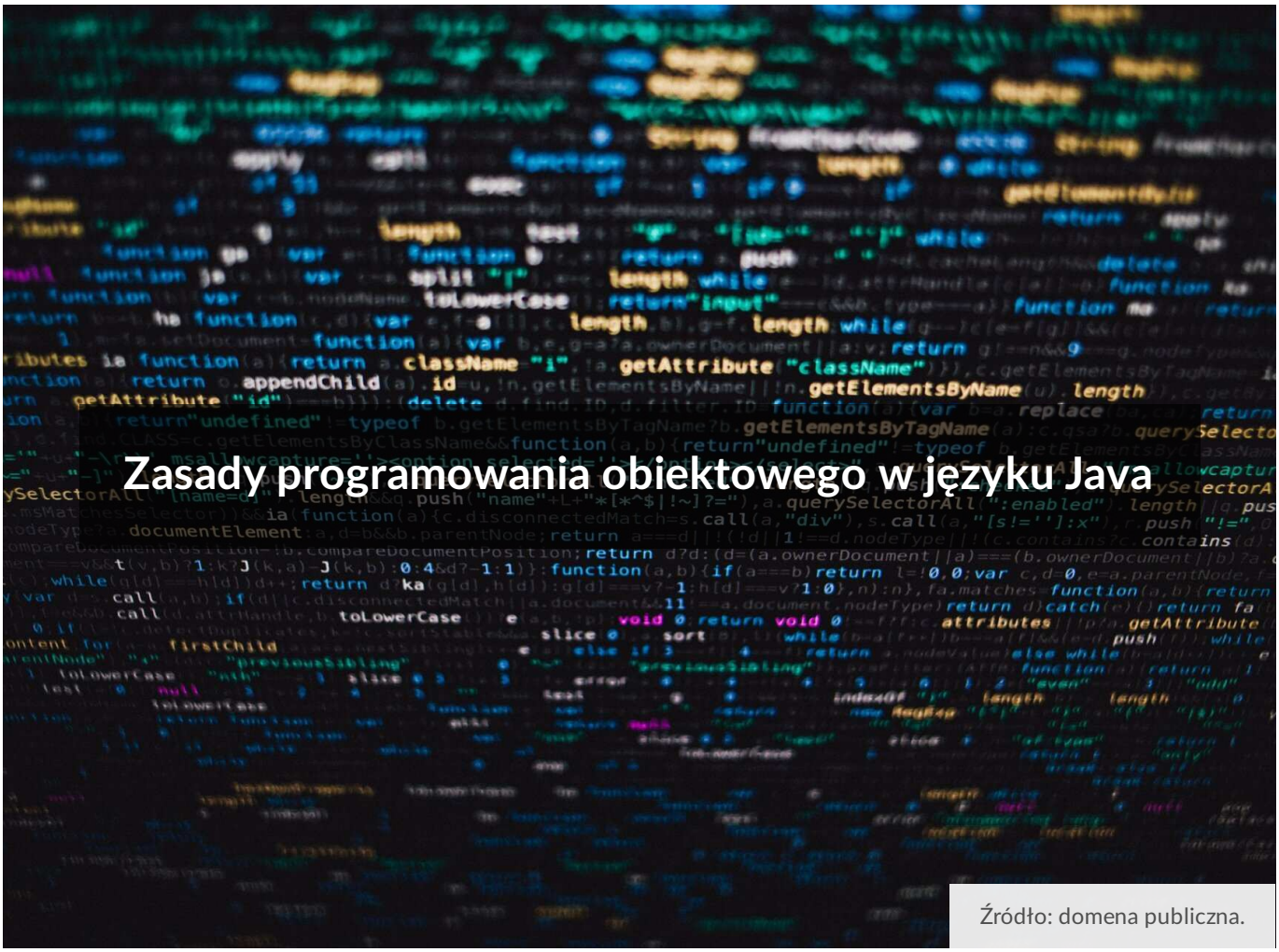
Zasady programowania obiektowego w języku Java

- Wprowadzenie
- Przeczytaj
- Prezentacja multimedialna
- Sprawdź się
- Dla nauczyciela



Źródło: domena publiczna.

Wiemy już, że programowanie obiektowe składa się z kilku zasad. W e-materiale Zasady programowania obiektowego dowiedzieliśmy się, czym są abstrakcja, dziedziczenie, polimorfizm i hermetyzacja.

