**Szkolenie 11**

**JavaScript**

**Dziedziczenie i prototypy**

**Wstęp**

Wiesz już, jak budować pełnoprawne klasy i tworzyć na ich podstawie obiekty. Czasami jednak zdarzają się sytuacje, kiedy musisz stworzyć wiele klas, które zawierają te same (lub podobne) pola oraz metody. Chyba nie muszę Ci już przypominać, że w takiej sytuacji, każdy wprawiony programista powinien stosować regułę DRY (Don’t Repeat Yourself) i starać się, aby tylko jednorazowo stworzyć powielające się linie kodu i w miarę możliwości przekazywać je między klasami, które składałyby się z tych własności. Jak już się zapewne domyślasz taki proces “przekazywania cech” nazywamy dziedziczeniem.

**Jakie problemy rozwiązuje dziedziczenie?**

Aby dobrze Ci nakreślić problem, dla którego idealnym rozwiązaniem będzie wprowadzenie dziedziczenia, spójrz na poniższy kod (klasy reprezentują postacie z gry RPG):

| class Elf {  name;  hp = 100;  gold = 50;  abilities;   constructor(name){  this.name = name;  this.abilities = new Map([['shooting', 'just shot a bow'], ['attack', 'just attacked enemy with knife']]);  }    use\_ability(ability){  if(this.abilities.has(ability)){  console.log(`${this.abilities.get(ability)}`);  }  else{  console.log("Unknown ability!");  }  }    tell(){  console.log(`Hello! I am ${this.name}.`);  } };  class Blacksmith {  name; // (\*)  hp = 100; // (\*)  gold = 30; // (\*)  abilities; // (\*)  inventory = [];   constructor(name){  this.name = name;  this.inventory.push("hammer");  this.abilities = new Map([['improving items', 'Improving... An item has been improved.']])  }   use\_ability(ability){ // (\*)  if(this.abilities.has(ability)){  console.log(`${this.name} ${this.abilities.get(ability)}`);  }  else{  console.log("Unknown ability!");  }  }    tell(){ // (\*)  console.log(`Hello! I am ${this.name}. I can repair your stuff`);  }   };  const legolas = new Elf('Legolas'); const jack = new Blacksmith('Jack');  legolas.tell(); // Hello! I am Legolas. legolas.use\_ability('shooting'); // Legolas just shot a bow  jack.tell(); // Hello! I am Jack. I can repair your stuff. jack.use\_ability('improving items'); // Improving... An item has been improved. jack.use\_ability('shooting'); // Unknown ability! |
| --- |

Powyższy kod nie wprowadza żadnego nowego zagadnienia. W programie utworzyliśmy dwa obiekty - legolas oraz jack, na których pokazałem przykładowe wykorzystanie metod z klas.

Zwróć jednak uwagę, że do niektórych linii kodu, dodałem w formie komentarza, zapis: **(\*)**. W ten sposób oznaczyłem linie, które są bardzo podobne między klasami Elf oraz Blacksmith. Co więcej, właśnie przez takie powielenie kodu, nasz program staje się mało estetyczny i łamie podstawowe zasady czystego i efektywnego kodu. Jak rozwiązać ten problem? Zastosować dziedziczenie!

Dziedziczenie (klasa przekazująca cechy to rodzic, a je otrzymująca to dziecko) umożliwia przekazanie określonych pól oraz metod między poszczególnymi klasami. W ten sposób możemy efektywnie powielać różne cechy klasy bez konieczności ich wielokrotnego zapisywania. Proces tworzenia odpowiedniej hierarchii klas wygląda następująco:

* Tworzymy klasę bazową (rodzica) - będzie ona zawierała te wszystkie cechy, które będziemy chcieli przekazywać jej potomkom. W naszym przykładzie będą to wszystkie linie, które oznaczyłem komentarzem (\*)
* Następnie budujemy klasy pochodne (potomków), które będą dziedziczyły uprzednio stworzonej bazie i nadawały odpowiednie wartości dla swoich (również tych odziedziczonych) pól.

Spójrz więc, jak zrefaktorujemy kod z poprzedniego przykładu - tak, aby stał się on maksymalnie czysty i bazował na koncepcji dziedziczenia.

