# Programowanie obiektowe – projekt, etap VI

- Wprowadzenie
- Przeczytaj
- Animacja
- Sprawdź się
- Dla nauczyciela


Źródło: Englishsquare.pl sp. z o.o., licencja: CC BY-SA 3.0.

Po wielu przygotowaniach w końcu nadszedł ten czas – napiszesz pierwszą linijkę kodu swojego projektu. Zaczniesz od implementacji dwóch wątków. Pierwszy z nich będzie odpowiadał za rozgrywkę, a drugi za sterowanie.

Z poszczególnymi etapami powstawania projektu możesz zapoznać się w pozostałych e-materiałach z serii:

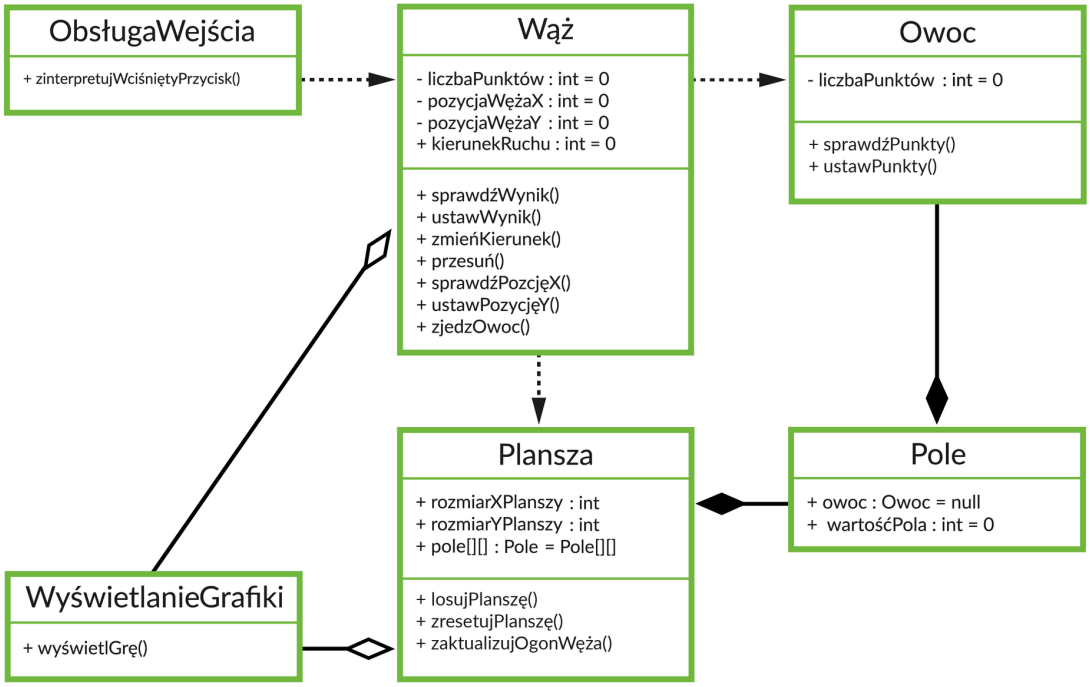
- Programowanie obiektowe – projekt, etap I,
- Programowanie obiektowe – projekt, etap II,
- Programowanie obiektowe – projekt, etap III,
- Programowanie obiektowe – projekt, etap IV,
- Programowanie obiektowe – projekt, etap V,
- Programowanie obiektowe – projekt, etap VII,
- Programowanie obiektowe – projekt, etap VIII,
- Programowanie obiektowe – projekt, etap IX.

> **Twoje cele**
>
> - Zaimplementujesz wątki w praktyce.
> - Stworzysz pierwsze klasy na podstawie diagramu.
> - Napiszesz proste sterowanie gry.

# Przeczytaj

## Wstęp

Zacznijmy od przypomnienia diagramu klas:




Źródło: Contentplus.pl sp. z o.o., licencja: CC BY-SA 4.0.

W tym e-materiale skupimy się na implementacji wątków oraz sterowania, dlatego istotne są dla nas dwie klasy: `ObsługaWejścia` oraz `WyświetlanieGrafiki`. Jednak, żebyśmy mogli je zaimplementować, należy napisać – co najmniej – pozostałe klasy.

