produkując bowiem kolejne obiekty, będziemy bazowali na jednym schemacie (klasie), który będzie definiował ich właściwości i zachowanie. Obiekty mogą odpowiadać rzeczom występującym w realnym świecie. Dla przykładu, program graficzny może zawierać obiekty takie jak koło, kwadrat, menu. System sklepu internetowego może zawierać obiekty typu koszyk, klient czy produkt.



Różnica między klasą a obiektem

Wyżej opisana cecha podejścia obiektowego dotyczy jednego z głównych założeń paradygmatu obiektowego: abstrakcji. Jak już się przekonałeś, mówi on o tym, że każdy obiekt w systemie służy jako model abstrakcyjnego „wykonawcy”.

Z OOP związane jest również zagadnienie dziedziczenia (które de facto ma swoje odzwierciedlenie również w otaczającym nas świecie). Proces ten polega na tym, iż możemy łatwo tworzyć “rodziny klas” i przekazywać cechy między poszczególnymi klasami (które będą pełniły rolę rodzica i dziecka). Dzięki temu nasz kod staje się bardziej przejrzysty, mniej w nim powtórzeń, a my nie musimy ręcznie dodawać te same cechy do różnych klas.

Następną cechą obiektowego podejścia do pisania programów jest enkapsulacja informacji. Cecha ta dotyczy chowania informacji wewnątrz klasy i umożliwienie ich odczytu spoza kontekstu klasy. Wówczas programista ma zapewnienie, że przez przypadek nie zmodyfikuje danej właściwości obiektu. Enkapsulację informacji można realizować przez tworzenie prywatnych pól klasy, o których więcej w dalszej części tego szkolenia.

Pozostała cechą programowania obiektowego jest również polimorfizm. Jednak z racji specyfiki JavaScript, pojęcia te nie stanowią trzonu pisanego kodu, dlatego pominiemy ich rozwinięcie. Z tymi zagadnieniami spotkasz się, gdy programując głównie w językach statycznie typowanymi, jak np. Java czy C++.

**Utworzenie klasy**

Klasę w JavaScript możemy utworzyć na dwa sposoby (przez deklarację lub wyrażenie przypisujące klasę do zmiennej). W zasadzie workflow jest taki sam, co w przypadku funkcji. Tam również mogliśmy albo stworzyć “wolnostojącą” funkcję przy użyciu słowa kluczowego function, bądź też przypisać ją do zmiennej, za pomocą której mieliśmy dostęp do jej ciała.

**Pierwsza klasa i konstruktor**

Poniżej pierwszy ze sposób utworzenia klasy Human wraz z konstruktorem:



| class Human {  constructor(name, surname){  this.name = name;  this.surname = surname;  } }  const krzys = new Human('Krzysztof', 'Nowak'); // utworzenie obiektu console.log(krzys.name); // Krzysztof krzys.name = "Krzyś"; // zmiana imienia console.log(krzys.name); // Krzyś |
| --- |

Konstruktor to specjalna metoda (zapamiętaj! metoda jest tym samym, co funkcja, z tą różnicą, że znajduje się w klasie), która jest wywoływana jako pierwsza podczas tworzenia obiektu danej klasy. To w nim definiujemy, jakie **właściwości** (nazywane też **polami** lub **atrybutami**) ma posiadać obiekt klasy. W powyższym przykładzie klasa Human ma dwa pola: name oraz surname. Nadanie im odpowiednich wartości jest możliwe dzięki umieszczeniu na liście parametrów konstruktora odpowiednich nazw zmiennych (constructor(**name, surname**)). Linia Human('Krzysztof', 'Nowak') przedstawia proces tworzenia obiektu. Tutaj tak naprawdę jest niejawnie wywoływany konstruktor. I to właśnie dlatego możliwe jest przekazanie wartości ‘Krzysztof’ oraz ‘Nowak’ podczas tworzenia obiektu.

