Zasady programowania obiektowego w języku C++

- Wprowadzenie
- Film samouczek
- Przeczytaj
- Sprawdź się
- Dla nauczyciela

Źródło: domena publiczna.

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

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

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 Java,
- Zasady programowania obiektowego w języku Python.

Twoje cele

- Przeanalizujesz zakresy widoczności pól oraz metod w klasach zaimplementowanych w języku C++.
- Prześledzisz mechanizmy dziedziczenia oraz hermetyzacji w języku C++.
- Przygotujesz hierarchiczne struktury danych z wykorzystaniem dziedziczenia.

Film samouczek

Problem 1

Przygotuj program, który będzie symulował hierarchię pojazdów. Zapisz definicję jednej klasy nadrzędnej (`Pojazd`) oraz przynajmniej jednej klasy dziedziczącej po klasie `Pojazd` (np. klasa `Samochod`, klasa `Rower`).

Założenia algorytmu:

- Pojazd może przyspieszać, zwalniać, hamować (czyli zwalniać o ściśle określoną prędkość, na przykład o $15\ \mathrm{km/h}$) oraz podawać informację o swojej prędkości.
- Pojazdy nie mogą jeździć szybciej niż $150\ \mathrm{km/h}$.
- Pojazdy nie mogą jeździć z ujemną prędkością.

Przetestuj działanie programu dla następujących wywołań:

```
Pojazd pojazd = Pojazd();
pojazd.przyspiesz(50);
pojazd.przyspiesz(70);
pojazd.wyswietlInformacje();

Rower rower = Rower();
rower.przyspiesz(40);
rower.hamuj();
```

Specyfikacja:

Dane:

- `Pojazd` – klasa nadrzędna
- `Rower` – klasa dziedzicząca po klasie `Pojazd`

Wynik:

Na konsoli wyświetli się (w zależności od wprowadzonych danych pojazdów oraz wywołanych metod):

- aktualna prędkość,
- czy pojazd zahamował,
- czy pojazd próbuje przekroczyć dozwoloną prędkość.

Przykładowe wyjście:

```
Nie mozesz jechac szybciej niz 150 km/h.
Aktualna predkosc: 150 km/h.
Nie mozesz jechac szybciej niez 25 km/h.
Rower hamuje.
Aktualna predkosc: 20 km/h.
```

Ćwiczenie 1

```cpp
#include <iostream>
using namespace std;

int main(){

    return 0;
}
```

```
1
```

Polecenie 1

Porównaj swoje rozwiązanie z zaprezentowanym w filmie.

Film dostępny pod adresem /preview/resource/RxZ0Jl3J7DXP4

Lekcja poświęcona paradygmatom programowania obiektowego w języku C++.

Kod programu zaprezentowanego w filmie:

Plik o rozmiarze 1.32 KB w języku polskim

Przeczytaj

> **Ciekawostka**
>
> Język C++ rozszerzył możliwości używanego wcześniej języka C o paradygmaty programowania obiektowego. Zmiana ta znacznie poprawiła czytelność kodu, dzięki czemu C++ stał się jednym z najchętniej używanych języków na świecie.

Polimorfizm

Polimorfizm to inaczej wielopostaciowość. Termin ten należy rozumieć jako mechanizmy zawarte w języku programowania, które pozwalają na traktowanie tych samych danych (np. zmiennych, obiektów) w różny sposób. Aby dokładnie zrozumieć ten paradygmat, musimy na początku zaznajomić się z metodami wirtualnymi. Dopiero później będziemy w stanie zrozumieć wielopostaciowość, która w języku C++ związana jest głównie z obiektami dynamicznymi.

Metody wirtualne

Metody wirtualne to ważna część polimorfizmu i abstrakcji. Są to metody, których definicja jest wielopostaciowa — mogą być dziedziczone, nadpisywane oraz rozszerzane w klasach pochodnych.

W języku C++ deklarację metody wirtualnej poprzedzamy słowem kluczowym `virtual`. Jeśli chcemy utworzyć metodę czysto wirtualną, po deklaracji funkcji dopisujemy `=0`. Przykład:

```
1 class Zwierze {
2 public:
3     virtual void jedz() = 0;
4 };
```

Metoda zadeklarowana w linijce 3. jest metodą czysto wirtualną.

Klasa `Zwierze` jest klasą abstrakcyjną — każda klasa, która oparta jest na klasie `Zwierze`, musi nadpisać metodę `jedz()`.