W tym e-materiale zajmiemy się realizacją zasad programowania obiektowego w języku Java.

Ciekawi cię, jak wyglądają implementacje w innych językach programowania? Możesz się z nimi zapoznać w dwóch pozostałych e-materiałach z tej serii:

- Zasady programowania obiektowego w języku C++,
- Zasady programowania obiektowego w języku Python.

Twoje cele

- Zaimplementujesz klasy w języku Java, które będą reprezentowały hierarchiczne struktury geometryczne.
- Połączysz wiedzę o dziedziczeniu w języku Java z paradygmatami hermetyzacji oraz polimorfizmu.
- Skonstruujesz model obiektowy struktur danych w języku Java, dzięki któremu obliczysz pola oraz obwody figur.

Przeczytaj

Metody wirtualne w języku Java

Java jest językiem czysto obiektowym. Dzięki temu realizuje wiele zagadnień oraz standardowych pojęć programowania obiektowego. Jak już wiemy, w kontekście dziedziczenia istotnym pojęciem są metody wirtualne, które mogą być dziedziczone albo nadpisywane w funkcjach pochodnych. Ze względu na swój charakter nie mogą być zatem statyczne. W języku Java ustanowiono, że każda metoda będzie domyślnie wirtualna. Zatem każda metoda niestatyczna może zostać nadpisana lub odziedziczona w klasie pochodnej.

> **Ciekawostka**
>
> Jeśli chcemy utworzyć metodę, której nie można nadpisać w klasie pochodnej, musimy po deklaracji funkcji dopisać słowo kluczowe `final`.

Istnieje również pojęcie metody czysto wirtualnej, czyli takiej, która musi zostać nadpisana w klasie pochodnej. Metody czysto wirtualne z uwagi na swój charakter mogą występować tylko w klasach abstrakcyjnych. W języku Java takie metody poprzedzamy słowem kluczowym `abstract`, podobnie jak w przypadku klas abstrakcyjnych. Z tego powodu metody czysto wirtualne nazywa się czasem abstrakcyjnymi.

Hermetyzacja w języku Java

W języku Java mamy do dyspozycji trzy standardowe modyfikatory dostępu, o których była mowa w e-materiale Paradygmaty programowania obiektowego czyli: prywatny (`private`), publiczny (`public`) oraz chroniony (`protected`), a także domyślny, w którym brak słowa kluczowego poprzedzającego element. Domyślnie dostęp do elementu jest publiczny, jednak tylko wewnątrz tego samego pakietu. Pakiety w języku Java nie są jednak częścią tego kursu, zatem będziemy pomijać domyślny modyfikator dostępu.

Polimorfizm w języku Java

Polimorfizm oznacza wielopostaciowość. W programowaniu paradygmat ten oznacza możliwość traktowania danych (np. zmiennych lub obiektów) w różny sposób. Ten sam obiekt może mieć zatem wiele różnych postaci, z których każda może rozszerzać lub modyfikować jego własności. W języku Java zawarto wiele mechanizmów, które pozwalają realizować polimorfizm.

Przykład

W celu obserwacji paradygmatów programowania obiektowego posłużmy się przykładem, w którym utworzymy dwie klasy: abstrakcyjną klasę bazową `Owoc` oraz klasę pochodną `Jablko`.

```
abstract class Owoc  {
    // przykładowe pole w klasie Owoc
    protected double masa;

    // przykładowa metoda w klasie Owoc
    public void ugryz() {
        // ugryzienie powoduje zmniejszenie masy owocu
        masa -= 0.01;
    }
}

class Jablko extends Owoc {
    // przykładowe pole w klasie Owoc, które nie występuje
    // w klasie pierwotnej
    private int liczbaPestek;

    // przykładowa metoda w klasie Jablko, która nie występuje
    // w klasie pierwotnej
    public void zabierzPestke() {
        liczbaPestek--;
    }

    // przykładowe nadpisanie metody wirtualnej w klasie Owoc
    @Override
    public void ugryz() {
        System.out.println("Ugryzienie jablka");

        // ugryzienie jabłka może wiązać się ze
        // zjedzeniem pestki z prawdopodobieństwem 10%
        if (Math.random() < 1) {
            liczbaPestek -= 1;
        }

        // wywołanie metody bazowej na danym obiekcie
        super.ugryz();
    }

```

```
38      // publiczny konstruktor
39      public Jablko(double masa, int liczbaPestek) {
40          this.masa = masa;
41          this.liczbaPestek = liczbaPestek;
42      }
43  }
```

W języku Java przewidziano rodzaj metadanych, który nazywamy adnotacją. Przykład można zaobserwować w wersie 24. Określenie `@Override` oznacza, że nadpisujemy metodę z klasy bazowej. Adnotacja ta jest opcjonalna, jednak sprawia, że kod zyskuje na czytelności.