## Wielowątkowość i sterowanie w języku Java

Zaczynamy od zadeklarowania istnienia wszystkich klas.

Wszystkie przedstawione klasy znajdują się w różnych plikach.

```
1 public class Main {
2
3     public static void main(String[] args) {
4
5     }
6 }
```

```
public class ObslugaWejscia {

}
```

```
public class WyswietlanieGrafiki implements Runnable{
    Waz waz;
    Plansza plansza;
    @Override
    public void run() {

    }
}
```

W klasie `WyswietlanieGrafiki` zaimplementujemy interfejs `Runnable`, aby potem uruchomić wyświetlanie jako osobny wątek.

```
public class Waz {
    private int liczbaPunktow = 0;
    private int pozycjaWezaX = 0;
    private int pozycjaWezaY = 0;
    private int kierunekRuchu = 0;

}
```

```
public class Plansza {
    public int rozmiarXPlanszy;
    public int rozmiarYPlanszy;
    public Pole pole[][];

}
```

```
public class Owoc {
    private int liczbaPunktow = 0;
}
```

```
public class Pole {
    Owoc owoc;
    int wartoscPola = 0;
```

```
}
```

Zwróćmy uwagę, że właśnie zaimplementowaliśmy niektóre zależności i wszystkie zmienne, które znajdowały się na grafie. Nie zaimplementowaliśmy jednak zależności, w których – chwilowo – korzystamy z innych klas.

Spróbujemy zacząć od zaimplementowania klasy `WyswietlanieGrafiki`.

```
public class WyswietlanieGrafiki implements Runnable{
    Waz waz;
    Plansza plansza;
    @Override
    public void run() {
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        wyswietlGre();
    }
    WyswietlanieGrafiki(Waz waz, Plansza plansza){
        this.waz = waz;
        this.plansza = plansza;
    }
    public void wyswietlGre(){
        for(int i = 0; i<plansza.rozmiarYPlanszy; i++){
            for(int j = 0; j<plansza.rozmiarXPlanszy; j++){
                if(plansza.pole[i][j].wartoscPola == 0){
                    System.out.print("+  ");
                }else{
                    System.out.print("x  ");
                }
            }
            System.out.println("\n");
        }
    }
}
```

Metoda `run()` została nadpisana przez naszą wersję, w której co 100 milisekund czyścimy ekran, a następnie rysujemy za pomocą `wyswietlGre()` obecny stan gry. Na ten moment

zaimplementowaliśmy wyświetlanie samej planszy – do węża wrócimy potem. Jako znak `+` wyświetlamy pola puste, a jako `x` pola, na których jest w tym momencie „coś" bliżej nieokreślonego.

Rozmiar mapy, zgodnie z wymaganiami funkcjonalnymi, ma być docelowo dostępny z konfigurowalnych plików gry, jednak nie możemy zaimplementować wszystkiego jednocześnie. Stworzymy więc zmienną, której wartość na razie ustawimy ręcznie z poziomu kodu. Postępujemy tak, ponieważ najpierw chcemy dokonać implementacji – żeby mapa zaczęła się generować w jakiejkolwiek formie.

Istotną wadą kodu jest to, że dany wątek uruchamia się tylko raz. Do samego wyświetlania wrócimy na późniejszym etapie. W międzyczasie kod ten ulegnie znacznej modyfikacji.

Sama inicjalizacja `Planszy` wyglądałaby w następujący sposób:

```
public class Plansza {
    public int rozmiarXPlanszy;
    public int rozmiarYPlanszy;
    public Pole pole[][];
    Plansza(int rozmiar){
        rozmiarXPlanszy = rozmiar;
        rozmiarYPlanszy = rozmiar;
        pole = new Pole[rozmiar][rozmiar];
        for(int i = 0; i<rozmiar; i++){
            for(int j = 0; j<rozmiar; j++){
                pole[i][j] = new Pole();
            }
        }
    }
}
```

Czas na zaimplementowanie klasy `Pole`.

```
public class Pole {
    Owoc owoc;
    int wartoscPola = 0;
    Pole(boolean czyZOwocem){
        if(czyZOwocem){
            owoc = new Owoc();
        }
    }
```

```
}
```

Klasa `Owoc` nie jest złożona, udało się ją zaimplementować już wcześniej.