Zaimplementujmy dwie dodatkowe klasy, które dziedziczą po sobie oraz klasie `Zwierze`.

```
1 #include <iostream>
```

```
2  using namespace std;
3
4  class Zwierze {
5  public:
6      virtual void jedz() = 0;
7  };
8
9  class Antylopa : public Zwierze {
10 public:
11     void jedz() override {
12         cout << "Antylopa zjada trawe" << endl;
13     }
14
15     void biegnij() {
16         cout << "Antylopa biegnie" << endl;
17     };
18 };
19
20 class Gnu : public Antylopa {
21 public:
22     void jedz() override {
23         cout << "Gnu ";
24         Antylopa::jedz();
25     }
26 };
```

Nadpisując metodę wirtualną, możemy zastosować słowo kluczowe `override`. Nie jest ono wymagane dla poprawności kodu, ale zwiększa jego czytelność. Nadpisanie metody nie może zmienić jej typu ani właściwości, a jedynie kod, który wykonuje.

Warto zauważyć, że jeśli oznaczymy metodę słowem kluczowym `virtual` w klasie bazowej, będzie ona wirtualna również we wszystkich klasach pochodnych. Przykładem jest klasa `Gnu` – zostaje w niej nadpisana wirtualna metoda z klasy nadrzędnej `Antylopa`, mimo że nie była oznaczona słowem `virtual`. Dzieje się tak, ponieważ klasa `Antylopa` dziedziczy po klasie `Zwierzę`, gdzie metoda `jedz()` jest oznaczona jako wirtualna.

Zwróć uwagę na odwołanie do metody `jedz()` z klasy bazowej w linijce 24. Możemy w trakcie realizacji metody pochodnej wykonać kod, który zdefiniowany jest w klasie bazowej. Dzięki temu nadpisywanie metody nie ogranicza się do zdefiniowania funkcji na nowo, ale umożliwia także jej rozszerzanie.

W linijce 15. została zadeklarowana metoda `biegnij()`. Nie jest ona metodą wirtualną, więc nie można jej nadpisać w klasie `Gnu`. Jeżeli spróbujemy to zrobić, kompilator potraktuje to jako błąd.

Spróbujmy zatem nadpisać metodę `biegnij()` z klasy `Antylopa` w klasie `Gnu`. Jednak na razie nie zmieniajmy kodu klasy `Antylopa`.

```
1  #include <iostream>
2  using namespace std;
3
4  class Zwierze {
5  public:
6      virtual void jedz() = 0;
7  };
8
9  class Antylopa : public Zwierze {
10 public:
11     void jedz() override {
12         cout << "Antylopa zjada trawe" << endl;
13     }
14
15     void biegnij() {
16         cout << "Antylopa biegnie" << endl;
17     };
18 };
19
20 class Gnu : public Antylopa {
21 public:
22     void jedz() override {
23         cout << "Gnu ";
24         Antylopa::jedz();
25     }
26
27     // błędne nadpisanie metody niewirtualnej
28     void biegnij() override {
29         cout << "Gnu ";
30         Antylopa::biegnij();
31     }
32 };
```

Podczas próby kompilacji przedstawionego kodu powinien wystąpić błąd. Aby nadpisać metodę `biegnij()` z klasy `Antylopa`, poprzedzamy jej definicję słowem kluczowym `virtual`. Przedstawiony poniżej kod powinien skompilować się poprawnie.

```
#include <iostream>
using namespace std;

class Zwierze {
public:
    virtual void jedz() = 0;
};

class Antylopa : public Zwierze {
public:
    void jedz() override {
        cout << "Antylopa zjada trawe" << endl;
    }

    virtual void biegnij() {
        cout << "Antylopa biegnie" << endl;
    };
};

class Gnu : public Antylopa {
public:
    void jedz() override {
        cout << "Gnu ";
        Antylopa::jedz();
    }

    void biegnij() override {
        cout << "Gnu biegnie";
    }
};
```

W celu przetestowania stworzonych klas, możemy napisać funkcję główną programu, w której utworzymy obiekty klasy `Antylopa` oraz `Gnu`. Następnie wywołamy na nich metody `jedz()` oraz `biegnij()`.

```
#include <iostream>
```

```cpp
using namespace std;

class Zwierze {
public:
    virtual void jedz() = 0;
};

class Antylopa : public Zwierze {
public:
    void jedz() override {
        cout << "Antylopa zjada trawe" << endl;
    }

    virtual void biegnij() {
        cout << "Antylopa biegnie" << endl;
    };
};

class Gnu : public Antylopa {
public:
    void jedz() override {
        cout << "Gnu ";
        Antylopa::jedz();
    }

    void biegnij() override {
        cout << "Gnu biegnie";
    }
};

int main() {
    Antylopa antylopa;
    Gnu gnu;

    antylopa.jedz();
    antylopa.biegnij();
    gnu.jedz();
    gnu.biegnij();

    return 0;
}
```

Przedstawiony program powinien wypisać w konsoli:

```
Antylopa zjada trawe
Antylopa biegnie
Gnu Antylopa zjada trawe
Gnu biegnie
```

Polimorfizm dynamiczny

W języku C++ możemy definiować tablice obiektów dynamicznych tego samego typu bazowego, jednak różniących się typem pochodnym. Warto w tym miejscu omówić dwa przykłady.