Warto zwrócić uwagę na wers nr 35, w którym użyliśmy konstrukcji `super()`. Konstrukcja ta powoduje odwołanie do klasy bazowej. Dzięki temu możemy rozszerzać metody, jednocześnie wykorzystując bazowy kod. Takie możliwości są zgodne z paradygmatem polimorfizmu, ponieważ klasa `Jablko` traktowana jest również jako klasa `Owoc`. Wywołanie metody `super.ugryz()` spowoduje przejście do metody `ugryz()` w klasie `Owoc`. Wers nr 35 będzie skutkować zatem zmniejszeniem masy obiektu klasy `Jablko` w taki sam sposób, który zdefiniowano w klasie bazowej. Jednak w przypadku klasy `Jablko` wiąże się to również z prawdopodobnym zmniejszeniem liczby pestek.

W dalszej części kodu występuje konstruktor. Dzięki ustawieniu widoczności pola `masa` na `protected` możemy zmodyfikować je w klasie pochodnej. Gdyby pole to było prywatne, aby zmodyfikować jego wartość, musielibyśmy zdefiniować wirtualny konstruktor w klasie `Owoc` oraz użyć konstrukcji `super()`. Wówczas kod mógłby wyglądać w następujący sposób:

```
1   abstract class Owoc  {
2       // przykładowe pole w klasie Owoc
3       private double masa;
4
5       // przykładowa metoda w klasie Owoc
6       public void ugryz() {
7           // ugryzienie powoduje zmniejszenie masy owocu
8           masa -= 0.01;
9       }
10
11      // publiczny wirtualny konstruktor
12      public Owoc(double masa) {
13          this.masa = masa;
14      }
15  }
```

```
16
17 class Jablko extends Owoc {
18     // przykładowe pole w klasie Owoc, które nie występuje
19     // w klasie pierwotnej
20     private int liczbaPestek;
21
22     // przykładowa metoda w klasie Jablko, która nie występuje
23     // w klasie pierwotnej
24     public void zabierzPestke() {
25         liczbaPestek--;
26     }
27
28     // przykładowe nadpisanie metody wirtualnej w klasie Owoc
29     @Override
30     public void ugryz() {
31         System.out.println("Ugryzienie jablka");
32
33         // ugryzienie jabłka może wiązać się ze
34         // zjedzeniem pestki z prawdopodobieństwem 10%
35         if (Math.random() < 1) {
36             liczbaPestek -= 1;
37         }
38
39         // wywołanie metody bazowej na danym obiekcie
40         super.ugryz();
41     }
42
43     // publiczny konstruktor
44     public Jablko(double masa, int liczbaPestek) {
45         super(masa);
46         this.liczbaPestek = liczbaPestek;
47     }
48 }
```

Niezależnie od wybranego rozwiązania, paradygmat polimorfizmu zawarty w języku Java pozwala nam traktować klasy pochodne również jako klasy bazowe. Dzięki temu, możemy utworzyć instancję klasy `Jablko`, którą przypiszemy do zmiennej referencyjnej typu `Owoc`. Taki zabieg spowoduje, że zmienna ta będzie mogła wywoływać jedynie te metody, które są zawarte w klasie `Owoc`, metody z klasy `Jablko` zostaną ukryte. Mechanizm ten umożliwia np. stworzenie wielu klas pochodnych klasy `Owoc` (np. `Pomarancza`, `Arbuz`), a następnie przechowywanie ich instancji w jednej tablicy typu `Owoc`.

```
1  public class Main {
2      public static void main(String[] args) {
3          // nowe instancje klasy Jablko
4          Jablko jablko1 = new Jablko(0.3, 3);
5          Jablko jablko2 = new Jablko(0.2, 2);
6
7          // utworzenie tablicy typu owoc
8          Owoc[] tablicaOwocow = { jablko1, jablko2 };
9
10         // dla każdego owocu z tablicyOwocow
11         for (Owoc owoc : tablicaOwocow) {
12             // poniższa linijka wypisze: "Ugryzienie jablka"
13             owoc.ugryz();
14             // poniższa linijka jest niepoprawna
15             owoc.zabierzPestke();
16         }
17     }
18 }
```

Warto zaznaczyć, że w wypadku wersu nr 13 zostanie wywołana metoda z klasy pochodnej – z klasy `Jablko`, ponieważ tylko te owoce zawarliśmy w tablicy.