Pojawia się jednak pewna nieścisłość – wcześniej w konstruktorze klasy `Plansza` używaliśmy bezargumentowych konstruktorów klasy `Pole`. Edytujemy więc jedną z dwóch klas w taki sposób, żeby zachować kompatybilność z diagramem klas.

W klasie `Plansza` ma się ostatecznie znaleźć metoda `losujPlansze()`, dlatego też w niej będziemy do konkretnych pól dodawać obiekty klasy `Owoc`. Wrócimy do tego później, jednak najpierw poprawimy klasę `Pole`.

```
public class Pole {
    Owoc owoc;
    int wartoscPola = 0;
    Pole(){
    }
}
```

W celu przetestowania, czy w grze dobrze działa wyświetlanie, wpierw implementujemy klasę `Wąż`, której instancję będziemy powoływać w klasie `Main()`.

```
public class Waz {
    private int liczbaPunktow = 0;
    private int pozycjaWezaX = 0;
    private int pozycjaWezaY = 0;
    private int kierunekRuchu = 0;
    Waz(int pozycjaWezaX, int pozycjaWezaY, int kierunekRuchu){
        this.pozycjaWezaX = pozycjaWezaX;
        this.pozycjaWezaY = pozycjaWezaY;
        this.kierunekRuchu = kierunekRuchu;
    }
}
```

Zrobienie tego w ten sposób, pozwala łatwo zmieniać miejsce na planszy, w którym wąż ma swoją pozycję startową oraz startowy kierunek ruchu.

Klasa `Main()` przyjmie postać:

```
public class Main {
```

```
2
3       public static void main(String[] args) {
4           WyswietlanieGrafiki watekGrafiki = new WyswietlanieGrafik
5           Thread watek = new Thread(watekGrafiki);
6           watek.run();
7       }
8   }
```

Po uruchomieniu program wyświetli się następujący obraz:

```
1  +  +  +  +  +
2
3  +  +  +  +  +
4
5  +  +  +  +  +
6
7  +  +  +  +  +
8
9  +  +  +  +  +
```

Teraz dochodzimy do problematycznej części – sterowania. W językach C++ i Python możliwe jest przechwytywanie klawiszy z konsoli za pomocą funkcji lub metody `getch()`, która nie ma swojego odpowiednika w języku Java.

Istnieje natomiast opcja tworzenia interfejsów graficznych, które mogą przechwytywać pojedyncze kliknięcia. Zależy nam na interpretacji pojedynczych kliknięć, ponieważ gra byłaby dużo mniej użytkowa, gdybyśmy co chwilę potwierdzali klawiszem Enter decyzję.

Zaimplementujmy więc interfejs graficzny. W tym celu skorzystamy z biblioteki Swing, dostępnej w języku Java.

```
1  import javax.swing.*;
2
3  public class WyswietlanieGrafiki implements Runnable{
4      Waz waz;
5      Plansza plansza;
6      private JFrame okienkoGraficzne;
7      private JTextArea grafikaTekstowa;
8      @Override
9      public void run() {
```

```
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        wyswietlGre();
    }
    WyswietlanieGrafiki(Waz waz, Plansza plansza){
        this.waz = waz;
        this.plansza = plansza;
        this.okienkoGraficzne = new JFrame();
        this.grafikaTekstowa = new JTextArea();
        grafikaTekstowa.setSize(500, 500);
        grafikaTekstowa.setEditable(false);
        okienkoGraficzne.setSize(500, 500);
        okienkoGraficzne.add(grafikaTekstowa);
        okienkoGraficzne.setVisible(true);
    }

    ....
}
```

Przypomnienie: `JFrame` to samo okno, a `JTextArea` to pole tekstowe, które musi być osadzone w oknie. Ustawiamy rozmiar obu obiektów i wyłączamy możliwość edycji okienka tekstowego. Następnie dodajemy okienko tekstowe do obiektu klasy `JFrame`, a potem wyświetlamy ten obiekt.