Kod pierwszego przykładu wygląda następująco:

```
#include <iostream>
using namespace std;

class Zwierze {
public:
    virtual void jedz() = 0;
};

class Kot : public Zwierze {
public:
    void jedz() override {
        cout << "Kot zjada mysz" << endl;
    }

    void biegnij() {
        cout << "Kot biegnie" << endl;
    };
};

class Antylopa : public Zwierze {
public:
    void jedz() override {
        cout << "Antylopa zjada trawe" << endl;
    }

    void biegnij() {
```

```
        cout << "Antylopa biegnie" << endl;
    };
};

class Gnu : public Antylopa {
public:
    void jedz() override {
        cout << "Gnu ";
        Antylopa::jedz();
    }
};

int main() {
    /* przykład pierwszy */
    Zwierze* tablica_zwierzat[3] = {
        new Antylopa(),
        new Gnu(),
        new Kot()
    };

    for (int i = 0; i < 3; ++i) {
        (*tablica_zwierzat[i]).jedz();
        // element tablicy jest traktowany jako Zwierze
        // (nie jako Antylopa)
        // - brak metody biegnij w klasie Zwierze
        // (*tablica_zwierzat[i]).biegnij();
    }

    // każdy obiekt dynamiczny musi zostać usunięty
    for (int i = 0; i < 3; ++i) {
        delete tablica_zwierzat[i];
    }

    return 0;
}
```

W przedstawionym przykładzie została dodana prosta klasa `Kot` dziedzicząca po klasie `Zwierze`.

W funkcji głównej programu deklarujemy trójelementową tablicę wskaźników na obiekty dynamiczne typu `Zwierze`. Dzięki wbudowanemu w język C++ polimorfizmowi możemy

taką tablicę wypełnić wskaźnikami na różne obiekty dynamiczne, których klasy dziedziczą po typie `Zwierze`. Jak widać na przykładzie, w deklaracji tablicy tworzymy nowe obiekty typu `Antylopa`, `Gnu` oraz `Kot`.

W linijce 48. dla każdego obiektu, na który wskazują wskaźniki z tablicy `tablica_zwierząt`, wywołujemy metodę `jedz()`. Zastanów się, jaki będzie wynik programu. Przetestuj jego działanie.

```
Antylopa zjada trawe
Gnu Antylopa zjada trawe
Kot zjada mysz
```

Zwróć uwagę na to, że wszystkie obiekty, na które wskazują elementy tablicy `tablica_zwierząt`, traktowane są jako obiekty klasy `Zwierze`. Oznacza to, że metody, które chcemy wywołać, muszą być widoczne w klasie bazowej `Zwierze`. Sprawdź, co się stanie, gdy odkomentujemy linijkę nr 52 i spróbujemy wywołać metodę, która nie występuje w klasie `Zwierze`. Mimo że każdy obiekt posiada metodę `biegnij()`, wystąpi błąd, gdyż nie jest ona widoczna z poziomu klasy `Zwierze`.

Przykład drugi:

```
#include <iostream>
using namespace std;

class Zwierze {
public:
    virtual void jedz() = 0;
};

class Kot : public Zwierze {
public:
    void jedz() override {
        cout << "Kot zjada mysz" << endl;
    }

    void biegnij() {
        cout << "Kot biegnie" << endl;
    };
};

class Antylopa : public Zwierze {
```

```
public:
    void jedz() override {
        cout << "Antylopa zjada trawe" << endl;
    }

    void biegnij() {
        cout << "Antylopa biegnie" << endl;
    };
};

class Gnu : public Antylopa {
public:
    void jedz() override {
        cout << "Gnu ";
        Antylopa::jedz();
    }
};

int main() {
    /* przykład drugi */
    Antylopa* tablica_antylop[3] = {
        new Antylopa(),
        new Gnu(),
        new Antylopa()
    };

    for (int i = 0; i < 3; ++i) {
        (*tablica_antylop[i]).jedz();
        (*tablica_antylop[i]).biegnij();
    }

    // każdy obiekt dynamiczny musi zostać usunięty
    for (int i = 0; i < 3; ++i) {
        delete tablica_antylop[i];
    }

    return 0;
}
```

W tym przykładzie zmieniamy typ tablicy na `Antylopa` i tworzymy w niej dwa obiekty klasy `Antylopa` oraz jeden obiekt klasy `Gnu`. Zwróć uwagę, że każdy obiekt typu `Gnu` jest

również obiektem typu `Antylopa`.