Wers nr 14 nie jest poprawny, ponieważ nie każda klasa oparta na klasie `Owoc` musi zawierać metodę `zabierzPestke()`.

Całość kodu, w celu jego przetestowania, wygląda następująco:

```
1  abstract class Owoc  {
2      // przykładowe pole w klasie Owoc
3      private double masa;
4
5      // przykładowa metoda w klasie Owoc
6      public void ugryz() {
7          // ugryzienie powoduje zmniejszenie masy owocu
8          masa -= 0.01;
9      }
10
11     // publiczny wirtualny konstruktor
12     public Owoc(double masa) {
13         this.masa = masa;
14     }
```

```
}

class Jablko extends Owoc {
    // przykładowe pole w klasie Owoc, które nie występuje
    // w klasie pierwotnej
    private int liczbaPestek;

    // przykładowa metoda w klasie Jablko, która nie występuje
    // w klasie pierwotnej
    public void zabierzPestke() {
        liczbaPestek--;
    }

    // przykładowe nadpisanie metody wirtualnej w klasie Owoc
    @Override
    public void ugryz() {
        System.out.println("Ugryzienie jablka");

        // ugryzienie jabłka może wiązać się ze
        // zjedzeniem pestki z prawdopodobieństwem 10%
        if (Math.random() < 1) {
            liczbaPestek -= 1;
        }

        // wywołanie metody bazowej na danym obiekcie
        super.ugryz();
    }

    // publiczny konstruktor
    public Jablko(double masa, int liczbaPestek) {
        super(masa);
        this.liczbaPestek = liczbaPestek;
    }
}

public class Main {
    public static void main(String[] args) {
        // nowe instancje klasy Jablko
        Jablko jablko1 = new Jablko(0.3, 3);
        Jablko jablko2 = new Jablko(0.2, 2);

        // utworzenie tablicy typu owoc
```

```
57          Owoc[] tablicaOwocow = { jablko1, jablko2 };
58
59          // dla każdego owocu z tablicyOwocow
60          for (Owoc owoc : tablicaOwocow) {
61              // poniższa linijka wypisze: "Ugryzienie jablka"
62              owoc.ugryz();
63              // poniższa linijka jest niepoprawna
64              // owoc.zabierzPestke();
65          }
66      }
67  }
```

Słownik

> **metadane**
>
> informacje opisujące dane; stosowane są do opisu zasobów informacji oraz obiektów informacji
>
> **metoda czysto wirtualna**
>
> metoda, która musi zostać nadpisana w klasie pochodnej; może występować tylko w klasie abstrakcyjnej
>
> **modyfikatory dostępu**
>
> słowa kluczowe, dzięki którym definiujemy zakres widoczności elementów takich jak klasy, pola oraz metody

Prezentacja multimedialna

Polecenie 1

Przeanalizuj prezentację multimedialną, w której zgodnie z paradygmatami programowania obiektowego tworzone są klasy reprezentujące figury geometryczne. Zastanów się, w jaki sposób można rozszerzać utworzony program.

1

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PLzKg8e5h

Pisząc program obiektowy, powinniśmy kierować się omawianymi paradygmatami programowania. Przypomnijmy je sobie. Są to:

- abstrakcja,
- hermetyzacja,
- polimorfizm,
- dziedziczenie.

2

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PLzKg8e5h

Wiele spośród praktyk, które poznajesz, jest zupełnie zbędnych z punktu widzenia komputera. Napisany prawidłowo program nie musi wykorzystywać abstrakcji, hermetyzacji, dziedziczenia czy polimorfizmu. Możliwe jest napisanie tego samego programu w sposób wykorzystujący paradygmaty programowania obiektowego, a także nie wykorzystując nawet programowania strukturalnego. Realizacja niektórych paradygmatów może na początku wydawać się stratą czasu. Problem pojawia się

jednak w momencie, gdy rozbudowa programu przestaje być możliwa ze względu na zbyt skomplikowany kod źródłowy. Dobranie odpowiedniego wzorca projektowego pozwala w znaczący sposób opóźnić ten moment lub nawet go wyeliminować.

