WPROWADZENIE DO OOP W JAVASCRIPT

Techniki programowania obiektowego

JS

Różne techniki programowania

Naukę programowania (przynajmniej w JS) najczęściej zaczyna się od programowanie proceduralnego, gdzie funkcje operują na danych, a obie te kategorie (stany i działanie) są niezależne. *To robiliśmy do tej pory.*

Kolejnym krokiem, który podnosi nas na kolejny poziom, jest programowanie obiektowe.

Programowanie obiektowe (OOP)

Sposób (technika, paradygmat) programowania, w którym program zbudowany jest z obiektów. Program jest zbiorem obiektów (połączone dane i metody) oraz relacji i zależności między nimi.

Czy JavaScript jest obiektowy?

JS jest językiem obiektowym (podobnie jak m.in. Java, C++, C#, Python), który umożliwia tworzenie programu za pomocą paradygmatu obiektowego JavaScript pozwala też tworzyć programy proceduralnie lub za pomocą programowania funkcyjnego.

Dlaczego o JS możemy powiedzieć, że jest językiem obiektowym? (same obiekty to za mało)

JS - Obiektowy język programowania

Obiekty, wzorce obiektów (konstruktory/klasy) hermetyzacja, dziedziczenie, polimorfizm, abstrakcja - elementy niezbędne, by język uznać za obiektowy.

Wszystko to ma JS, choć pewne koncepcje realizuje inaczej niż inne obiektowe języki programowania.

OOP - składnia

m.in. obiekt (właściwości i metody), konstruktor, klasa, prototyp, instancja, this

Oczywiście składnia jest niezależna od OOP (i można z niej korzystać nawet nie tworząc programowania w oparciu o paradygmat obiektowy), ale znajomość składni jest niezbędna do programowania obiektowego.

JS

OOP - Teoria i praktyka

Przed nami prezentacja, więc przede wszystkim teoria, której niektóre fragmenty (dotyczące zarówno technik programowania jak i składni) mogą być trudne. Wrócimy jednak do tego w kodzie i będzie wtedy łatwiej :)

OOP - myślenie

MYŚLENIE OBIEKTOWE - umiejętność tworzenie programów za pomocą obiektów.

Zmiana sposobu myślenia względem programowania proceduralnego, gdzie dane i zachowania występują osobno. I to jest naprawdę coś innego, nie tylko jeśli chodzi o składnię, ale przede wszystkim jeśli chodzi o sposób projektowania.

JS

Modelowanie (myślenie w OOP)

MODELOWANIE - kluczowy aspekt myślenia w paradygmacie obiektowym.

Model - (uproszczona) rzeczywistość zbudowana za pomocą obiektów (określamy to często abstrakcją). Dla tworzenia modelu kluczowa jest perspektywa programisty, który musi uchwycić całość rzeczywistości, którą opisuje. Dobry programista, w tym kontekście, to ktoś kto potrafi modelować (projektować obiektowo) program.

Modelowanie (myślenie w OOP)

Warto pamiętać, że nie chodzi tu tylko o tworzenie obiektów o określonych cechach i celach, ale także o tworzenie relacji między obiektami (zależności, powiązania).

Cechy obiektu - właściwości i metody potrzebne do wykonania zadania

Relacja między obiektami - np. zależność, agregacja, wbudowanie

Zadanie obiektu - obiekt tworzymy po to, by realizował określone zadanie

JS

OBIEKT - podstawowy budulec

DANE (Właściwości)

ZACHOWANIE (Metody)

Obiekt kontroluje też dostęp do składowych.

Technicznie obiekt to nieuporządkowany (kolejność nie ma znaczenia) zbiór właściwości, które składają się z pary klucz (nazwa właściwości) - wartość (którą może być dowolny typ). Jeśli wartością jest funkcja, to taką właściwość nazywamy metodą.

JS

OOP - Wprowadzenie do składni.