Poniżej wtrącę alternatywny sposób na zrobienie tego samego i zbudowanie klasy (ale przy użyciu operacji przypisania do zmiennej):

| let Human = class {  constructor(name, surname){  this.name = name;  this.surname = surname;  } };  const krzys = new Human('Krzysztof', 'Nowak'); console.log(krzys.name); // Krzysztof krzys.name = "Krzyś"; console.log(krzys.name); // Krzyś |
| --- |

Zapytasz, który sposób lepszy? Kwestia własnych preferencji i upodobania. Ja z racji, iż programuję również dużo w innych językach, wolę korzystać z pierwszego sposobu, gdy bezpośrednio definiuję klasę i nie przypisuję jej do żadnej zmiennej.

**Czy konstruktor jest zawsze wymagany?**

Wracając jeszcze na chwilę do tworzenia konstruktora w klasie. Bądź świadomy tego, iż każda klasa zawiera konstruktor. W momencie, gdy nie zdefiniujesz go jawnie, to i tak zostanie on automatycznie wygenerowany przez interpreter - powstanie tzw. **konstruktor domyślny**.

**Pola klasy**

Zdążyłem już wspomnieć o tym, czym są pola wewnątrz klasy. Definiujemy je w konstruktorze, poprzedzając słowem kluczowym **this**. Pola przechowują informacje o obiekcie danej klasy, jak np. w przypadku klasy Human: wiek, imię, nazwisko.

Możemy wyróżnić następujące rodzaje pól:

* Pola niestatyczne (czyli takie, jakie już poznałeś z powyższych przykładów)
* Pola statyczne (to takie, które przyjmują taką samą postać dla każdego obiektu klasy; więcej o tym typie pól niebawem)

A także ze względu na widoczność pól w obrębie programu:

* Pola publiczne (do takich pól możemy się odnosić również poza klasą; wszystkie utworzone przez nas do tej pory pola były publiczne)
* Pola prywatne (to takie, które są dostępne tylko w obrębie klasy)

**Pola publiczne**

Spójrzmy jeszcze raz na stworzony wyżej snippet kodu.

| let Human = class {  constructor(name, surname){  this.name = name;  this.surname = surname;  } };  const krzys = new Human('Krzysztof', 'Nowak'); console.log(krzys.name); // Krzysztof krzys.name = "Krzyś"; console.log(krzys.name); // Krzyś |
| --- |

Klasa Human zawiera dwa **pola publiczne** - name oraz surname. To, że są one publiczne możemy poznać po możliwości odwoływania się do nich poza klasą. Takie odwołanie odbywa się w linii console.log(krzys.name). Zauważ, że do pola name odwołujemy się spoza klasy Human.

**Ładniejsza konwencja tworzenia pól?**

Standard JS przyjmuje również możliwość deklarowania pól wewnątrz klas. Ma to poprawić czytelność kodu oraz rozdzielenie procesu definiowania od deklarowania pól. Klasę Human moglibyśmy przeobrazić do następującej postaci:

| class Human {  name;   surname;  constructor(name, surname){  this.name = name;  this.surname = surname;  } }; |
| --- |

Przed konstruktorem umieściłem deklaracje pól name oraz surname. Taka modyfikacja nie wpływa w ogóle na działanie programu, ale w mojej opinii poprawia jakość i czytelność kodu.

Ponadto, wykorzystując taki zapis, możemy określać wartości pól przed samym konstruktorem.

| class Human {  name;   surname = "Snow";  constructor(name){  this.name = name;  } };  const jon = new Human("Jon"); console.log(jon.name, jon.surname); // Jon Snow |
| --- |

**Pola prywatne**

Po omówieniu, czym są pola publiczne, możemy śmiało przejść do zapoznania się ze sposobem na tworzenie prywatnych atrybutów klas. Pojęcie to jest mocno zazębione z ideą enkapsulacji informacji w klasie (o której wspomnieliśmy już w sekcji “Czym jest to całe programowanie obiektowe”).

Aby utworzyć prywatny atrybut w klasie, wystarczy jako umieścić znak **#** jako prefix danego pola. Zmodyfikujmy poznany przykład kodu, tak aby nie mieć dostępu do żadnego pola klasy poza jej kontekstem. Spójrz poniżej:

| class Human {  #name;   #surname = "Snow";  constructor(name){  this.#name = name;  } };  jon = new Human("Jon"); console.log(jon.#name, jon.#surname); // Uncaught SyntaxError: Private field '#name' must be declared in an enclosing class |
| --- |

Rezultat wprowadzenia pól prywatnych, możemy zauważyć w metodzie console.log. Od teraz nie wyświetla ona właściwych wartości spod pól klasy - a to dlatego, że pola prywatne nie są widoczne poza klasą. Dlatego próba wydrukowania wartośći spod name oraz surname kończy się rzuceniem błędu SyntaxError.