**Wprowadzenie konceptu dziedziczenia**

| class Character {  name;   hp = 100;   gold;  abilities;     constructor(name, gold, abilities){  this.name = name;  this.gold = gold;  this.abilities = abilities;  }    use\_ability(ability){  if(this.abilities.has(ability)){  console.log(`${this.abilities.get(ability)}`);  }  else{  console.log("Unknown ability!");  }  }    tell(){  console.log(`Hello! I am ${this.name}.`);  }  };  class Elf extends Character {  constructor(name, gold, abilities){  super(name, gold, abilities);  } };  class Blacksmith extends Character {  inventory;    constructor(name, gold, abilities, inventory){  super(name, gold, abilities);  this.name = name;  this.abilities = new Map([['improving items', 'Improving... An item has been improved.']]);  this.inventory = inventory;  }    tell(){   super.tell();  console.log(`I can repair your stuff`);  }   };  const legolas = new Elf('Legolas', 50, new Map([['shooting', 'just shot a bow'], ['attack', 'just attacked enemy with knife']])); const jack = new Blacksmith('Jack', 30, new Map([['improving items', 'Improving... An item has been improved.']]));  legolas.tell(); // Hello! I am Legolas. legolas.use\_ability('shooting'); // Legolas just shot a bow  jack.tell(); // Hello! I am Jack. I can repair your stuff. jack.use\_ability('improving items'); // Improving... An item has been improved. jack.use\_ability('shooting'); // Unknown ability! |
| --- |

**Omówienie**

Zacznijmy od zagłębienia się w stworzoną klasę bazową.

Nazwaliśmy ją Character (uważam, że nazwa ta jest bardzo trafna, ponieważ w ten sposób ujednolica nazwy klas potomnych - Elf oraz Blacksmith i pokazuje, że odnosi się do obu klas jednocześnie). Zgodnie z ideą dziedziczenia, która polega na tym, że wszystkie publiczne składowe klasy bazowej są przenoszone do klas potomnych, w klasie Character umieściliśmy pola: **name**, **hp**, **gold**, **abilities** oraz metody: **tell**, **use\_ability**. Wszystkie te cechy przekazywane są do klas potomnych - Elf oraz Blacksmith.

**Jak zapisywać dziedziczenie?**

Zwróć szczególną uwagę na to, jak określiliśmy sposób dziedziczenia po danej klasie. Aby to zrealizować, przy nazwie odpowiednich klas użyliśmy słowa kluczowego **extends** (class Blacksmith extends Character; class Elf extends Character). To właśnie ono jawnie mówi nam, z której klasy ma dziecko “zaciągać” cechy. Tak więc, gdybyśmy chcieli zapisać szablon definiowania, co ma dziedziczyć po czym, to stworzymy:

| **class** nazwaKlasyPotomnej **extends** klasaBazowa |
| --- |

Ostatnim krokiem, którym musimy się zająć, aby przeprowadzić proces dziedziczenia, to **zmodyfikowanie konstruktora** klasy dziedziczącej. Otóż zasada jest prosta - za każdym razem, gdy chcemy utworzyć klasę potomną, musimy w pierwszej kolejności wywołać konstruktor jej rodzica. Dlatego też realizujemy to przez jawne umieszczenie polecenia **super()** na początku konstruktora klasy. Super() odnosi się do klasy bazowej i powoduje wywołanie jej konstruktora. Jest to dość logicznie uzasadniony krok, o którym nie możemy zapomnieć. A to dlatego, że przecież żadne dziecko nie może istnieć bez narodzin swojego rodzica. Stąd też wzięła się konieczność wywoływania super z poziomu kodu.

Dodatkowo, super() może między swoimi nawiasami przyjmować argumenty. Będzie to uzależnione od tego, czy klasa bazowa posiada jakieś parametry w swoim konstruktorze. W naszym przykładzie, super wywoływaliśmy następująco: super(name, gold, abilities). A więc przekazywaliśmy do niego argumenty name, gold, abilities. W ten sposób mówiliśmy, że klasa bazowa ma powstać z polami o konkretnych wartościach i przekazywać już tak utworzone atrybuty do swoich potomków.

**Efekt dziedziczenia?**

Gdybyś teraz uruchomił tak powstały kod, zauważysz, że działa on identycznie do tego, który stworzyliśmy uprzednio - bez wykorzystania zasad SOLID oraz Clean Code. Skróciliśmy długość klas, nasz kod stał się bardziej przejrzysty i czytelny. Same pozytywy!