Lepiej zrozumiesz ją, kiedy przejdziemy do edytora i zaczniemy pisać kod.

Tworzenie obiektu - literał obiektu

```
const user = {
        //właściwość
        name: "Janek",   //para klucz-wartość
        age: 28,
        showName: function() {
                console.log(`Cześć ${this.name}!`)
        } //jeśli wartością jest funkcja, to mówimy o metodzie
}

user.name // "Janek" (odczytanie - get)
user.age = 29;// (przypisanie nowej wartości - set)
user.gender = "male"; // (stworzenie i przypisanie)
-------------------------------------------------
//Obiekty są dynamiczne (zmiany, dodawanie, usuwanie)
```

Tworzenie obiektu - literał obiektu
alternatywny sposób tworzenia metody

```
const user = {
        name: "Janek",
        age: 28,
        showName: function() {
                console.log(`Cześć ${this.name}!`)
                //this ma przypisywane znaczenie w chwili wywołania metody
        },
        //alternatywny (od ES6) sposób tworzenia metod w obiekcie
        showAge() {
                console.log(`Wiek użytkownika ${this.name},to ${this.age} `);
        }
}
```

Tworzenie obiektu - new + konstruktor Object

```
const cat = new Object()

//po utworzeniu dodajemy właściwości

cat.name = "fafik";
cat.meow = function(){
     console.log(this.name + ": miau miau!")
}
```

KONSTRUKTOR (wzór dla obiektu)

Konstruktor to funkcja, której używamy jako wzór do tworzenia nowego obiektu (nowej instancji). Konwencja wymaga napisania nazwy konstruktora wielką literą.

Konstruktory (funkcje) wbudowane to np.: Array, Object, Date, Function, String, Number.

W programowaniu obiektowym nie obędziemy się bez pisania własnych konstruktorów czy to w postaci funkcji, czy za pomocą składni klas, która pojawiła się w ES6.

JS

Tworzenie KONSTRUKTORA (wzór)

```
const Animal = function(name, species) {
      this.name = name;
      this.species = species;
      this.eat = function() {
        console.log(this.name + ' auuuu (jakie dobre)');
      }
}
```

JS

Tworzenie KONSTRUKTORA (wzór)

```
function Animal(name, species) {
      this.name = name;
      this.species = species;
      this.eat = function() {
        console.log(this.name + ' auuuu (jakie dobre)');
      }
}
```

Tworzenie KONSTRUKTORA (wzór)

```
const Animal = function(name, species) {
        this.name = name;
        this.species = species;
        this.eat = function() {
                console.log(this.name + ' auuuu (jakie dobre)');
        }
}

UTWORZENIE OBIEKTU (INSTANCJI)
const dog = new Animal("azor", "owczarek");  //Instancja obiektu
const dog2 = new Animal("muszka", "nikt nie wie"); //Instancja obiektu

dog.name // "azor" - odczytanie właściwości
dog2.name = "muszeczka"  //przypisanie nowej wartości do właściwości
dog2.eat() // "muszeczka auuuu (jakie dobre)" - wywołanie metody
```

JS

Tworzenie INSTANCJI - proces

```
const Animal = function(name, species) {
        this.name = name;
        this.species = species;
}
const dog = new Animal("azor", "owczarek");
```

PROCES TWORZENIA INSTANCJI

1. operator new, który tworzy w połączeniu z konstruktorem nowy (pusty) obiekt
2. this od tego momentu wskazuje na ten obiekt (następuje wiązanie this z nowym obiektem)
3. *Nowy obiekt zostaje połączony z prototypem funkcji konstruktora.*
4. nowy obiekt jest zwracany (i przypisywany, referencja do niego, w zmiennej)

JS

czym jest INSTANCJA i co robi KONSTRUKTOR

```
//Konstruktor (wzorzec, instrukcja)
const Animal = function(name, species) {
        this.name = name;
        this.species = species;
}
```

```
//Tworzenie instancji
const dog = new Animal("azor", "owczarek");
```

