dr inż. Michał Malinowski
michal.malinowski@warszawa.merito.pl

https://www.linkedin.com/in/michał-malinowski-malkon/

JavaScript: Funkcje, Obiekty i Prototypy

Wykład 02

JĘZYKI PROGRAMOWANIA
STUDIUM PRZYPADKU

Zagadnienia (2h)

- Zadanie: Wykonanie założeń dla projektu
- Funkcje
- Obiekty
- Klasy
- Przykłady Użycia Obiektów

Biały Dom pilnie domaga się, aby programiści używali języków programowania bezpiecznych dla pamięci.

Luty 2024

Biuro Krajowego Dyrektora ds. Cyberbezpieczeństwa (ONCD)
w Białym Domu

Zadanie: Wykonanie założeń dla projektu

Wykonanie założeń dla projektu

Opracowanie dokumentu zawierającego informacje o projekcie

- temat
- opis
- funkcjonalności (co najmniej 8)
- środowisko programistyczne bazujące na ECMAScript2015

Przedstawienie wyniku

- Dokument tekstowy PDF
- Umieszczenie wyniku w Moodle

Termin

- Podany przy zadaniu w Moodle (około 2 tygodnie)

Bezpieczne języki programowania

Automatyczne zarządzanie pamięcią

Eliminuje błędy związane z ręcznym zarządzaniem pamięcią, takie jak wycieki pamięci i podwójne zwalnianie pamięci, co zapobiega wielu potencjalnym problemom bezpieczeństwa.

Bezpieczne typowanie

Pomaga wykrywać błędy na wczesnym etapie (podczas kompilacji), zapobiegając nieprawidłowym operacjom na danych, które mogłyby prowadzić do błędów

Sprawdzanie granic

Dla tablic i innych struktur danych zapobiega przepełnieniom bufora, które mogą prowadzić do nadpisania danych i potencjalnych luk w bezpieczeństwie.

Zapobieganie wstrzyknięciu kodu

Uniemożliwia wstrzyknięcie i wykonanie nieautoryzowanego kodu, na przykład przez odpowiednie sanitowanie danych wejściowych i stosowanie bezpiecznych API.

Model własności i zarządzanie dostępem

Stosowanie modeli własności i pożyczania, które na poziomie kompilacji zapewniają, że dostęp do danych jest zarządzany w sposób bezpieczny, eliminując wyścigi danych i inne błędy związane z konkurencyjnym dostępem.

Immutability (niemutowalność)*

Wymuszanie stosowania niemutowalnych struktur danych pomaga w zapobieganiu błędom wynikającym ze zmian stanu, co jest szczególnie ważne w aplikacjach współbieżnych i rozproszonych.

*Immutability, czyli niemutowalność, to właściwość obiektów (zmiennych) w programowaniu, która nie pozwala na zmianę ich stanu po utworzeniu

Bezpieczne języki programowania

Bezpieczne języki	Pośrednie języki	Niebezpieczne języki
Rust	JavaScript	C
Swift	TypeScript	C++
Go	Kotlin	Assembly
Java	Scala	
C#	Python	
Haskell	Ruby	
Elixir	PHP	

Podstawowe operatory w języku JavaScript

Operator	Opis	Przykład Użycia	Wynik Przykładu
+	Dodawanie	5 + 3	8
-	Odejmowanie	10 - 5	5
*	Mnożenie	4 * 7	28
/	Dzielenie	20 / 4	5
%	Reszta z dzielenia (modulo)	7 % 2	1
++	Inkrementacja (zwiększenie o 1)	let x = 5; x++;	x staje się 6
--	Dekrementacja (zmniejszenie o 1)	let y = 10; y--;	y staje się 9
==	Równość (bez sprawdzania typu)	5 == '5'	true
===	Równość (ze sprawdzaniem typu)	5 === '5'	false
!=	Nierówność (bez sprawdzania typu)	5 != '5'	false
!==	Nierówność (ze sprawdzaniem typu)	5 !== '5'	true
>	Większe niż	5 > 3	true
<	Mniejsze niż	5 < 10	true
>=	Większe lub równe	5 >= 5	true
<=	Mniejsze lub równe	5 <= 10	true
&&	Operator logiczny AND	true && false	false
||	Operator logiczny OR	true || false	true
!	Operator logiczny NOT	!true	false
? :	Operator warunkowy (ternary)	5 > 10 ? 'tak' : 'nie'	'nie'