W celu umożliwienia dostępu do danych pól prywatnych, możemy utworzyć odpowiednie metody, które będą zapewniały dostęp do takich pól składowych. Jak tworzyć metody, poznasz w następnej sekcji.

**Publiczne pola statyczne**

W JavaScript możemy wyróżnić jeszcze jeden typ pól, który oznacza się słowem kluczowym static. Jest to jednak o wiele rzadziej wykorzystywany typ pola od omówionych wyżej poprzedników, także skupimy się na nim na końcu. W tym momencie przejdźmy do zapoznania się z metodami.

**Definiowanie metod**

Klasa, w której znajdowałyby się tylko i wyłącznie pola, byłaby niezbyt użytecznym tworem w programie. Dlatego poniżej przedstawię Ci, w jaki sposób dodać do klasy Human przykładowe metody. Przypomnę Ci tylko, że metoda jest tym samym, co funkcja, jednak nosi ona nieco odmienną nazwę z racji, iż znajduje się w klasie.

| let Human = class {  constructor(name, surname, age){  this.name = name;  this.surname = surname;  this.age = age;  }    introduce(){  console.log(`Hi! I am ${this.name} ${this.surname}. I am ${this.age} years old.`);   }   get\_older(diff){  this.age += diff  } };  const krzys = new Human('Krzysztof', 'Nowak', 30); krzys.introduce(); // Hi! I am Krzysztof Nowak. I am 30 years old. krzys.get\_older(5); krzys.introduce(); // Hi! I am Krzysztof Nowak. I am 35 years old. |
| --- |

Zwróć uwagę, w jaki sposób odwołujemy się do danych pól z poziomu metod. Musimy przed odwołaniem, wykorzystać słowo kluczowe **this**. Występuje ono, np. w metodzie introduce():

| console.log(`Hi! I am ${**this.**name} ${**this.**surname}. I am ${**this.**age} years old.`); |
| --- |

Jak widzisz, sposób tworzenia metod w klasie jest prosty. Definicje metod umieszczamy pod konstruktorem, a sygnatura metody wygląda następująco:

| nazwa(*parametry*){  // ciało  } |
| --- |

**Metody w połączeniu z polami prywatnymi. Gettery i settery.**

Metody służą nam do dostarczania odpowiedniej funkcjonalności w klasie. Mogą też posłużyć do zapewnienia dostępu do pól prywatnych. Załóżmy, że operujesz na atrybucie prywatnym - name. Chcąc odczytywać i ustawiać dla niego wartość poza kontekstem klasy, możesz stworzyć “metody pośredniczące”, które będą zwracały i odpowiednio ustawiały jego wartość. Spójrz na poniższy przykład:

| class Human {  #name;   #surname = "Snow";  constructor(name){  this.#name = name;  }  getName(){  return this.#name;  }  getSurname(){  return this.#surname;  }  setSurname(surname){  this.#surname = surname;  } };  const jon = new Human("Jon"); console.log(jon.getName(), jon.getSurname()); // Jon Snow jon.setSurname("Jones"); console.log(jon.getName(), jon.getSurname()); // Jon Jones |
| --- |

Wspomnianymi metodami są getName, getSurname oraz setSurname. Zauważ, że umożliwiają one bezproblemowe operowanie na zmiennych prywatnych. Wykorzystanie metod takiego rodzaju stało się w kodzie na tyle popularne, że nazywamy je jako: **gettery** i **settery**. Oczywiście miej na uwadze to, że metody również możemy tworzyć jako prywatne i czasami będzie to miało swoje uzasadnienie (np. gdy standardowo chcemy uniemożliwić programistom wywoływanie pewnej funkcjonalność “z zewnątrz”). Wtedy zgodnie z poznaną nomenklaturą, przed nazwą metody umieszczamy znak **#**.