- Instancja to obiekt stworzony zgodnie z wzorcem w konstruktorze.
- Powstający obiekt jest niezależnym bytem, który posiada własne właściwości i metody nadane mu przez obiekt wzorcowy (konstruktor) oraz ma dostęp do metod i właściwości będących w posiadaniu takiego obiektu wzorcowego (poprzez odwołanie się do prototypu konstruktora).
- W wielu obiektowych językach programowania mamy taką strukturę jak klasa, na podstawie której tworzone są obiekty. W JavaScript taka struktura wzorcowa (który jest funkcją konstruktorem, ale też od ES6 ma postać klasy) ma charakter zbioru instrukcji, które są wykonywane na nowo tworzonym obiekcie.

JS

Prototyp - bez użycia

```
const Animal = function(name) {
        this.name = name;
        this.children = [];
        this.addChildren = function(childName) {
                this.children.push(childName)
        }
}

const hamster = new Animal('bobik');;
hamster.addChildren("romuś");
```

Prototyp - składnia

```
const Animal = function(name) {
        this.name = name;
        this.children = [];
}

Animal.prototype.addChildren = function(childName) {
        this.children.push(childName)
}

const hamster = new Animal('bobik');;
hamster.addChildren("romuś");
```

Prototyp - składnia

```
const Animal = function(name) {
        this.name = name;
        this.children = [];
}

const hamster = new Animal('bobik');
const canary = new Animal('spiewak');

Animal.prototype.addChildren = function(childName) {
        this.children.push(childName)
}
Animal.prototype.age = 2;

hamster.addChildren("romuś");

canary.age // 2
hamster.age // 2
```

Prototyp - co to jest?

PROTOTYP - SPECJALNY OBIEKT W FUNKCJI KONSTRUKTORA (W KLASIE), który przechowuje wspólne metody i właściwości dla wszystkich instancji.

Obiekt (instancja) może mieć własne właściwości i metody, a może je też dziedziczyć (bez przypisania).

Dziedziczenie w JS oparte jest przede wszystkim na prototypach.

JS

Konstruktor, prototyp, instancja

Trochę tego jest, zrozumiemy to tak naprawdę dopiero w kodzie.

Przejdźmy teraz do innego sposobu (od ES6). Do tworzenia instancji użyjemy klasy.

KLASY w JavaScript (class)

KLASA (ES6) - Klasy w JS pojawiły się w 2015 roku. Ale nie jest to nowy mechanizm. Pod nową składnią kryją się te same konstruktory.

Klasy są rozwiązaniem z wielu innych języków programowania. Ich wprowadzenie czyni JS zbliżonym (składniowo) do innych języków obiektowych. Kod staje się też (w mojej opinii) czystszy, łatwiejszy w pisaniu i przejrzysty, dlatego współcześnie większość programistów JS korzysta z klas zamiast z konstruktora. Przy czym zapamiętajmy, że mechanizm jest ten sam. Klasy to tylko syntax sugar (cukier/lukier składniowy - robi to samo, ale "ładniej").

Tworzenie KLASY (wzór)

```
class Animal {
      constructor(name, species) {
            this.name = name;
            this.species = species;
            this.eat = function() {
                  console.log(this.name + ' mniam mniam)');
            }
      }
}
```

KLASA a KONSTRUKTOR

```
class Animal {
   constructor(name, species) {
      this.name = name;
      this.species = species;
      this.eat = function() {
         console.log(this.name
        + ' mniam mniam)');
       }
   }
}
```

```
function Animal (name, species) {
      this.name = name;
      this.species = species;
      this.eat = function() {
         console.log(this.name
        + ' mniam mniam)');
          }
   }
```

JS

Tworzenie KLASY (wzór)

```
class Animal {
	constructor(name, species) {
		this.name = name;
		this.species = species;
	}

	//do prototypu klasy jest dodawana taka funkcja
	eat(){
		console.log(this.name + ' auuuu (jakie dobre)');
	}
}
```