Funkcje

Funkcje w programowaniu

to zorganizowane bloki instrukcji, które wykonują określone zadania.

Służą one do strukturyzowania kodu, umożliwiają jego wielokrotne wykorzystanie i ułatwiają zarządzanie złożonymi programami.

Można je wywołać, używając ich nazwy, aby uruchomić zawarty w nich kod, który może opcjonalnie zwracać wynik operacji.

Deklaracja funkcji

```
function nazwaFunkcji(parametr) {
    const wynik = parametr * parametr;
    return wynik;
}
```

Wywołanie funkcji

```
nazwaFunkcji(2); // Zwraca 4
nazwaFunkcji(3); // Zwraca 9
nazwaFunkcji(5); // Zwraca 25
```

Funkcje

Deklaracja funkcji wewnątrz pliku

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Proste Funkcje w JavaScript</title>
</head>
<body>
  <h1>Wyniki Działań</h1>
  <script>
    function dodaj(a, b) { return a + b; }
    function mnoz(a, b) { return a * b; }
    function odejmij(a, b) { return a - b; }
    console.log('Dodaj 2 + 3:', dodaj(2, 3));
    console.log('Mnóż 4 * 5:', mnoz(4, 5));
    console.log('Odejmij 10 - 6:', odejmij(10, 6));
  </script>
</body>
</html>
```

Funkcje

Deklaracja funkcji w pliku zewnętrznym „bibliotece”

index.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Przykład Użycia Prostej Biblioteki</title>
</head>
<body>
  <h1>Wyniki Działań</h1>
  <script src="simpleLib.js"></script>
  <script>
    // Użycie funkcji z biblioteki simpleLib.js
    console.log('Dodaj 2 + 3:', dodaj(2, 3)); // 5
    console.log('Mnóż 4 * 5:', mnoz(4, 5)); // 20
    console.log('Odejmij 10 - 6:', odejmij(10, 6)); // 4
  </script>
</body>
</html>
```

simpleLib.js

```javascript
// Funkcja do dodawania dwóch liczb
function dodaj(a, b) {
  return a + b;
}

// Funkcja do mnożenia dwóch liczb
function mnoz(a, b) {
  return a * b;
}

// Funkcja do odejmowania dwóch liczb
function odejmij(a, b) {
  return a - b;
}
```

Funkcje

Parametry (argumenty) funkcji

W ES6 i nowszych wersjach JavaScript można określić domyślne wartości dla parametrów funkcji. Jeśli funkcja zostanie wywołana bez argumentu dla danego parametru, zostanie użyta wartość domyślna:

```
function powitaj(imie = "Gościu") {
  console.log("Witaj, " + imie + "!");
}
```

W ES6 wprowadzono także rest parameters, które pozwalają funkcji przyjąć nieokreśloną liczbę argumentów jako tablicę:

```
function suma(...numery) {
  return numery.reduce((a, b) => a + b, 0);
}
```

Obiekt arguments, który jest dostępny wewnątrz funkcji. Pozwala on na dostęp do wszystkich argumentów przekazanych do funkcji, niezależnie od tego, czy zostały one zdefiniowane w deklaracji funkcji, czy nie.

```
function przykladowaFunkcja() {
  for(let i = 0; i < arguments.length; i++) {
    console.log(arguments[i]);
  }
}
```

Funkcje

Różnica między arguments a rest parameters (...)