Wszystkie publiczne metody oraz pola z klasy Character zostały przekazane dalej do klas potomnych - Elf oraz Blacksmith. Zauważ przecież, że np. w tej linii:

jack.use\_ability('shooting'); // Unknown ability!

wywołujemy metodę, która nie jest bezpośrednio umieszczona w klasie obiektu legolas (Elf). Zdefiniowaliśmy ją w klasie bazowej i została sprytnie przekazana do swojego dziecka.

Zauważ również, że z poziomu klas potomnych, możemy rozszerzać działanie metod z klasy bazowej (proces ten nazywa się przeciążeniem metod, z ang. overriding). Przykładem takiego przeciążenia jest metoda tell(). W klasie bazowej ma ona następującą postać:

| tell(){  console.log(`Hello! I am ${this.name}.`); } |
| --- |

Jej definicja jednak na tym się nie kończy. Zwróć uwagę, że w klasie Blacksmith przeciążyliśmy ją tak jak poniżej:

| tell(){   super.tell();  console.log(`I can repair your stuff`); } |
| --- |

Zrobiliśmy tak, ponieważ dla Blacksmitha, nie chcieliśmy wykorzystywać standardowego działania metody tell. Naszym celem było zapewnienie również wyświetlania komunikatu “I can repair your stuff” dla każdego obiektu Blacksmith przy równoczesnym zachowaniu komunikatu z klasy bazowej: `Hello! I am ${this.name}.`.

Dlatego też w rozpatrywanej klasie umieściliśmy definicję metody tell(), która w pierwszej kolejności wywołuje tę z klasy bazowej (dzięki zapisowi super.tell(); przypominam, że super to inaczej odwołanie do rodzica). Pod takim wywołaniem natomiast umieszczamy już funkcjonalność metody właściwą dla klasy, w której się znajduje.

**Wielokrotne dziedziczenie?**

Jeżeli programujesz również w innych językach, jak np. Python, to możesz zastanawiać się, jak wprowadzić w JS proces multidziedziczenia (wówczas rodzicami jest kilka klas bazowych). Domyślnie w JavaScript proces ten jest niewykonalny. Klasy mogą maksymalnie dziedziczyć po jednym rodzicu. Oczywiście są alternatywne sposoby na wprowadzenie wielodziedziczenia, ale to jest na tyle specyficzny temat, że jego omówienie zostawimy sobie, gdy faktycznie sytuacja będzie wymagała od nas jego wdrożenia.

**Dynamiczne przypisywanie pól**

JavaScript jest językiem dynamicznym. Nie wspomniałem o tym do tej pory, ale bądź świadomy tego, że możliwe jest przypisanie pól do obiektów danej klasy poza jej definicją.

| class Human{  #name;  constructor(name){  this.#name = name;  }    get name() {  return this.#name;  } };   const stefan = new Human("Stefan"); const jan = new Human("Jan"); console.log(stefan.name); console.log(jan.name);  stefan.age = 15; // dynamiczne przypisanie nowego pola console.log(stefan.age); # 15 console.log(jan.age); // undefined, bo age tylko w obiekcie stefan |
| --- |

Proces dynamicznego przypisania pola przeprowadziliśmy na obiekcie stefan. Odwołaliśmy się do niezdefiniowanego pola age oraz nadaliśmy mu konkretną wartość (15). W ten sposób obiekt dynamicznie zyskał nowy atrybut (co możemy ujrzeć w linii console.log(stefan.age)), która wyświetla 15.

Zwróć jednak uwagę, że pole to zostało przypisane tylko i wyłącznie w obrębie obiektu stefan. Próba odwołania się do pola age spod innego obiektu (w naszym przykładu jan) kończy się zwróceniem typu undefined.

Istnieje jednak ciekawe rozwiązanie umożliwiające m.in dynamiczne przypisywanie pól nie do obiektów, ale całych klas. A to wszystko dzięki **prototypom**.