KLASA a KONSTRUKTOR (z prototypem)

```
class Animal {
        constructor(name, species) {
                this.name = name;
                this.species = species;
        }
        eat(){}
}
```

```
function Animal (name, species) {
        this.name = name;
        this.species = species;
}

Animal.prototype.eat = function() {};
```

KLASA a KONSTRUKTOR (z prototypem)

```
class Animal {
  constructor(name, species) {
        this.name = name;
        this.species = species;
  }
  eat(){
        console.log('jedz!' + this.name)
  }
}
```

```
function Animal (name, species) {
        this.name = name;
        this.species = species;
}

Animal.prototype.eat = function() {
        console.log('jedz!' + this.name)
};
```

Zwróć uwagę, że zawartość klasy to zbiór metod

Tworzenie INSTANCJI

```
const dog = new Animal("muszka", "nikt nie wie");
//Składnia dokładnie taka jak przy konstruktorze
i dokładnie tak samo działający mechanizm.

typeof Animal //"function" - klasa jest funkcją
```

Nie stworzymy obiektu bez konstruktora (klasy)

const obj = { } //literał wykorzystuje konstruktor Object
const arr = [ ] //literał wykorzystuje konstruktor Array

```
//Funkcja konstrukcyjna (konstruktor)
const Animal = function(){}

//Instancja
const dog = new Animal()
```

```
//Klasa (pełni rolę konstruktora)
class User{}

//Instancja
const marek = new User()
```

Będą przykłady

Składnia klas to jest podstawa pracy z OOP, ale znajomość mechanizmów (funkcja, konstruktor i prototyp), też jest niezbędna.

Kontynuujemy temat składni – zobaczmy, czym jest this.

this - wiązanie z obiektem

this to mechanizm w JavaScript, który niejeden raz bardzo Cię zaskoczy :)

Dzięki this nasz kod jest bardziej uniwersalny.

"Weź to, co masz w ręku i podrzuć" jest bardziej uniwersalne niż
"Weź jabłko, które masz w ręku i podrzuć",
"Weź kluczyki, które masz w ręku i podrzuć" itd.

this - przykład

```
btn.addEventListener("click", function() {
      this.classList.toggle("on");
})
```

this będzie odnosiło się do przycisku, czyli do obiektu, na którym wykonuje się funkcja.

Zwróć uwagę, że ta sama funkcja może być wykonana na wielu elementach, jest uniwersalna, ponieważ wiązanie this z obiektem następuje w chwili wywołania funkcji a nie w chwili tworzenia funkcji (i to trzeba zapamiętać!)

JS

this i arrow function

```
btn.addEventListener("click", () => {
     this.classList.toggle("on");
})
```

this będzie odnosiło się do obiektu globalnego, bo funkcja strzałkowa nie tworzy własnego wiązania this, tylko go przejmuje z wyższego zakresu (więc gdy addEventListener jest w zakresie globalnym, to z zakresu globalnego).

this

```
const car = {
      name: 'polonez',
      year: 1999,
      age() {
            console.log(`Wiek samochodu to ${2019 - this.year}
            lat`);
      }
}
car.age()  //"Wiek samochodu to 20 lat"
```

this będzie odnosiło się do obiektu car, ale pamiętaj, że this (wiązanie, określenie obiektu) nastąpi w chwili wywołania metody, czyli tu car.age()

JS

OOP - wprowadzenie do zasad

This będzi Ci towarzyszył w Twojej podróży z JS...

Teraz jednak zostawmy już składnię i przejdźmy do zasad OOP. Tu też potem (w kodzie) będą przykłady dla lepszego zrozumienia.

HERMETYZACJA (ENCAPSULATION)

Grupowanie (zamykanie) metod i właściwości w obiekcie.

Integralność danych i ukrywanie danych. Kontrolowaniem dostępu do danych ma zajmować się sam obiekt.