Cecha	arguments	Rest Parameters (...)
Typ	Obiekt podobny do tablicy	Prawdziwa tablica
Dostęp do wartości	Tak, poprzez indeksowanie arguments[i]	Tak, jako tablica z argumentami funkcji
Metody tablicowe	Nie, arguments nie jest instancją Array i nie posiada wbudowanych metod tablicowych	Tak, obsługuje wszystkie metody tablicowe (map, filter, reduce, itd.)
Definicja	Nie wymaga specjalnej składni, automatycznie dostępny w funkcjach	Definiowany jawnie w deklaracji funkcji, np. function przyklad(...args)
Dostępność	Dostępny we wszystkich typach funkcji, oprócz funkcji strzałkowych	Dostępny we wszystkich typach funkcji, włączając funkcje strzałkowe
ES6+	Dostępny, ale użycie rest parameters jest zalecane dla lepszej czytelności i funkcjonalności	Wprowadzony w ES6, oferuje nowoczesne i elastyczne podejście do obsługi argumentów

Funkcje

Zwracane wartości

Funkcje w JavaScript zwracają wartości przy użyciu słowa kluczowego return.
Instrukcji return może być wiele dla jednej funkcji.

Metoda zwracania	Przykład	Opis
Zwracanie prostej wartości	function dodaj(a, b) { return a + b; }	Funkcja zwraca wynik operacji arytmetycznej.
Zwracanie wartości zmiennej	function obliczKwadrat(liczba) { const wynik = liczba * liczba; return wynik; }	Funkcja oblicza kwadrat liczby, przypisuje go do zmiennej i zwraca tę zmienną.
Zwracanie obiektów	function stworzOsobe(imie, wiek) { return { imie: imie, wiek: wiek }; }	Funkcja tworzy i zwraca obiekt z właściwościami imie i wiek.
Zwracanie funkcji	function zewnetrzna() { return function wewnetrzna() { console.log("Witaj Świecie!"); }; }	Funkcja zwraca inną funkcję jako wartość.
Zwracanie tablic	function zwrocTablice() { return [1, 2, 3, 4, 5]; }	Funkcja tworzy i zwraca tablicę z pięcioma elementami.
Zwracanie wartości z warunkami	function porownaj(a, b) { if (a > b) { return "a jest większe"; } else if (a < b) { return "b jest większe"; } else { return "a i b są równe"; } }	Funkcja zwraca różne łańcuchy znaków w zależności od spełnionego warunku.
Zwracanie undefined	function nicNieRob() { }	Funkcja nie ma zadeklarowanego return, więc domyślnie zwraca undefined.

Funkcje

Wyrażenie funkcyjne

to sposób tworzenia funkcji przez przypisanie ich do zmiennych. Są one anonimowe, co znaczy, że zazwyczaj nie posiadają nazwy i są traktowane jako wartości, które:

- mogą być przypisane do zmiennych,
- przekazywane jako argumenty do innych funkcji,
- zwracane przez funkcje.

W przeciwieństwie do deklaracji funkcji, wyrażenia funkcyjne nie są hostowane, co oznacza, że nie można ich wywołać przed ich definicją w kodzie.

```
const nazwaFunkcji = function(parametr) {
  // ciało funkcji
};
```

Wyrażenie funkcyjne można wywołać, używając nazwy zmiennej, do której została przypisana. Ponieważ są one traktowane jak zwykłe zmienne, funkcje zdefiniowane jako wyrażenia funkcyjne przy użyciu const lub let nie są dodawane do globalnego obiektu window w kontekście przeglądarki internetowej.