Uruchom przedstawiony kod i sprawdź, jakie otrzymasz wyniki. Zauważ, że w pętli w linijce 47. można wywołać obie metody `jedz()` oraz `biegnij()`, ponieważ wszystkie użyte metody są widoczne w klasie `Antylopa`. Ostatecznie wynik prezentuje się następująco:

```
Antylopa zjada trawe
Antylopa biegnie
Gnu Antylopa zjada trawe
Antylopa biegnie
Antylopa zjada trawe
Antylopa biegnie
```

Hermetyzacja w języku C++

Hermetyzacja to inaczej ukrywanie implementacji. Paradygmat ten odnosi się do danych oraz funkcji, które je modyfikują. Dzięki hermetyzacji możemy ukrywać dostęp do danych oraz metod, kontrolując tym samym interakcję między obiektami.

Z mechanizmem hermetyzacji powiązane są trzy podstawowe modyfikatory dostępu:

- prywatny (`private`) – zmienna może być modyfikowana i odczytywana tylko w klasie, w której została zadeklarowana,
- chroniony (`protected`) – zmienna może być modyfikowana i odczytywana w klasie, w której została zadeklarowana oraz w klasach pochodnych od klasy, w której została zadeklarowana,
- publiczny (`public`) – zmienna może być modyfikowana i odczytywana w dowolnym miejscu programu, w którym mamy dostęp do obiektu.

Dzięki nim mamy możliwość wydzielenia w strukturze klasy sekcji o różnym dostępie do pól oraz metod.

Aby zweryfikować działania modyfikatorów dostępu, umieśćmy w klasie bazowej `Zwierze` trzy zmienne: publiczną, chronioną i prywatną:

```
class Zwierze {
private:
    int zmiennaA = 1;
protected:
    int zmiennaB = 2;
public:
    int zmiennaC = 3;
```

```
    virtual void jedz() = 0;
};
```

Mamy poprawną deklarację trzech pól w klasie `Zwierze`. Napiszmy dwie metody, po jednej w klasie `Zwierze` i `Antylopa`, w których zbadamy dostęp do tych trzech zmiennych. Spróbujemy uzyskać dostęp do zmiennych oraz wypisać je na ekran. Jednocześnie w funkcji `main` umieścimy kod, w którym również spróbujemy sprawdzić dostęp do zawartości zmiennych. Kod prezentuje się następująco:

```
#include <iostream>
using namespace std;

class Zwierze {
private:
    int zmiennaA = 1;
protected:
    int zmiennaB = 2;
public:
    int zmiennaC = 3;
    virtual void jedz() = 0;

    void metoda1() {
        cout << zmiennaA << endl;
        cout << zmiennaB << endl;
        cout << zmiennaC << endl;
    }
};

class Antylopa : public Zwierze {
public:
    void jedz() override {
        cout << "Antylopa zjada trawe" << endl;
    }

    void biegnij() {
        cout << "Antylopa biegnie" << endl;
    };

    void metoda2() {
        cout << zmiennaA << endl;
        cout << zmiennaB << endl;
```

```
33          cout << zmiennaC << endl;
34      }
35  };
36
37  int main() {
38      // instancja klasy Antylopa
39      Antylopa antylopa;
40
41      // dostęp z metody z klasy bazowej
42      antylopa.metoda1();
43
44      // dostęp z metody z klasy pochodnej
45      antylopa.metoda2();
46
47      // dostęp funkcji głównej programu
48      cout << antylopa.zmiennaA << endl;
49      cout << antylopa.zmiennaB << endl;
50      cout << antylopa.zmiennaC << endl;
51
52      return 0;
53  }
```

Możemy przypuszczać, że program nie uruchomi się, ponieważ w wielu przypadkach nie mamy dostępu do zmiennych, których wartość próbujemy poznać. Kluczowe są dla nas linijki nr 14, 15, 16 oraz nr 31, 32, 33, a także nr 48, 49, 50. Zastanów się, które z nich zostaną uznane przez kompilator z błędne. Oczywiście będą to linijki nr 31, 48 i 49.