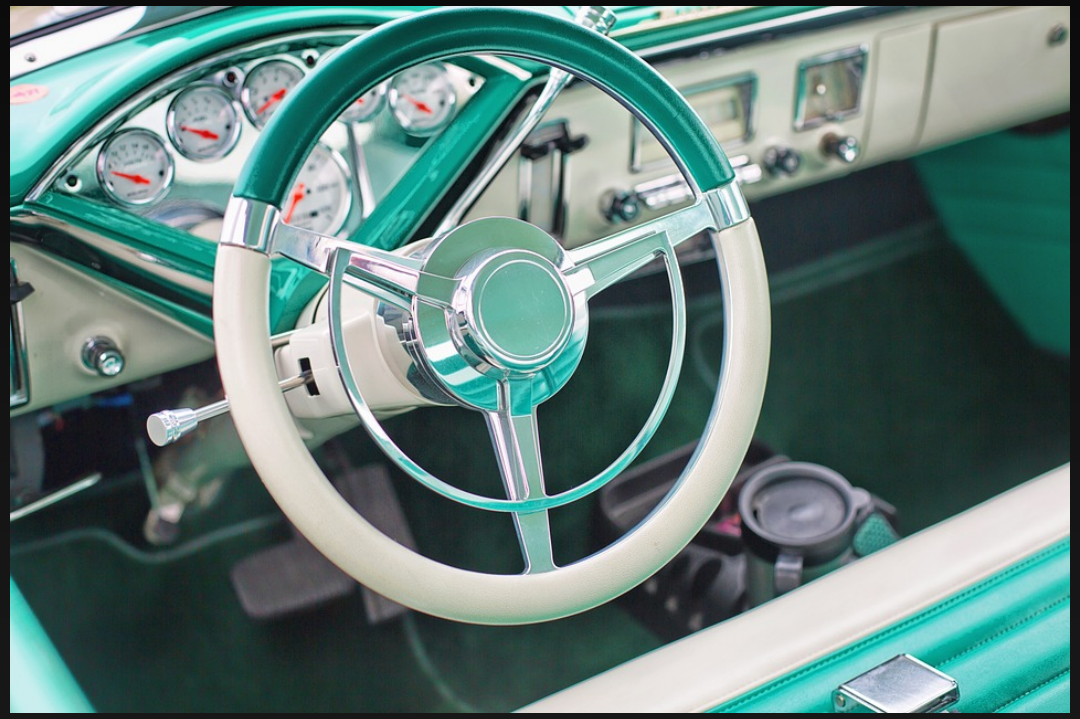
Popularnym rozwiązaniem jest używanie metod pobierających (getter) i dostępowych (setter) do danych w obiekcie.

Implementacja staje się ukryta a dostęp do danych w jednym obiekcie jest udostępniany innemu obiektowi za pomocą interfejsu.