Do zmiennych obiektu `WyswietlanieGrafiki()` dodaliśmy również obiekty utworzone w konstruktorze. Zrobiliśmy to, ponieważ później będziemy potrzebowali tych informacji, do swobodnej modyfikacji zawartości okna.

Wyświetlanie w nowej postaci, która wypisuje stan planszy do zdefiniowanego przed chwilą okienka tekstowego, wyglądałoby w następujący sposób:

```
public void wyswietlGre(){
    for(int i = 0; i<plansza.rozmiarYPlanszy; i++){
       for(int j = 0; j<plansza.rozmiarXPlanszy; j++){
          if(plansza.pole[i][j].wartoscPola == 0){
              grafikaTekstowa.append("+  ");
          }else{
              grafikaTekstowa.append("x  ");
```

```
            }
        }
     grafikaTekstowa.append("\n ");
     }
}
```

Metoda `append()` dopisuje tekst do okienka analogicznie do metody `System.out.print()`.

Z racji tego, że chcemy, aby okienko odświeżało się co jakiś czas automatycznie, to potrzebujemy również sposobu na czyszczenie go. Zrobimy to w następujący sposób:

```
grafikaTekstowa.setText("");
```

Zapisany przed chwilą fragment kodu ustawi zawartość okienka na pusty tekst, efektywnie czyszcząc je.

Spróbujmy teraz zaimplementować sterowanie. W tym celu skorzystamy z interfejsu `KeyListener`, za pomocą którego zaimplementujemy wykrywanie kliknięcia poszczególnych klawiszy. Klasa `ObslugaWejscia()` będzie teraz wyglądać w następujący sposób:

```
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;

public class ObslugaWejscia implements KeyListener {
    @Override
    public void keyTyped(KeyEvent e) {

    }
    @Override
    public void keyPressed(KeyEvent e) {
        char znak = e.getKeyChar();
        if(znak == 'a') System.out.println("Naciśnięto a");
        else if(znak == 'd') System.out.println("Naciśnięto d");
        else if(znak == ' ') System.out.println("Naciśnięto spacj
    }

    @Override
    public void keyReleased(KeyEvent e) {

```

```
19
20      }
21  }
```

Obiekt klasy `KeyListener` , aby mógł działać, musi zostać dołączony do `JFrame'a` lub innego kontenera. Postępujemy więc podobnie jak w przypadku dodawania `JTextArea`. W tym celu najprościej będzie dodać obiekt klasy `KeyListener` do konstruktora klasy `WyswietlanieGrafiki` – w ten sposób przyjmie on następującą postać:

```
1  WyswietlanieGrafiki(Waz waz, Plansza plansza, KeyListener obsluga
2      this.waz = waz;
3      this.plansza = plansza;
4      this.okienkoGraficzne = new JFrame();
5      this.grafikaTekstowa = new JTextArea();
6      grafikaTekstowa.setSize(500, 500);
7      grafikaTekstowa.setEditable(false);
8      okienkoGraficzne.setSize(500, 500);
9      okienkoGraficzne.add(grafikaTekstowa);
10     grafikaTekstowa.addKeyListener(obslugaWejscia);
11     okienkoGraficzne.setVisible(true);
12 }
```

W ten sposób, gdy wybierzemy okienko, a potem przyciski:
`a`, `d`, `spacja` , `shift` + `a`, `shift` + `d`, to otrzymamy następujący wydruk w konsoli:

```
1  Naciśnięto a
2  Naciśnięto d
3  Naciśnięto spacje
```

Możemy zatem wywnioskować, że metoda `getKeyChar()` jest case sensitive.

W tym momencie klasa `main` przyjmie następującą postać:

```
1  public class Main {
2
3      public static void main(String[] args) {
4          ObslugaWejscia wejscie = new ObslugaWejscia();
5          WyswietlanieGrafiki grafika = new WyswietlanieGrafiki(new
6          Thread watekGrafiki = new Thread(grafika);
```

```
        watekGrafiki.start();
    }
}
```

Następne mechaniki rozgrywki zaimplementujemy w kolejnych e-materiałach.