Funkcje

Deklaracja funkcji jest dostępna w całym zakresie kodu, w którym została zadeklarowana, niezależnie od miejsca w skrypcie, dzięki procesowi hostingu. JavaScript "przenosi" taką deklarację na początek zakresu, co pozwala na odwołanie się do funkcji przed jej fizyczną deklaracją w kodzie.

```
// Możemy wywołać funkcję przed jej deklaracją
powitanie();

// Deklaracja funkcji
function powitanie() {
  console.log("Witaj świecie!");
}
```

Wyrażenie funkcyjne, w przeciwieństwie do deklaracji, nie jest hoistowane. Oznacza to, że odwołanie się do funkcji zdefiniowanej jako wyrażenie funkcyjne przed jej zadeklarowaniem spowoduje błąd.

```
// To odwołanie spowoduje błąd,
powitanie();

// Wyrażenie funkcyjne
const powitanie = function() {
  console.log("Witaj świecie!");
};
```

Funkcje

Funkcje zadeklarowane jako **deklaracja funkcji** stają się kluczami obiektu globalnego (**window** w przeglądarkach), gdy są zadeklarowane w zakresie globalnym. To samo dotyczy zmiennych zadeklarowanych przy użyciu **var**.

```
// Funkcja dostępna jako metoda obiektu window
function powitanieGlobalne() {
  console.log("Witaj globalnie!");
}

window.powitanieGlobalne(); // Działa
```

Natomiast funkcje zadeklarowane jako **wyrażenie funkcyjne** przy użyciu **const** lub **let** nie są dodawane do obiektu globalnego.

```
// Funkcja nie jest dodawana do obiektu window
const powitanieLokalne = function() {
  console.log("Witaj lokalnie!");
}

window.powitanieLokalne; // undefined
```

Funkcje

Funkcje anonimowe

to funkcje, które nie posiadają nazwy. Są często wykorzystywane w sytuacjach:

- gdzie funkcja jest używana jako wartość – na przykład jako argument przekazywany do innej funkcji,
- przy przypisaniu funkcji do zmiennej.

Bez nazwy, taka funkcja jest wywoływana za pomocą zmiennej, do której została przypisana, lub bezpośrednio, jeśli jest przekazana jako argument.

Funkcje anonimowe są szczególnie przydatne w przypadku callbacków* i zdarzeń.

```
const przyklad = function() {
  console.log("Jestem funkcją anonimową.");
};
```

*callback to funkcja, którą przekazuje się do innej funkcji jako argument, aby mogła zostać wywołana

```
setTimeout(function() {
  console.log("Jestem funkcją anonimową wywołaną z setTimeout.");
}, 2000);
```

Funkcje

Funkcje strzałkowe "arrow functions"

to skrócona forma wyrażenia funkcji, która zapewnia syntaktyczne uproszczenie i rozwiązuje niektóre trudności związane ze słowem kluczowym this. Są one często używane jako anonimowe funkcje, szczególnie w przypadkach, gdy funkcja jest krótka lub gdy jest używana jako callback

```
const funkcjaStrzalkowa = (parametry) => {
  // ciało funkcji
};
```

W JavaScript this jest słowem kluczowym odnoszącym się do kontekstu obiektu, w którym jest używane. Odnosi się do obiektu, który wywołał daną funkcję. Jego wartość jest ustawiana w zależności od tego, jak i gdzie funkcja jest wywoływana. Na przykład:

- W metodzie obiektu this odnosi się do obiektu, na którym metoda została wywołana.
- W zwykłej funkcji, this może odnosić się do globalnego obiektu (w przeglądarkach jest to window).
- W funkcji strzałkowej this jest dziedziczone z kontekstu otaczającego, tzn. nie jest wiązane na nowo, jak to ma miejsce w zwykłych funkcjach, lecz odnosi się do wartości this z zakresu, w którym funkcja strzałkowa została zdefiniowana.