Gdybyśmy więc przez przypadek, nazwę getName zamienili na #getName, program kończyłby się błędem, ukazując problem z widocznością tej metody.

| class Human {  #name;   #surname = "Snow";  constructor(name){  this.#name = name;  }  #getName(){ // (\*) tutaj nastąpiła zmiana  return this.#name;  }  getSurname(){  return this.#surname;  }  setSurname(surname){  this.#surname = surname;  } };  jon = new Human("Jon"); console.log(jon.#getName(), jon.getSurname()); // Uncaught SyntaxError: Private field '#getName' must be declared in an enclosing class |
| --- |

**Ładniejszy sposób na stworzenie getterów i setterów**

Tak jak Ci już wspomniałem - proces tworzenia getterów i setterów dla pól prywatnych jest niezwykle często wykorzystywanym rozwiązaniem w programach. Dlatego komitet standaryzacyjny JS zdecydował w pewnym momencie, aby wprowadzić charakterystyczną konwencję tworzenia tego typu metod. Sposób wprowadza możliwość umieszczenia słowa get/set przed metodą, która jest określonym accessorem dla pola. Sam accessor powinien natomiast nazywać się tak samo, jak pole, do którego zapewnia dostęp. Poniżej zrefaktorowany przykład:

| class Human {  #name;   #surname = "Snow";  constructor(name){  this.#name = name;  }  get name(){  return this.#name;  }  get surname(){  return this.#surname;  }  set surname(surname){  this.#surname = surname;  } };  const jon = new Human("Jon"); console.log(jon.name, jon.surname); // Jon Snow jon.surname = "Jones"; // wywołanie settera console.log(jon.name, jon.surname); // Jon Jones |
| --- |

Zauważ, że dzięki wprowadzeniu takiego rozwiązania, nie musimy jawnie wywoływać metod name() oraz surname(). Interpreter JavaScript rozumie je, jakby były normalnymi polami. I tak na zapis, np. jon.name niejawnie wywoływany jest getter, a w przypadku jon.surname = “Jones”, następuje odwołanie się do metody setującej.

Zauważ, jak bardzo taki zapis usprawnia i porządkuje nasz kod. Mamy pewność, że gettery i settery będą nazywane i tworzone według określonej nomenklatury.

**Metody i pola statyczne**

Po tym, jak poznałeś, w jaki sposób tworzyć pola i metody w klasie, jesteś w pełni gotowy, aby dowiedzieć się, czym są **statyczne** twory w klasie.

Najpierw odpowiedzmy sobie na pytanie, co znaczy, że coś jest statyczne. Możemy tak określać pola, tudzież metody, które do swojego istnienia nie potrzebują obiektu klasy ich przechowujących.

**Pola statyczne**

Dlatego też pola statyczne często wykorzystuje się, aby definiować w klasie stałe lub informacje specyficzne dla klasy. Aby utworzyć takie pole, przed jego nazwą umieszcza się słowo static, np. static serverName = “locahost”.

Sprawdźmy, jak mogłoby wyglądać wykorzystanie takich pól w praktyce.

| class User {  static type = 'admin';  static system = 'Windows';   constructor(name){  this.name = name;  } };  console.log(`We are working on ${User.system}`);  console.log(`Creating new ${User.type} user...`); const new\_user = new User('devs'); |
| --- |

Zauważ, że, tak jak wspomniałem, aby wykorzystać dowolne pole statyczne klasy, wcale nie potrzebujemy jej istniejącego obiektu. Do pól statycznych odwołujemy się, bezpośrednio podając nazwę przechowującej je klasy. Statyczne pola w klasie umożliwiają więc nam porządkowanie kodu i tworzenie zmiennych, które jasno będą przywiązane do danej klasy.

**Metody statyczne**

Co do zasady, sposób odwoływania się do metod statycznych ma się tak samo, jak manipulowanie polami statycznymi. Tak więc, aby wywołać metodę statyczną, wcale nie potrzebujemy obiektu danej klasy.