Oto kod, w którym komentarzami oznaczono błędne wiersze. Dodano również wyjaśnienie, na czym polegał błąd:

```
1  #include <iostream>
2  using namespace std;
3
4  class Zwierze {
5  private:
6      int zmiennaA = 1;
7  protected:
8      int zmiennaB = 2;
9  public:
10     int zmiennaC = 3;
11     virtual void jedz() = 0;
```

```
12
13      void metoda1() {
14          cout << zmiennaA << endl;
15          cout << zmiennaB << endl;
16          cout << zmiennaC << endl;
17      }
18  };
19
20  class Antylopa : public Zwierze {
21  public:
22      void jedz() override {
23          cout << "Antylopa zjada trawe" << endl;
24      }
25
26      void biegnij() {
27          cout << "Antylopa biegnie" << endl;
28      };
29
30      void metoda2() {
31          // dostęp do prywatnej zmiennej klasy bazowej
32          // nie jest możliwy z klasy pochodnej
33          // cout << zmiennaA << endl;
34          cout << zmiennaB << endl;
35          cout << zmiennaC << endl;
36      }
37  };
38
39  int main() {
40      // instancja klasy Antylopa
41      Antylopa antylopa;
42
43      // dostęp z metody z klasy bazowej
44      antylopa.metoda1();
45
46      // dostęp z metody z klasy pochodnej
47      antylopa.metoda2();
48
49      // dostęp funkcji głównej programu
50      cout << antylopa.zmiennaC << endl;
51      // dostęp do zmiennych prywatnych oraz chronionych
52      // nie jest możliwy z poziomu funkcji głównej programu
53      // cout << antylopa.zmiennaA << endl;
```

```
54      // cout << antylopa.zmiennaB << endl;
55
56      return 0;
57  }
```

Słownik

> **metoda czysto wirtualna**
>
> możliwa do zadeklarowania tylko w klasie abstrakcyjnej metoda, która musi zostać zaimplementowana w klasach dziedziczących po danej klasie; w klasie bazowej nie posiada implementacji, a jedynie deklarację
>
> **modyfikatory dostępu**
>
> słowa kluczowe, za pomocą których definiujemy zakres widoczności elementów takich jak klasy, pola oraz metody

Sprawdź się

Pokaż ćwiczenia: 

Ćwiczenie 1

Zapoznaj się z przedstawionym programem, a następnie uzupełnij klasę `Komputer` w taki sposób, aby program mógł wyświetlić poprawnie komunikat informujący o działaniu. Napis ma być wypisywany przez obiekt klasy `Komputer` za pomocą metody wirtualnej `dzialanie()`. Przetestuj działanie programu dla komunikatu: `"Przeprowadzam obliczenia!"`.

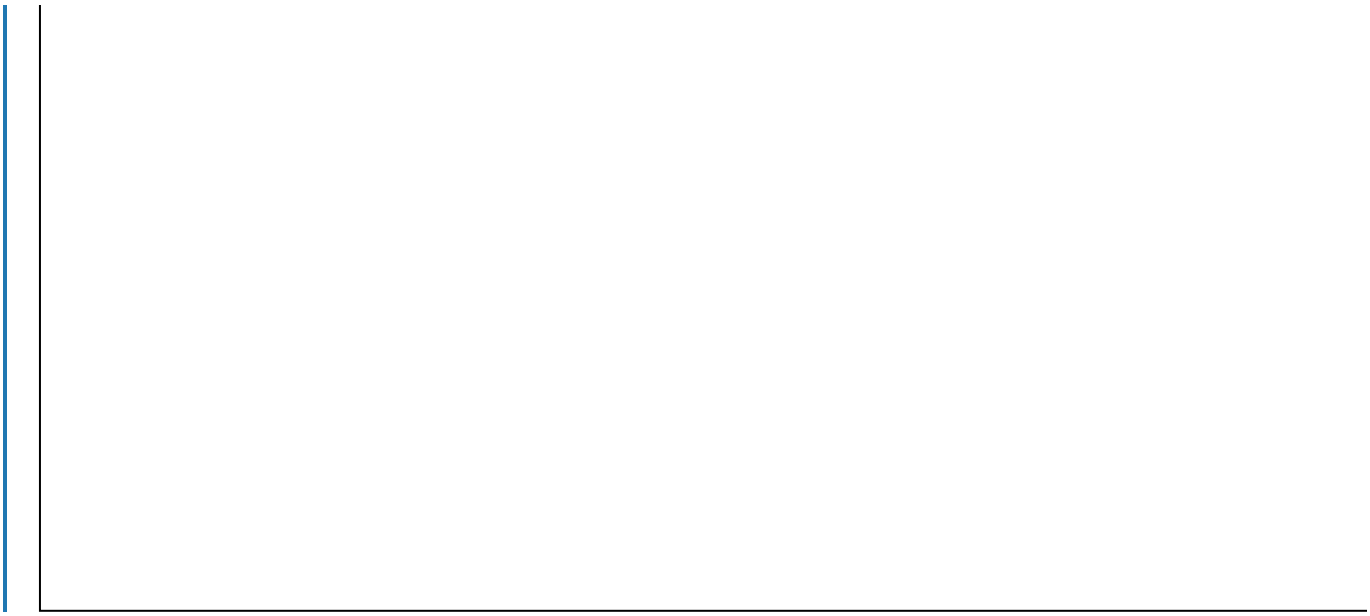
Specyfikacja:

Wynik:

Program wypisuje komunikat informujący o działaniu klasy `Komputer`.

Twoje zadania

1. Program wypisuje komunikat „`Przeprowadzam obliczenia!`”.



Ćwiczenie 2

Napisz program, w którym stworzysz klasę abstrakcyjną o nazwie `Zwierze`, zawierającą wirtualną metodę `daj_glos()`. Następnie stwórz w programie klasę `Kot`, która będzie dziedziczyła publicznie z klasy `Zwierze`. Klasa `Kot` ma posiadać implementację wirtualnej metody `daj_glos()`. W programie ma być utworzony obiekt `Kot`, który wywołuje metodę `daj_glos()` i w ten sposób wypisuje odpowiednią zawartość. Przetestuj działanie programu dla komunikatu „`Miau miau!`".

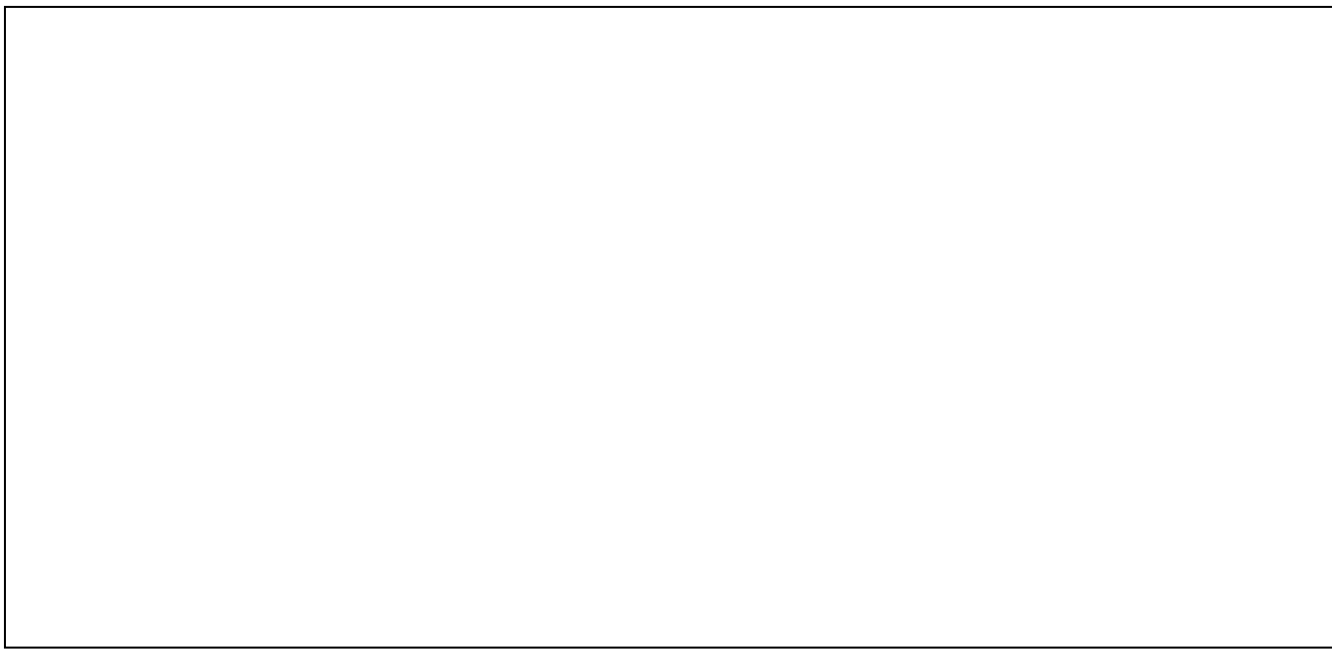
Specyfikacja:

Wynik:

Program wypisuje komunikat zadeklarowany w odpowiedniej metodzie klasy `Kot`.