3

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PLzKg8e5h

Napiszemy prosty program w skomplikowany obiektowy sposób, aby przećwiczyć programowanie obiektowe. Zadaniem programu będzie wypisanie pola prostokąta oraz kwadratu.

4

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PLzKg8e5h

Zacznijmy od pokazania abstrakcji – utworzymy klasę abstrakcyjną `Ksztalt`. Klasa ta nie będzie zawierała żadnych zmiennych, a jedynie metody, które musi implementować klasa pochodna. W programowaniu obiektowym takie klasy nazywamy interfejsami. Warto podkreślić, że interfejs nie definiuje pól klasy, a jedynie publiczne metody czysto wirtualne. W języku Java interfejs definiujemy w następujący sposób:

```
interface Ksztalt {
    double pole();
}
```

Do interfejsu dodaliśmy przy okazji metodę czysto wirtualną typu `double` – `pole()`. Dzięki temu zabiegowi, wszystkie klasy

pochodne klasy `Ksztalt` muszą zapewnić implementację metody `pole()`.

5

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PLzKg8e5h

Utwórzmy klasę implementującą `Ksztalt` – `Prostokat`. Aby zademonstrować hermetyzację, umieścimy w niej dwie prywatne zmienne typu zmiennoprzecinkowego – `szerokosc` oraz `wysokosc`.

Musimy nadpisać metodę `pole()` tak, aby zwracała ona pole naszego prostokąta, czyli `szerokosc * wysokosc`.

Aby klasa `Prostokat` była gotowa, powinniśmy utworzyć jej konstruktor. Konstruktor będzie przyjmował dwie zmienne – `wysokosc` oraz `szerokosc`.

```
1  class Prostokat implements
   Ksztalt {
2      private double
   szerokosc;
3      private double
   wysokosc;
4
5      public
   Prostokat(double
   szerokosc, double
   wysokosc) {
6          this.szerokosc =
   szerokosc;
7          this.wysokosc =
   wysokosc;
8      }
9
10     @Override
11     public double pole() {
```

```
12          return (szerokosc
* wysokosc);
13      }
14  }
```

6

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PLzKg8e5h

Spróbujmy utworzyć pochodną klasy `Prostokat` – klasę `Kwadrat`.

W klasie tej zdefinujemy konstruktor, który będzie przyjmował jedną wartość – `dlugosc_boku`.

Ponieważ w klasie `Prostokat` pola `szerokosc` i `wysokosc` ustawione są w sekcji `private`, mamy dwie możliwości. Możemy zmienić ich zakres z `private` na `protected` lub skorzystać z poliformicznego mechanizmu wywołania konstruktora bazowego. W celu przećwiczenia polimorfizmu skorzystamy z drugiej opcji.

Zauważ, że w klasie `Kwadrat` nie musimy już niczego modyfikować, ponieważ metody oraz pola klasy `Prostokat` w wypadku kwadratu zachowują pełną zgodność.

```
1  class Kwadrat extends
Prostokat {
2      public Kwadrat(double
dlugosc_boku) {
3
   super(dlugosc_boku,
dlugosc_boku);
4      }
5  }
```

7

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PLzKg8e5h

Nadeszła pora na przetestowanie napisanych klas. W tym celu utworzymy główną funkcję programu, a w niej tablicę typu `Ksztalt`. W języku Java zastosowano polimorficzny mechanizm, dzięki któremu możemy przechowywać referencje na obiekty, które są tego samego typu bazowego. W utworzonej tablicy będziemy mogli zatem przechowywać zarówno referencje na obiekty typu `Prostokat`, jak i `Kwadrat`.

```
1  public class Main {
2      public static void
   main(String[] args) {
3          Ksztalt[] ksztalty
   = new Ksztalt[2];
4          ksztalty[0] = new
   Prostokat(2, 3.2);
5          ksztalty[1] = new
   Kwadrat(5.2);
6
7          for (int i = 0; i
   < 2; ++i) {
8
    System.out.println(ksztal
   ty[0].pole());
9
    System.out.println(ksztal
   ty[1].pole());
10         }
11     }
12 }
```