Dotychczas napisany kod źródłowy w języku Java można pobrać, korzystając z linku:

Kod źródłowy Java
Plik o rozmiarze 2.27 KB w języku polskim

## Wielowątkowość i sterowanie w języku C++

Zacznijmy od napisania samych nagłówków klas, wraz ze zmiennymi o odpowiednich specyfikatorach dostępu.

```
class ObslugaWejscia
{
public:
    void zinterpretujWcisnietyPrzycisk()
    {

    }
};

class Waz
{
private:
    int liczbaPunktow = 0;
    int pozycjaWezaX = 0;
    int pozycjaWezaY = 0;
public:
    int kierunekRuchu = 0;
};

class Owoc
{
private:
    int liczbaPunktow = 0;
public:
    int sprawdzPunkty()
    {
```

```
27          return this->liczbaPunktow;
28      }
29      void ustawPunkty(int liczbaPunktow)
30      {
31          this->liczbaPunktow = liczbaPunktow;
32      }
33  };
34
35  class WyswietlanieGrafiki
36  {
37  public:
38      void wyswietlGre()
39      {
40
41      }
42      WyswietlanieGrafiki()
43      {
44
45      }
46  };
47
48  class Pole
49  {
50  public:
51      Owoc owoc;
52      int wartoscPola = 0;
53  };
```

Przepisaliśmy tu jedynie informacje ze stworzonego przez nas diagramu klas. Problem zaczyna się podczas tworzenia klasy `Plansza`. Według diagramu trzeba w niej utworzyć dwuwymiarową tablicę obiektów klasy `Pole`. Z racji tego, że wiemy z wyprzedzeniem, że plansza ma mieć modyfikowalną postać z poziomu plików konfiguracyjnych, nie możemy zadeklarować jej wielkości na stałe w kodzie.

> **Już wiesz**
>
> O ile w C++ możemy zadeklarować wskaźnik na tablicę jednowymiarową:
>
> ```
> 1  int tab[];
> ```
>
> o tyle nie możemy zrobić tego w identyczny sposób z tablicą dwu- lub więcej wymiarową.

```
int tab[][]; //błąd kompilacji
```

Żeby zadeklarować tablicę dwu- lub więcej wymiarową, podajemy jej drugi (i każdy kolejny) wymiar.

```
int tab[][100]; //działa
```

Zadeklarujemy zatem klasę `Plansza` w następujący sposób:

```
class Plansza
{
public:
    int rozmiarXPlanszy;
    int rozmiarYPlanszy;
    Pole** pole;            // wskaźnik do tablicy wskaźników

    void losujPlansze()
    {

    }
    void zresetujPlansze()
    {

    }
    void zaaktualizujOgonWeza()
    {

    }
    Plansza(int rozmiarX, int rozmiarY)
    {
        rozmiarXPlanszy = rozmiarX;
        rozmiarYPlanszy = rozmiarY;

        pole = new Pole* [rozmiarYPlanszy];

        for (int i = 0; i < rozmiarYPlanszy; i++)
        {
            pole[i] = new Pole[rozmiarXPlanszy];
            for (int j = 0; j < rozmiarXPlanszy; j++)
            {
```

```
32                  pole[i][j] = Pole();
33              }
34          }
35      }
36  };
```

Pamiętamy, że każda „normalna” tablica jest wskaźnikiem pozwalającym nam bezpośrednio odwoływać się do elementów, które w sobie agreguje.

> **Już wiesz**
>
> Zapis:
>
> ```
> 1  tab[5] = 10;
> ```
>
> Co w zasadzie jest przez program rozumiane jako:
>
> ```
> 1  *(tab+5) = 10;
> ```
>
> To z kolei na język naturalny przekładamy następująco: „Weź adres pamięci, który zapisany jest pod zmienną tab, a następnie zwiększ ją o 5 * długość pojedynczego elementu w tablicy. Zapisz liczbę 10 pod obliczonym w ten sposób adresem”.
>
> Jeśli `tab` byłoby tablicą liczb `int`, to wtedy, zakładając że mamy czterobajtową liczbę całkowitą (`int`), odwołalibyśmy się do następującej komórki pamięci o numerze:
>
> $$tab + 5 \cdot 4$$
>
> W przedstawionym równaniu `tab` reprezentuje adres logiczny w pamięci.