Metodę statyczną tworzymy bardzo podobnie jak pola statyczne. Wystarczy przed jej nazwą umieścić słowo static, następnie normalnie podajemy jej nazwę wraz z listą ewentualnych parametrów. Metody statyczne zawsze umieszczane są w klasie.

| class Example{  static static\_method(){  console.log("I am static!");  } }; |
| --- |

Pracując jednak z takim rodzajem metod, musimy pamiętać o dwóch rzeczach:

| **1.** Metody statyczne mogą odwoływać się tylko i wyłącznie do pól statycznych  **2.** Metody statyczne nie mogą odwoływać się do pól i metod niestatycznych (brak w nich dostępu do this) |
| --- |

I znowu - metody statyczne tworzymy, aby zapewnić generyczny dostęp do danych funkcjonalności. Metody statyczne mogą, np. służyć do generowania obiektów, odczytywania nazw plików z systemu itd. Jednym słowem wszędzie tam, gdzie nie jest wymagany dostęp do pól składowych obiektu.

Zanim przedstawię Ci przykład wykorzystania metody statycznej, chciałbym zaprezentować Ci jeszcze, co implikują wyżej przedstawione wyjątki (brak dostępu do pól i metod niestatycznych z poziomu metod statycznych).

| class User{  static type = 'admin';  #nickname;   constructor(nickname){  this.nickname = nickname;  };   static example(){  console.log(User.type); // admin  console.log(this.#nickname); // Uncaught SyntaxError: Invalid or unexpected token   } };  User.example(); // wywołanie metody statycznej |
| --- |

Zauważ, że próba odwołania się do pola niestatycznego z poziomu metody statycznej kończy się błędem. Natomiast bez problemu możemy odwołać się do pola statycznego.

**Przykład funkcjonalności**

Załóżmy, że mamy następującą klasę:

Mamy klasę Employee, która przechowuje pola name, id oraz experience (pole informujące, ile lat zatrudniony jest już pracownik). Atrybuty te są określane na podstawie przesłanych parametrów: name, id, experience. Tak jak poniżej:

| class Employee {  #id;  #name;  #experience;    constructor(id, name, experience){  this.id =id;  this.name = name;  this.experience = experience;  }  display(){  console.log(`Hi! I am ${this.name} with ID: ${this.id}. I've been working here for ${this.experience} years.`)  } };  const e = new Employee(1, 'Jan', 3); e.display(); |
| --- |

Problem, który mógłby się teraz pojawić przy takim tworzeniu klas jest następujący: A co, jeżeli chcemy stworzyć obiekt klasy z odpowiednimi wartościami pól, nie mając do dyspozycji pola experience, a jedynie datę rozpoczęcia pracy? Oczywiście moglibyśmy samodzielnie wyliczyć, ile już lat pracuje pracownik, wykonując odejmowanie “w głowie” (czyli, gdy rozpoczął pracę, np. w 2020 roku, to od bieżącego roku odjąć tę wartość), jednak byłoby to mało zautomatyzowane.

Z pomocą przychodzi metoda statyczna, która będzie niejako drugim konstruktorem klasy i będzie tworzyła obiekt o odpowiednich polach.

| class Employee {  #id;  #name;  #experience;    constructor(id, name, experience){  this.id = id;  this.name = name;  this.experience = experience;  }   static create\_from\_year(id, name, year\_joined){  return new Employee(id, name, new Date().getFullYear() - year\_joined);  }   display(){  console.log(`Hi! I am ${this.name} with ID: ${this.id}. I've been working here for ${this.experience} years.`)  } };  const jan = new Employee(1, 'Jan', 3); const mateusz = Employee.create\_from\_year(2, 'Mateusz', 2019) jan.display(); mateusz.display(); |
| --- |

W rezultacie program wyświetli dwa komunikaty:

| Hi! I am Jan with ID: 1. I've been working here for 3 years.  Hi! I am Mateusz with ID: 2. I've been working here for 2 years |
| --- |

Zwróć szczególną uwagę na to, w jaki sposób stworzyliśmy drugi obiekt. Dzięki wprowadzeniu metody statycznej, możliwe jest produkowanie obiektów klasy, mając do dyspozycji jedynie specyficzne informacje (np. rok rozpoczęcia pracy zamiast doświadczenia wyrażonego w latach) - program i tak automatycznie obliczy potrzebne dane.

**Zakończenie**

Na tym zakończymy szkolenie dotyczące podstaw programowania obiektowego. Przed nami zapoznanie się z jeszcze jednym - niezwykle ważnym zagadnieniem w programowaniu - dziedziczeniem.