Funkcje

Przekazywanie argumentów w do funkcji strzałkowej

Typ parametru	Przykład Kodu	Opis
Bez parametrów	const przywitajSie = () => console.log('Witaj świecie!');	Funkcja strzałkowa bez parametrów wywołuje instrukcję console.log.
Jeden parametr	const podwoj = x => x * 2;	Funkcja strzałkowa z jednym parametrem, zwracająca podwojoną wartość tego parametru. Nawiasy nie są potrzebne.
Dwa parametry	const dodaj = (a, b) => a + b;	Funkcja strzałkowa z dwoma parametrami, zwracająca ich sumę. Parametry są umieszczone w nawiasach.
Więcej niż dwa	const suma = (a, b, c) => a + b + c;	Funkcja strzałkowa z trzema parametrami, zwracająca sumę wartości. Wszystkie parametry są umieszczone w nawiasach.
Parametry domyślne	const przywitaj = (imie = 'Gościu') => \Witaj, ${imie}!`;`	Funkcja strzałkowa z jednym parametrem i wartością domyślną. Jeżeli parametr nie zostanie przekazany, użyta zostanie wartość domyślna.
Rest parameters	const sumaWszystkich = (...args) => args.reduce((sum, curr) => sum + curr, 0);	Funkcja strzałkowa przyjmująca nieokreśloną liczbę argumentów. Wykorzystuje rest parameters do zsumowania wszystkich przekazanych wartości.

Obiekty

Obiekty

to kolekcja powiązanych danych i funkcjonalności, składająca się z wielu zmiennych i funkcji, które są nazywane właściwościami i metodami obiektu.

Właściwości mogą być wartościami dowolnego typu, takiego jak liczby, łańcuchy znaków czy inne obiekty.

Metody są funkcjami związanymi z obiektem, które mogą manipulować jego właściwościami lub wykonywać zadania związane z obiektem.

```
const prostokat = {
  szerokosc: 5,
  wysokosc: 10,

  // Metoda do obliczania pola powierzchni prostokąta
  obliczPole: function() {
    return this.szerokosc * this.wysokosc;
  },

  // Metoda do obliczania obwodu prostokąta
  obliczObwod: function() {
    return 2 * (this.szerokosc + this.wysokosc);
  }
};

// Użycie metod obiektu prostokat
console.log(`Pole prostokąta: ${prostokat.obliczPole()} jednostek kwadratowych`);
console.log(`Obwód prostokąta: ${prostokat.obliczObwod()} jednostek`);
```

Obiekty

Zalety używania obiektów

Strukturyzacja danych

Obiekty pozwalają na zorganizowanie danych dotyczących jednej "rzeczy" lub "entytetu" w jedną logiczną jednostkę.

Modularyzacja i ponowne wykorzystanie kodu

Metody (funkcje w obrębie obiektów) mogą być używane wielokrotnie dla różnych instancji obiektów, co sprzyja ponownemu użyciu kodu i łatwości jego utrzymania.

Zakapsułkowanie

Obiekty umożliwiają zakapsułkowanie związanych ze sobą danych i funkcji, co zmniejsza ryzyko konfliktów w nazwach i ułatwia zarządzanie stanem.

Przedstawianie rzeczywistości

Obiekty są używane do modelowania bytów z rzeczywistego świata, takich jak użytkownicy, zamówienia, produkty itp.

Umożliwiają one przechowywanie informacji w jednym miejscu, ułatwiając manipulowanie danymi i implementowanie logiki biznesowej

Obiekty

Tworzenie Obiektów

Literały Obiektowe

Najprostszy i najczęściej używany sposób tworzenia obiektów.

Definiowany przez umieszczenie listy zerowej lub więcej par klucz-wartość w nawiasach klamrowych {}.

```
const samochod = {
  marka: 'Ford',
  model: 'Mustang',
  rok: 1969
};
```

Konstruktor Object()

Używa wbudowanego konstruktora Object() do tworzenia nowego obiektu.

Sposób mniej popularny, ale użyteczny, gdy potrzebujemy instancji czystego obiektu.

```
// Utworzenie nowego obiektu za pomocą konstruktora Object()
const obiekt = new Object();

// Przypisanie nowej właściwości do utworzonego obiektu
obiekt.wlasciwosc = 'wartość';
```

Obiekty

Tworzenie Obiektów

Funkcje Konstruujące

Pozwalają na tworzenie skomplikowanych obiektów, które mogą zawierać metody i właściwości.