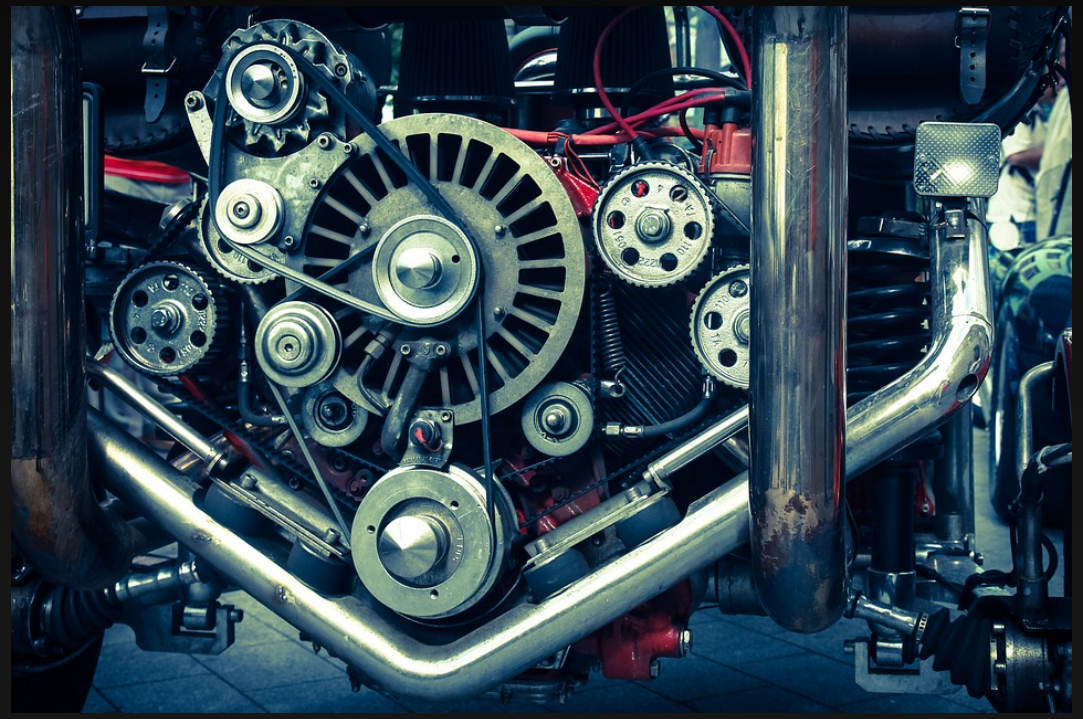
JS

HERMETYZACJA W OBIEKCIE

INTERFEJS



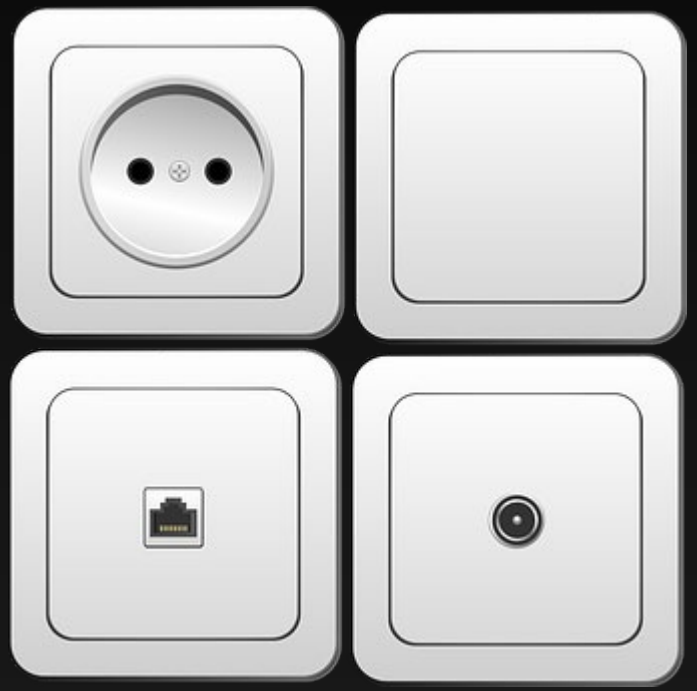
IMPLEMENTACJA



JS

HERMETYZACJA

INTERFEJS



IMPLEMENTACJA

JS

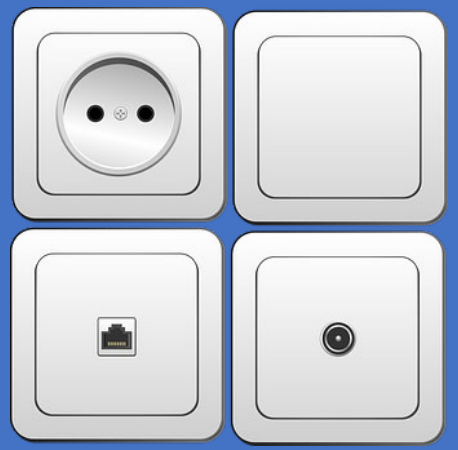
HERMETYZACJA

Obiekt 1

Obiekt 2

Obiekt 3

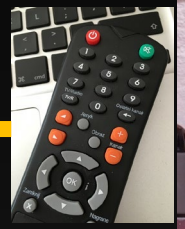
INTERFEJS



IMPLEMENTACJA

JS

HERMETYZACJA



INTERFEJS

DAJ MI SYGNAŁ

INTERFEJS

DAJ MI OBRAZ I WIDEO

ZŁY POMYSŁ :)