W omawianym przypadku utworzymy dynamiczną tablicę, tzn. taką, której rozmiar nie jest znany w czasie kompilacji. Wykorzystujemy do tego celu możliwości, jakie daje nam dynamiczne alokowanie pamięci za pomocą słowa kluczowego `new`. W klasie `Plansza` deklarujemy wskaźnik do tablicy wskaźników.

Konstruktor klasy `Plansza` alokuje dynamicznie pamięć na tablicę wskaźników o rozmiarze `rozmiarYPlanszy`. Dzieje się to w linii 25 przedstawionego kodu. Będą to wskaźniki do kolejnych tablic, które będą odpowiadać wierszom planszy. Następnie dla każdego wiersza odpowiadającemu mu wskaźnikowi przypisywana jest kolejna tablica, alokowana dynamicznie w linii 29. Te tablice będą już zawierały konkretne pola, a więc

odpowiadające im obiekty klasy `Pole` (linia 32). Rozmiary tych tablic wynoszą `rozmiarXPlanszy` (a więc odpowiadają liczbie kolumn).

W ten sposób utworzyliśmy faktycznie dynamiczną tablicę dwuwymiarową, która zawiera w pierwszym wymiarze wskaźniki do tablic odpowiadających konkretnym polom planszy. Jak widać w linii 32, odwołanie się do tak zadeklarowanej i zaalokowanej tablicy jest bardzo proste – wystarczy zapis `pole[wiersz][kolumna]`.

Dodatkowo powinniśmy również utworzyć destruktor, ponieważ używaliśmy słowa kluczowego `new`. W destruktorze będziemy usuwać obiekty utworzone przez wspomniane słowo kluczowe.

```
1 ~Plansza()
2 {
3     for (int i = 0; i < rozmiarYPlanszy; ++i)
4         delete[] pole[i];
5     delete[] pole;
6 }
```

Mamy już wygenerowaną planszę w pamięci, spróbujmy teraz ją wyświetlić w konsoli – zaimplementujmy klasę `WyswietlanieGrafiki`.

> **Ważne!**
>
> Żeby program dobrze zadziałał, klasa `WyswietlaniaGrafiki` powinna zostać umieszczona po deklaracjach klas `Waz` i `Plansza`.

W samym konstruktorze nie ma potrzeby, abyśmy cokolwiek uzupełniali. W diagramie klas umieściliśmy pomiędzy klasą `WyswietlanieGrafiki` a `Plansza` agregację częściową. Zrobiliśmy to, ponieważ sam obiekt klasy `WyświetlanieGrafiki` może istnieć bez istnienia obiektów klasy `Waz` i klasy `Plansza`, dlatego też sam konstruktor zostawiamy pusty.

Za to uzupełniamy samą metodę `wyswietlGre()`.

```
1 void wyswietlGre(Waz* waz, Plansza* plansza)
2 {
3     system("CLS");
4     for (int i = 0; i < plansza->rozmiarYPlanszy; i++)
5     {
6         for (int j = 0; j < plansza->rozmiarXPlanszy; j++)
7         {
8             std::cout << plansza->pole[i][j].wartoscPola << "\t";
```

```
        }
        std::cout << std::endl;
    }
}
```

Jako argumenty tej metody podaliśmy wskaźnik do obiektu klasy `wąż` i wskaźnik do obiektu klasy `plansza`. Informacje o tych obiektach będziemy przekazywać bezpośrednio przez referencje, zamiast przez kopie obiektów. Postępujemy tak, gdyż w przypadku obiektu klasy `plansza`, nasza plansza nie zostałaby poprawnie skopiowana – doszłoby do kopiowania płytkiego.

Funkcja `system()` pozwala przekazywać polecenia, które podamy jako argument, bezpośrednio do konsoli. Polecenie `cls` służy do czyszczenia konsoli. Jeśli chcielibyśmy to sprawdzić, wystarczy uruchomić konsole systemu Windows, a następnie wpisać w niej `cls`, polecenie to nie jest case sensitive. Używamy tutaj poleceń z systemuWindows, ponieważ zgodnie z założeniami podanymi w pierwszym materiale mamy zrobić grę pod „System Windows 10 lub nowsze".