Definiowane przez zwykłe funkcje, ale używane z operatorem new do tworzenia instancji obiektów.

Umożliwiają inicjalizację obiektów z przekazanymi wartościami.

```
function Osoba(imie, wiek) {
  this.imie = imie;
  this.wiek = wiek;
  this.przedstawSie = function() {
    console.log(`Mam na imię ${this.imie} i mam ${this.wiek} lat.`);
  };
}
const jan = new Osoba('Jan', 30);
jan.przedstawSie(); // "Mam na imię Jan i mam 30 lat."
```

Obiekty

Prototyp

to mechanizm, za pomocą którego obiekty mogą dziedziczyć właściwości i metody od innych obiektów.

Obiekty mają właściwość [[Prototype]], która wskazuje na inny obiekt, zwany prototypem.

Dziedziczenie Prototypowe

- Obiekty mogą używać właściwości i metod swoich prototypów, jakby były one ich własnymi.
- Mechanizm ten umożliwia ponowne używanie kodu i tworzenie hierarchii obiektów.
- Można dodawać nowe właściwości i metody do prototypu w dowolnym momencie, co wpłynie na wszystkie obiekty utworzone na bazie tego prototypu.

```
function Osoba(imie) {
  this.imie = imie;
}

Osoba.prototype.przedstawSie = function() {
  console.log(`Mam na imię ${this.imie}.`);
};

const jan = new Osoba('Jan');
jan.przedstawSie(); // "Mam na imię Jan."

// Dodanie console.dir() do zbadania obiektu 'jan'
console.dir(jan);
```

Obiekty

Dziedziczenie

jest realizowane za pomocą prototypów. Każdy obiekt może mieć prototyp, od którego "dziedziczy" właściwości i metody.

Za pomocą Object.create() można ustawić prototyp dla danego obiektu, co jest podstawą dziedziczenia.

Przykład: obiekt Pracownik może dziedziczyć od ogólniejszego obiektu Osoba.

Obiekty potomne mogą nadpisywać lub rozszerzać funkcjonalność dziedziczoną od swoich prototypów, dodając własne metody lub modyfikując istniejące.

```
const Osoba = {
  przedstawSie: function() {
    console.log(`Mam na imię ${this.imie}.`);
  }
};

const Pracownik = Object.create(Osoba);
Pracownik.przedstawSieJakoPracownik = function() {
  console.log(`Mam na imię ${this.imie} i pracuję jako ${this.stanowisko}.`);
};

// Tworzenie instancji Pracownik z bezpośrednim ustawieniem właściwości
const jan = Object.create(Pracownik);
jan.imie = 'Jan';
jan.stanowisko = 'Programista';

// Wywołanie metod
jan.przedstawSie(); // Korzysta z metody zdefiniowanej w Osoba
jan.przedstawSieJakoPracownik(); // Korzysta z metody zdefiniowanej w Pracownik
```

Obiekty

JSON

JSON.stringify()

ta metoda przekształca obiekt JavaScript w ciąg tekstowy JSON.

Jest to przydatne, gdy chcemy wysłać dane obiektu do serwera lub zapisać je w lokalnym przechowywaniu.

```
const obiekt = { imie: 'Jan', wiek: 30 };
const tekstJSON = JSON.stringify(obiekt);
// tekstJSON to teraz '{"imie":"Jan","wiek":30}'
```

JSON.parse()

ta metoda przekształca ciąg tekstowy w formacie JSON na obiekt JavaScript.