**Prototypy**

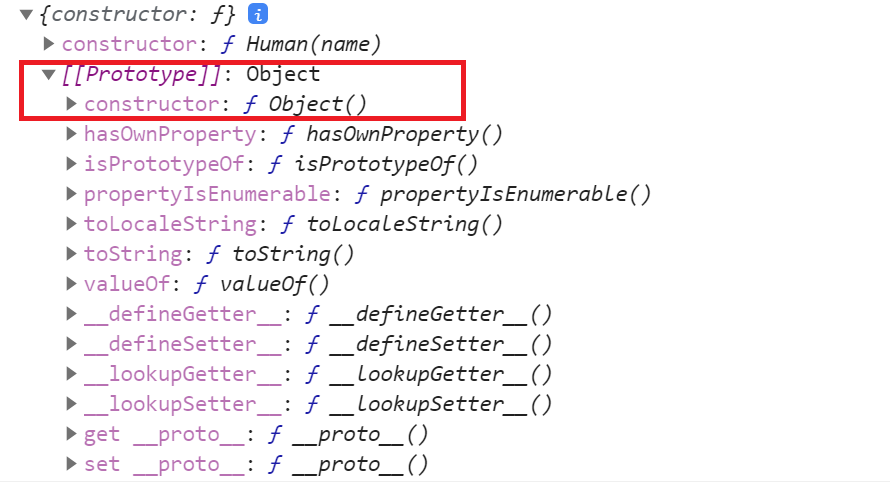
Prototyp to specjalny obiekt, który jest dodawany do każdej z funkcji. Do takiego obiektu możemy dodawać dowolne metody i pola, również w sposób dynamiczny.

Spójrz na poniższy przykład, który dobrze pokazuje, jak tworzyć obiekt w niecodzienny sposób:

| function Human(name) {  this.name = name };  console.log(Human.prototype); |
| --- |

W powyższym przykładzie tworzymy tak naprawdę funkcję i przypisujemy do niej (właściwie do jej obiektu prototypu) dowolne pola.

Umieszczony w kodzie console.log(Human.prototype) tylko ugruntowuje nas w przekonaniu, że niejawnie z funkcją związany jest nowy obiekt (prototyp).



Wynik operacji console.log

**Dynamiczne pola**

Tak jak już wspomniałem, mając do dyspozycji taki prototyp, możemy dynamicznie definiować nowe pola, niekoniecznie w obrębie definicji funkcji. Dodajmy więc dynamicznie do prototypu nowe pole - age.

Wykorzystamy zapis taki jak poniżej:

| Human.prototype.age = 5; |
| --- |

W rezultacie nasz kod możemy zapisać następująco:

| function Human(name) {  this.name = name };  const stefan = new Human("Stefan"); const jan = new Human("Jan");  Human.prototype.age = 25;  console.log(stefan.age); // 25 console.log(jan.age); // 25 |
| --- |

**Definiowanie metod w prototypie**

Chcąc teraz dodać do naszej funkcji prototypowej Human, metodę, np. showName(), stworzymy kod taki jak poniżej:

| Human.prototype.showInfo = function(){  console.log(`Hello! I'm ${this.name} and I'm ${this.age} years old.`); }; |
| --- |

Dzięki powyższemu zapisowi, JS dodaje nową metodę do obiektu prototypu.

| function Human(name) {  this.name = name };  Human.prototype.showInfo = function(){  console.log(`Hello! I'm ${this.name} and I'm ${this.age} years old.`); };  const stefan = new Human("Stefan"); const jan = new Human("Jan");  Human.prototype.age = 25;  stefan.showInfo(); // Hello! I'm Stefan and I'm 25 years old.  jan.showInfo(); // Hello! I'm Jan and I'm 25 years old. |
| --- |

**Dodatkowe korzyści płynące z prototype**

Powyżej przedstawiłem Ci, jak korzystać z prototypów. Poznałeś również zaletę ich wykorzystania - możliwe jest dynamiczne przypisywanie metod oraz pól do każdego obiektu będącego instancją prototypu. Bądź świadomy, że to ma duże znaczenie przy optymalizacji kodu. Dobrym przykładem jest biblioteka jQuery.

Dowolny obiekt jQuery posiada wiele metod, które często są takie same i korzystają z tych samych zasobów. Wówczas ich duplikowanie byłoby marnotrawieniem pamięci. Twórcy biblioteki rozwiązali to tak, że wszystkie te metody są zdefiniowane na prototypie, a każdy z obiektów dziedziczy taki prototyp i uzyskuje dostęp do wszystkich potrzebnych metod przez niewielki nakład pamięci i zasobów.

**Podsumowanie**

Chciałbym ceremonialnie ogłosić, że tym szkoleniem kończymy serię podstawowych szkoleń z JS. Dzięki tak gruntownej i solidnej wiedzy zawartej w ostatnich materiałach, możemy rozwijać się dalej jako FrontEnd Developerzy. Przed nami zaawansowane zagadnienia z JavaScript oraz takie frameworki jak React, Redux.