DAJ MI PRĄD

DAJ MI PRĄD

INTERFEJS

DAJ MI PRĄD

INTERFEJS

JS

HERMETYZACJA

```
Math.floor(2.22)
```

//Implementacja nas nie obchodzi i nie mamy do niej dostępu. Interfejsem dla obiektu math jest metoda obiektu, w tym wypadku floor.

Przykład braku ukrycia danych (minimalna hermetyzacja związana z przestrzenią danych, bez ukrycia danych).

```
const user = {
 age: 20
}
user.age = 21; //zmiana
```

HERMETYZACJA

Jak ukryć pewne dane w JS? Wrócimy w kodzie, by zobaczyć przykład.

DZIEDZICZENIE (INHERITANCE)

Dzięki dziedziczeniu redukujemy ilość powtarzanego kodu.

Stworzenie relacji między obiektami i grupowanie obiektów.

Zmniejszenie ilości potencjalnych błędów

DZIEDZICZENIE W JS

1. Instancje mają dostęp do wspólnych metod i właściwości konstruktora (klasy) oraz jego konstruktora itd. (na szczycie jest prototyp konstruktora Object). Takie dziedziczenie odbywa się poprzez prototype chain (łańcuch prototypów)
2. Jedne klasy mogą przyjąć właściwości innych klas (i je rozszerzać). W klasie w JavaScript dzieje się to za pomocą eleganckiego słowa extends.

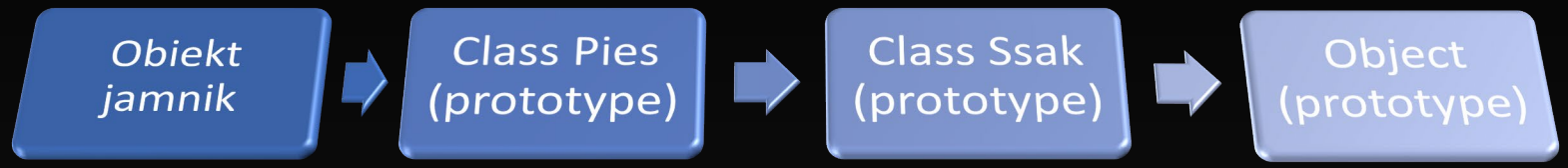
```
class Dog extends Animal { }

    //docelowo to, co ma klasa Animal plus właściwości i metody dodane w klasie Dog

    // Dog jest podklasą (klasą potomną, subclass) klasy Animal (która jest klasą
    nadrzędną, superclass)
```

Prototyp

Schemat



jamnik.hauhau()

jamnik.toString()