Jest to niezbędne, gdy otrzymujemy dane w formacie JSON z serwera i chcemy je używać w naszym kodzie JavaScript.

```
const tekstJSON = '{"imie":"Jan","wiek":30}';
const obiekt = JSON.parse(tekstJSON);
// obiekt to teraz { imie: 'Jan', wiek: 30 }
```

Klasy

Klasy

Klasy w ES6 wprowadzają bardziej zrozumiałą i czytelną formę zapisu dla typowych wzorców obiektowych w JavaScript, pełniąc funkcję uproszczenia składniowego nad istniejącym systemem opartym na prototypach.

```
class Osoba {
  constructor(imie, wiek) {
    this.imie = imie;
    this.wiek = wiek;
  }

  przedstawSie() {
    console.log(`Mam na imię ${this.imie} i mam ${this.wiek} lat.`);
  }
}
```

Podstawowa składnia klasy w ES6 obejmuje słowo kluczowe class, nazwę klasy, a następnie blok klasy zawierający konstruktor i definicje metod.

Aby utworzyć instancję klasy, używamy słowa kluczowego new.

```
const jan = new Osoba('Jan', 30);
jan.przedstawSie(); // "Mam na imię Jan i mam 30 lat."
```

Klasy

Klasy

Klasy w ES6 wprowadzają bardziej zrozumiałą i czytelną formę zapisu dla typowych wzorców obiektowych w JavaScript, pełniąc funkcję uproszczenia składniowego nad istniejącym systemem opartym na prototypach.

```
class Osoba {
  constructor(imie, wiek) {
    this.imie = imie;
    this.wiek = wiek;
  }

  przedstawSie() {
    console.log(`Mam na imię ${this.imie} i mam ${this.wiek} lat.`);
  }
}
```

Podstawowa składnia klasy w ES6 obejmuje słowo kluczowe class, nazwę klasy, a następnie blok klasy zawierający konstruktor i definicje metod.

Aby utworzyć instancję klasy, używamy słowa kluczowego new.

```
const jan = new Osoba('Jan', 30);
jan.przedstawSie(); // "Mam na imię Jan i mam 30 lat."
```

Klasy

Dziedziczenie i Rozszerzanie Klas

Klasy w ES6 mogą dziedziczyć od innych klas za pomocą słowa kluczowego extends.

Można nadpisywać metody nadrzędne lub odwoływać się do nich za pomocą super.

```
class Pracownik extends Osoba {
  constructor(imie, wiek, stanowisko) {
    super(imie, wiek);
    this.stanowisko = stanowisko;
  }

  przedstawSie() {
    super.przedstawSie();
    console.log(`Pracuję jako ${this.stanowisko}.`);
  }
}
```

Przykłady Użycia Obiektów

Obiekt Konfiguracyjny

są często stosowane do przechowywania ustawień aplikacji, które mogą być łatwo zmieniane lub rozszerzane.

```
const konfiguracja = {
  jasnyMotyw: true,
  jezyk: 'pl',
  wolumen: 70
};
```

Obiekty Reprezentujące Modele Danych

używane do reprezentowania modeli danych, takich jak użytkownicy, produkty, zamówienia itp.

```
const uzytkownik = {
  imie: 'Jan',
  nazwisko: 'Kowalski',
  email: 'jan.kowalski@example.com'
};

const zamowienie = {
  id: 1234,
  produkty: ['produkt1', 'produkt2'],
  kwota: 299.99,
  uzytkownik: uzytkownik  // Dodanie referencji do obiektu uzytkownik
};

// Dostęp do danych użytkownika poprzez obiekt zamowienie
console.log(zamowienie.uzytkownik.imie);  // Wyświetli "Jan"
```

Przykłady Użycia Obiektów

Singleton

to wzorzec projektowy, w którym klasa ma tylko jedną instancję w całym cyklu życia aplikacji. Jest często używany do zarządzania globalnym stanem aplikacji

```
const aplikacja = {
  stan: {
    zalogowanyUzytkownik: null,
    ustawienia: {
      jezyk: 'pl'
    }
  },
  zaloguj(uzytkownik) {
    this.stan.zalogowanyUzytkownik = uzytkownik;
  },
  wyloguj() {
    this.stan.zalogowanyUzytkownik = null;
  }
};
```

Homo PC