8

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PLzKg8e5h

Cały program prezentuje się następująco:

```
interface Ksztalt {
    double pole();
}

class Prostokat implements Ksztalt {
    private double szerokosc;
    private double wysokosc;

    public Prostokat(double szerokosc, double wysokosc) {
        this.szerokosc = szerokosc;
        this.wysokosc = wysokosc;
    }

    @Override
    public double pole() {
        return (szerokosc * wysokosc);
    }
}

class Kwadrat extends Prostokat {
    public Kwadrat(double dlugosc_boku) {

     super(dlugosc_boku, dlugosc_boku);
    }
}

public class Main {
    public static void main(String[] args) {
```

```
28          Ksztalt[] ksztalty
= new Ksztalt[2];
29          ksztalty[0] = new
Prostokat(2, 3.2);
30          ksztalty[1] = new
Kwadrat(5.2);
31
32          for (int i = 0; i
< 2; ++i) {
33
 System.out.println(ksztal
ty[0].pole());
34
 System.out.println(ksztal
ty[1].pole());
35          }
36      }
37  }
```

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PLzKg8e5h

9

Uruchomiony program wyświetla następujące wyjście:

```
1  6.4
2  27.040000000000003
3  6.4
4  27.040000000000003
```

Należy pamiętać, że wynik, który przechowujemy w zmiennej typu zmiennoprzecinkowego, nie zawsze będzie w 100% dokładny. Jednak w naszym przypadku błąd jest niewielki.

Polecenie 2

Utwórz klasę `Kolo` implementującą interfejs `Ksztalt`, w której zawrzesz prywatne pole typu `double` o nazwie `promien`. W klasie dodaj konstruktor, który przyjmuje wartość promienia. Pamiętaj, aby utworzyć metodę `pole()`. Skorzystaj ze stałej `PI` z biblioteki `Math`. Przetestuj utworzoną klasę przez utworzenie obiektu typu `Kolo` w tablicy `ksztalty`.

Polecenie 3

Do interfejsu `Ksztalt` dodaj metodę `obwod()`. W każdej klasie implementującej ten interfejs nadpisz jej definicję w taki sposób, aby zwracała obwód danej figury. Aby przetestować utworzoną metodę, w pętli `for` w funkcji `main()` dodaj kod, odpowiedzialny za wypisanie obwodu każdej figury z tablicy `ksztalty`.

Sprawdź się

Pokaż ćwiczenia: 

Ćwiczenie 1

Utwórz klasę abstrakcyjną `Bryla`, w której zadeklarujesz dwie metody czysto wirtualne typu `double objetosc()` oraz `polePowierzchni()`. Utwórz klasę `Prostopadloscian`, która posiada prywatne pola typu `double dlugosc`, `szerokosc` oraz `wysokosc`. Nadpisz metody `objetosc()` oraz `polePowierzchni()` w taki sposób, aby zwracały odpowiednie dla prostopadłościanu wielkości. Utwórz konstruktor, który ustawi pola `dlugosc`, `szerokosc` oraz `wysokosc`. Następnie utwórz instancję klasy `Prostopadloscian`, którą przechowasz w zmiennej referencyjnej typu `Bryla`. Wypisz objętość oraz pole powierzchni prostopadłościanu o wymiarach:

```
dlugosc = 3
szerokosc = 2
wysokosc = 5
```

Oddziel je znakiem nowej linii.

Specyfikacja problemu

Dane:

- `Bryla` – klasa
- `double objetosc()` – metoda wirtualna klasy `Bryla`
- `polePowierzchni()` – metoda wirtualna klasy `Bryla`
- `Prostopadloscian` – klasa
- `dlugosc` – prywatne pole typu `double` klasy `Prostopadloscian`
- `szerokosc` – prywatne pole typu `double` klasy `Prostopadloscian`
- `wysokosc` – prywatne pole typu `double` klasy `Prostopadloscian`

Wynik:

Program wypisuje objętość oraz pole powierzchni prostopadłościanu o wymiarach:

```
dlugosc = 3
szerokosc = 2
wysokosc = 5
```

Wyniki są oddzielone znakiem nowej linii.