Twoje zadania

1. Program wypisuje komunikat „`Miau miau!`".



Ćwiczenie 3

Uzupełnij program, w którym zawarto klasę `Gracz` oraz dziedziczącą po niej klasę `Biegacz`. Do klasy `Biegacz` dodaj prywatne pole typu `int` o nazwie `kondycja`. Nadpisz metodę `wykonaj_ruch()` w taki sposób, aby modyfikowała ona współrzędne biegacza zgodnie z podanym kierunkiem (0 – góra, 1 – prawo, 2 – dół, 3 – lewo). Współrzędne mają być zgodne z kartezjańskim układem współrzędnych. Każdy ruch powoduje u biegacza stratę kondycji (góra – strata 2 punktów, prawo – strata 1 punktu, dół – zysk 1 punktu, lewo – strata 1 punktu). W momencie, gdy kondycja biegacza nie pozwala na wykonanie danego ruchu, program powinien wypisać na ekran komunikat „`Nie mam sily`". Przetestuj swój program dla danych zawartych w tablicy `trasa`. Jeśli nie może on wykonać jakiegoś ruchu, pomija go – nie wykonujemy ruchu ponownie. Na koniec programu powinny zostać wypisane współrzędne gracza.

Specyfikacja:

Dane:

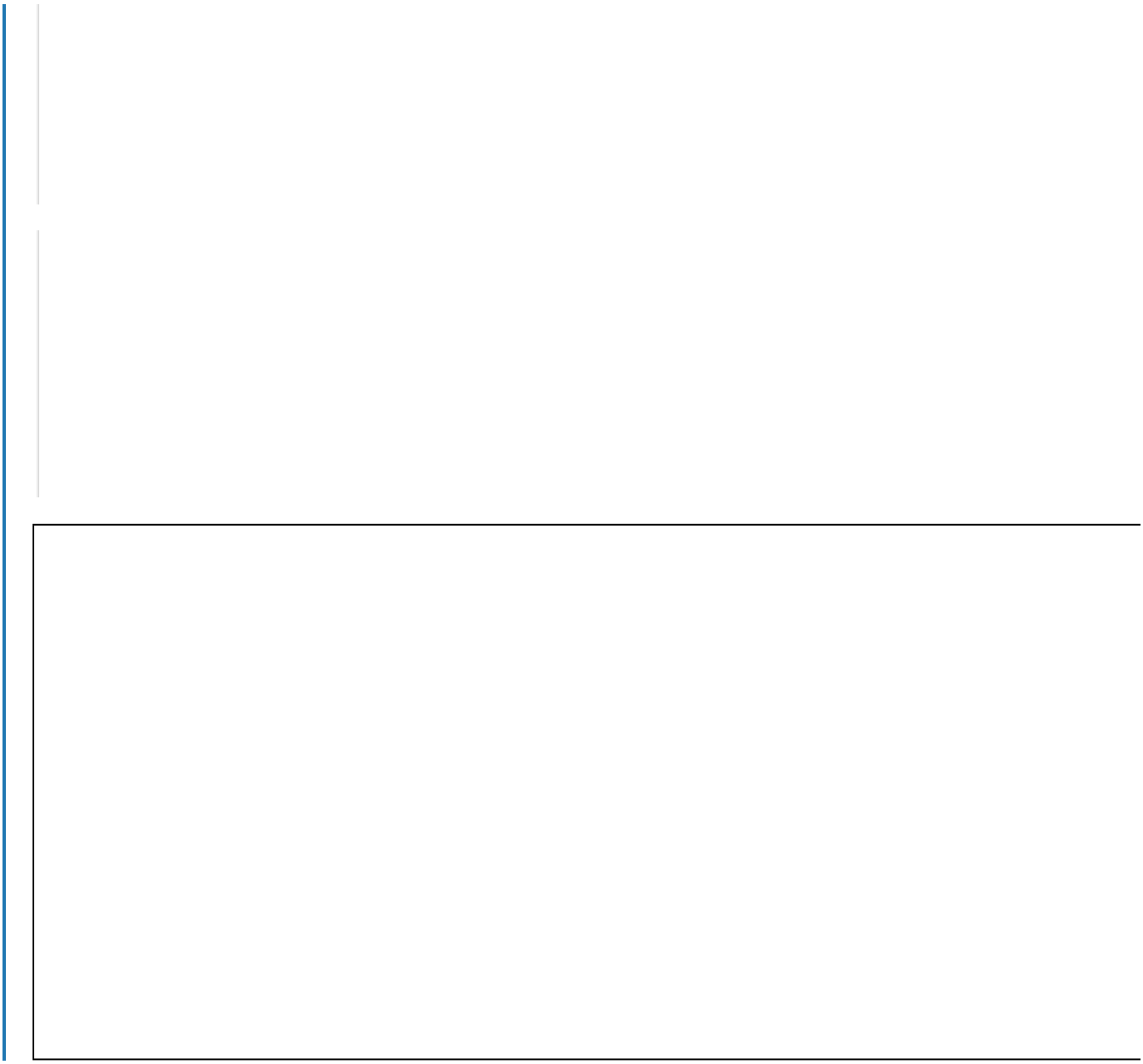
- `trasa` – tablica liczb naturalnych, określających kolejne ruchy gracza