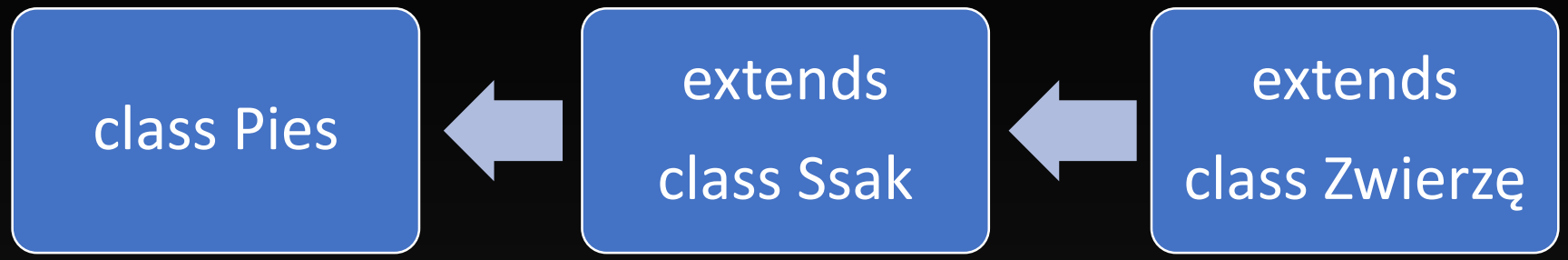
jamnik.ogon

jamnik.wiek

JS

Extends

Schemat



szczekanie

kończyny

stałocieplność

płuca

włosy

mózg

oddychanie

jedzenie

rozmnażanie

stałocieplność

płuca

włosy

mózg

oddychanie

jedzenie

rozmnażanie

mózg

oddychanie

jedzenie

rozmnażanie

imie:

JS

DZIEDZICZENIE

Do prototypów i extends wrócimy w kodzie

POLIMORFIZM

Obiekt zachowuje się inaczej w zależności od dostarczonych danych. Polimorficzny w podstawowym znaczeniu to różne postacie tej samej rzeczy, zmiana kształtu.

Przejawy polimorfizmu:

Implementacja metod, które zachowują się inaczej w zależności od tego, jakie/ile argumentów otrzymają.

Różna implementacja tej samej metody w różnych obiektach.

POLIMORFIZM - przykłady JS

1. Mechanizm przeciążenia w innych językach programowania, ale w JS też można uzyskać ten sam efekt, choć w inny sposób (jedna metoda obsługuje różne dane, lub/i różną liczbę)

2. Wykorzystanie łańcucha dziedziczenia (prototyp). Metoda jednego obiektu o tej samej nazwie przysłania metodą innego obiektu (nadrzędnego).

POLIMORFIZM

Do przysłaniania wrócimy kodzie

ABSTRAKCJA (ABSTRACTION)

Model rzeczywistości, który upraszcza złożoność i pozwala przedstawić problem (zadanie) za pomocą obiektów i relacji miedzy nimi.

Abstrakcja określa, jakie cechy musi posiadać i jakie zadanie realizować obiekt.

Abstrakcja to umiejętność pozwalająca na modelowanie programu/projektu.

JS

ABSTRAKCJA (ABSTRACTION)

Realizujący określone przez projekt zadania, uproszczony model rzeczywistości, oparty o obiekty. Obiekty te mają określoną budowę, interfejs i implementacje, cel, procesy, oraz co bardzo ważne posiadające relację z innymi obiektami.

PRZYKŁADOWE RELACJE MIĘDZY OBIEKTAMI

Kompozycja - obiekt zawierają inne obiekty. Kompozycja zakłada, że jeden jest wbudowany w inny obiekt. Przy czym obiekt wbudowany jest zależny od obiektu, w którym się znajduje, a jego istnienie poza obiektem, w który jest wbudowany, nie ma sensu.

Asocjacja - każdy obiekt tej relacji istnieje niezależnie. Pomimo to istnieje relacja (powiązanie) między obiektami. Istnienie jednego obiektu nie jest potrzebne do istnienia drugiego.

Agregacja - obiekt składa się z innych obiektów (agreguje je). Sens istnienia agregatu (obiektu głównego) polega na posiadaniu obiektów, które przechowuje (choć te obiekty mogą istnieć poza nim).

JS

OOP w praktyce

1. Omówienie składni i mechanizmów związanych z OOP (w tej prezentacji zarysowaliśmy jedynie tematykę)
2. Projekt OOP

Jeśli czegoś nie rozumiesz z tej prezentacji, to wróć do niej po skończeniu tego rozdziału.

A teraz przejdźmy do kodu...