Twoje zadania

1. Program wypisuje objętość oraz pole powierzchni prostopadłościanu o wymiarach 3x2x5.

```java
// W tym miejscu utwórz implementację klasy Bryla oraz Prostopadloscian.

public class Main {
    public static void main(String[] args) {
        Bryla b1 = new Prostopadloscian(3, 2, 5);

        System.out.println(b1.objetosc());
        System.out.println(b1.polePowierzchni());
    }
}
```

```

```

Ćwiczenie 2

Korzystając z przedstawionego kodu, utwórz klasę `Policjant`, która będzie dziedziczyć po klasie `Pracownik`. W klasie `Policjant` zadeklaruj tylko jeden konstruktor, który będzie przyjmował dwa parametry typu `String`: `imie` oraz `nazwisko`. Zadbaj, aby pole `miejscePracy` w klasie `Policjant` było ustawione na `komisariat`. Następnie, przez nadpisanie metody `przedstawSię()`, popraw błąd językowy tak, aby program wypisywał komunikat `Nazywam sie Jan Kowalski. Pracuje na komisariacie.` Nie modyfikuj istniejącej klasy `Pracownik` oraz funkcji głównej programu.

Twoje zadania

1. Program wypisuje komunikat "Nazywam sie Jan Kowalski. Pracuje na komisariacie."

```
class Pracownik {
    private String imie;
    private String nazwisko;
    private String miejscePracy;

    public Pracownik(String imie, String nazwisko, String
miejscePracy){
        this.imie = imie;
        this.nazwisko = nazwisko;
        this.miejscePracy = miejscePracy;
    }

    public void przedstawSie() {
        System.out.println("Nazywam sie " + imie + " " +
```

```

```

Dla nauczyciela

Autor: Maurycy Gast
Przedmiot: Informatyka

Temat: Zasady programowania obiektowego w języku Java

Grupa docelowa:

Szkoła ponadpodstawowa, liceum ogólnokształcące, technikum, zakres podstawowy i rozszerzony

Podstawa programowa:

Cele kształcenia – wymagania ogólne

II. Programowanie i rozwiązywanie problemów z wykorzystaniem komputera oraz innych urządzeń cyfrowych: układanie i programowanie algorytmów, organizowanie, wyszukiwanie i udostępnianie informacji, posługiwanie się aplikacjami komputerowymi.

IV. Rozwijanie kompetencji społecznych, takich jak: komunikacja i współpraca w grupie, w tym w środowiskach wirtualnych, udział w projektach zespołowych oraz zarządzanie projektami.

Treści nauczania – wymagania szczegółowe

II. Programowanie i rozwiązywanie problemów z wykorzystaniem komputera i innych urządzeń cyfrowych.

Zakres podstawowy. Uczeń:

2) do realizacji rozwiązań problemów prawidłowo dobiera środowiska informatyczne, aplikacje oraz zasoby, wykorzystuje również elementy robotyki;

Zakres rozszerzony. Uczeń spełnia wymagania określone dla zakresu podstawowego, a ponadto:

2) stosuje zasady programowania strukturalnego i obiektowego w rozwiązywaniu problemów;

3) sprawnie posługuje się zintegrowanym środowiskiem programistycznym przy pisaniu, uruchamianiu i testowaniu programów;

IV. Rozwijanie kompetencji społecznych.

Zakres podstawowy. Uczeń:

1) aktywnie uczestniczy w realizacji projektów informatycznych rozwiązujących problemy z różnych dziedzin, przyjmuje przy tym różne role w zespole realizującym projekt i prezentuje efekty wspólnej pracy;

Kształtowane kompetencje kluczowe:

- kompetencje cyfrowe;
- kompetencje osobiste, społeczne i w zakresie umiejętności uczenia się;
- kompetencje matematyczne oraz kompetencje w zakresie nauk przyrodniczych, technologii i inżynierii.

Cele operacyjne (językiem ucznia):

- Zaimplementujesz klasy w języku Java, które będą reprezentowały hierarchiczne struktury geometryczne.
- Połączysz wiedzę o dziedziczeniu w języku Java z paradygmatami hermetyzacji oraz polimorfizmu.
- Skonstruujesz model obiektowy struktur danych w języku Java, dzięki któremu obliczysz pola oraz obwody figur.

Strategie nauczania:

- konstruktywizm;
- konektywizm.