Wynik:

Program wypisuje komunikaty informujące o braku kondycji obiektu klasy `Gracz` oraz jego współrzędne końcowe.

Twoje zadania

1. Program wypisuje komunikaty informujące o braku kondycji obiektu klasy `Gracz` oraz jego współrzędne końcowe.



Dla nauczyciela

Autor: Maurycy Gast
Przedmiot: Informatyka

Temat: Zasady programowania obiektowego w języku C++

Grupa docelowa:

Szkoła ponadpodstawowa, liceum ogólnokształcące, technikum, zakres 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.

Treści nauczania – wymagania szczegółowe

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

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):

- Przeanalizujesz zakresy widoczności pól oraz metod w klasach zaimplementowanych w języku C++.
- Prześledzisz mechanizmy dziedziczenia oraz hermetyzacji w języku C++.

- Przygotujesz hierarchiczne struktury danych z wykorzystaniem dziedziczenia.

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 C++, w tym kompilator GCC/G++ 4.5 (lub nowszej wersji) i Code::Blocks 16.01 (lub nowszej wersji), Orwell Dev-C++ 5.11 (lub nowszej wersji) lub Microsoft Visual Studio.

Przebieg lekcji

Przed lekcją:

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

Faza wstępna:

1. Nauczyciel wprowadza uczniów szczegółowo w temat lekcji i jej cele. Może posłużyć się wyświetloną na tablicy zawartością sekcji „Wprowadzenie".
2. **Rozpoznanie wiedzy uczniów.** Uczniowie tworzą pytania dotyczące tematu zajęć, na które odpowiedzą w trakcie lekcji.

Faza realizacyjna:

1. **Praca z tekstem.** Nauczyciel ocenia, na podstawie informacji na platformie, stan przygotowania uczniów do zajęć. Jeśli jest ono niewystarczające prosi o ciche zapoznanie się z treścią w sekcji „Przeczytaj".
2. **Praca z multimedium.** Nauczyciel wyświetla zawartość sekcji „Film samouczek". Uczniowie zapoznają się z treścią Polecenia 1. W parach opracowują rozwiązanie, a następnie porównują je z zaprezentowanym w filmie.
3. **Ćwiczenie umiejętności.** Prowadzący zapowiada uczniom, że będą rozwiązywać ćwiczenie nr 1 z sekcji „Sprawdź się". Każdy z uczniów robi to samodzielnie. Po ustalonym czasie następuje porównanie napisanych kodów podczas wspólnego omówienia rozwiązań.
4. Liga zadaniowa – uczniowie pracując w parach, wykonują ćwiczenie nr 2 z sekcji „Sprawdź się", a następnie dzielą się swoimi wynikami przez porównywanie napisanego kodu z inną grupą, która również zakończyła zadanie.

Faza podsumowująca:

1. Wybrany uczeń podsumowuje zajęcia, zwracając uwagę na nabyte umiejętności, omawia ewentualne problemy podczas rozwiązania ćwiczeń z programowania w języku C++.

Praca domowa:

1. Uczniowie wykonują ćwiczenie nr 3 z sekcji „Sprawdź się".

Materiały pomocnicze:

- Oficjalna dokumentacja techniczna dla języka C++.
- Oficjalna dokumentacja techniczna dla kompilatora GCC/G++ 4.5 (lub nowszej wersji).
- Oficjalna dokumentacja techniczna dla oprogramowania Code::Blocks 16.01 (lub nowszej wersji), Orwell Dev-C++ 5.11 (lub nowszej wersji) lub Microsoft Visual Studio.

Wskazówki metodyczne:

- Treści w sekcji „Film samouczek" można wykorzystać jako podsumowanie i utrwalenie wiedzy uczniów.