Metody i techniki nauczania:

- dyskusja;
- rozmowa nauczająca z wykorzystaniem multimedium i ćwiczeń interaktywnych;
- ćwiczenia praktyczne.

Formy pracy:

- praca indywidualna;
- praca w parach;
- praca w grupach;
- praca całego zespołu klasowego.

Środki dydaktyczne:

- komputery z głośnikami, słuchawkami i dostępem do internetu;
- zasoby multimedialne zawarte w e-materiale;
- tablica interaktywna/tablica, pisak/kreda;

- oprogramowanie dla języka Java SE 8 (lub nowszej wersji), w tym Eclipse 4.4 (lub nowszej wersji).

Przebieg lekcji

Przed lekcją:

1. **Przygotowanie do zajęć.** Nauczyciel loguje się na platformie i udostępnia e-materiał: „Zasady programowania obiektowego w języku Java". Uczniowie zapoznają się z treściami w sekcji „Przeczytaj" w kontekście programowania.

Faza wstępna:

1. Nauczyciel wyświetla temat oraz cele zajęć, omawiając lub ustalając razem z uczniami kryteria sukcesu.
2. **Rozpoznanie wiedzy uczniów.** Nauczyciel prosi wybranego ucznia lub uczniów o przedstawienie sytuacji problemowej związanej z tematem lekcji.

Faza realizacyjna:

1. **Praca z tekstem.** Jeżeli przygotowanie uczniów do lekcji jest niewystarczające, nauczyciel prosi o indywidualne zapoznanie się z treścią zawartą w sekcji „Przeczytaj". Każdy uczestnik zajęć podczas cichego czytania wynotowuje najważniejsze kwestie poruszane w tekście.
2. **Praca z multimedium.** Nauczyciel wyświetla zawartość sekcji „Prezentacja multimedialna". Uczniowie wspólnie analizują prezentację multimedialną, w której zgodnie z paradygmatami programowania obiektowego tworzone są klasy reprezentujące figury geometryczne. Zastanawiają się, w jaki sposób można rozszerzać powstały program.
3. **Ćwiczenie umiejętności.** Uczniowie, pracując w parach, wykonują ćwiczenie nr 1 z sekcji „Sprawdź się". Nauczyciel sprawdza poprawność pisanych kodów, porównuje je i omawia wraz z uczniami. Wskazuje najbardziej efektywne rozwiązanie.

Faza podsumowująca:

1. Wybrany uczeń podsumowuje zajęcia, zwracając uwagę na nabyte umiejętności, omawia ewentualne problemy podczas rozwiązywania ćwiczeń z programowania w języku Java.
2. Nauczyciel wyświetla na tablicy temat lekcji i cele zawarte w sekcji „Wprowadzenie". W kontekście ich realizacji podsumowuje przebieg zajęć, a także wskazuje mocne i słabe strony pracy uczniów.

Praca domowa:

1. Uczniowie wykonują ćwiczenie nr 2 z sekcji „Sprawdź się".

2. Uczniowie wykonują „Zadania dla ciebie” zapisane w prezentacji multimedialnej:
   1. Utwórz klasę `Kolo` implementującą interfejs `Ksztalt`, w której zawrzesz prywatne pole typu `double` o nazwie `promien`. W klasie dodaj konstruktor, który przyjmuje wartość promienia. Pamiętaj, aby utworzyć metodę `pole()`. Skorzystaj ze stałej `PI` z biblioteki `Math`. Przetestuj utworzoną klasę przez utworzenie obiektu typu `Kolo` w tablicy `ksztalty`.
   2. Do interfejsu `Ksztalt` dodaj metodę `obwod()`. W każdej klasie implementującej ten interfejs, nadpisz jej definicję w taki sposób, aby zwracała obwód danej figury. Aby przetestować utworzoną metodę, w pętli `for` w funkcji `main()` dodaj polecenie wypisujące obwód każdej figury z tablicy `ksztalty`.

Materiały pomocnicze:

- Oficjalna dokumentacja techniczna dla języka Java SE 8 (lub nowszej wersji).
- Oficjalna dokumentacja techniczna dla oprogramowania Eclipse 4.4 (lub nowszej wersji).

Wskazówki metodyczne:

- Treści w sekcji „Przeczytaj” można wykorzystać jako podsumowanie i utrwalenie wiedzy uczniów.