Jak widać w linijkach 4 – 9, pętle przechodzą kolejno po wierszach i kolumnach planszy, odwołując się do obiektów reprezentujących pola w taki sposób jak pokazaliśmy, omawiając konstruktor klasy `Plansza`.

Teraz, jeśli wywołamy `main` w ten sposób:

```
int main()
{
    Plansza plansza = Plansza(5, 5);
    WyswietlanieGrafiki grafika = WyswietlanieGrafiki();
    Waz waz = Waz();
    grafika.wyswietlGre(&waz, &plansza);
}
```

Wyświetli się nam w konsoli następujący wydruk:

```
0       0       0       0       0
0       0       0       0       0
0       0       0       0       0
0       0       0       0       0
0       0       0       0       0
```

Jednak pojawia się problem: program wyświetla to tylko raz. Jeśli zrobilibyśmy nieskończoną pętlę w samej metodzie `wyswietlGre()`, uzyskalibyśmy wtedy odświeżanie.

> **Ważne!**
>
> Z racji tego, że mamy tutaj do czynienia z nieskończoną pętlą, w której nie występują żadne opóźnienia, pętla ta będzie wykonywała się najszybciej jak jest to możliwe dla danego procesora. Tym samym, niestety, będzie ona próbować zajmować możliwie jak najwięcej zasobów, potencjalnie zajmując istotną ich część – spowoduje to powolne działanie twojego komputera.
>
> Warto też dodać, że nie ma sensu, by odświeżanie wykonywało się tak często. **Dlatego też spowalniamy działanie tej pętli przed przetestowaniem zaprezentowanego fragmentu kodu.**

```
1  void wyswietlGre(Waz* waz, Plansza* plansza)
2  {
3      while (true)
4      {
5          system("CLS");
6          for (int i = 0; i < plansza->rozmiarYPlanszy; i++)
7          {
8              for (int j = 0; j < plansza->rozmiarXPlanszy; j++)
9              {
10                 std::cout << plansza->pole[i][j].wartoscPola << "
11             }
12             std::cout << std::endl;
13         }
14     }
15 }
```

Ewentualnie linijkę trzecią zmienimy potem, aby wyświetlanie wykrywało, kiedy gra się kończy. Problemem pozostaje jednak fakt, że sama gra musi otrzymywać jakieś sterowanie. W tym celu posłużymy się wątkami.

> **Ważne!**
>
> W kodzie korzystamy z wątków dostępnych od wersji C++11.

```
1  int main()
2  {
3      Plansza plansza = Plansza(5, 5);
4      WyswietlanieGrafiki grafika = WyswietlanieGrafiki();
```

```
    Waz waz = Waz();
    std::thread watekGrafiki(&WyswietlanieGrafiki::wyswietlGre, &
    watekGrafiki.join();
}
```

Uruchomienie wątku będzie wyglądać tak, jak to przedstawiliśmy. W tym przypadku w pierwszym argumencie wskazujemy metodę, którą będziemy się posługiwać, a w drugim argumencie – wskaźnik `this` wybranego przez nas obiektu tej klasy, w ramach którego wskazaną wcześniej metodę będziemy wywoływać w wątku. W dalszych argumentach znajdują się dane, które przekazujemy jako argumenty samej metody, czyli w naszym przypadku wskazujemy na adresy obiektów.

**Przykład 1**

Dla metody z klasy WyswietlanieGrafiki o następującej deklaracji:

```
void wyswietlGre(Waz* waz, Plansza* plansza)
```

Deklaracja wątku będzie wyglądała następująco:

```
std::thread watekGrafiki(&WyswietlanieGrafiki::wyswietlGre, &gr
```

Gdzie:

- `std::thread` – nazwa klasy, z przestrzeni nazw `std`,
- `watekGrafiki` – to nazwa zmiennej przechowujących obiekt klasy `thread`,
- `&WyswietlanieGrafiki::wyswietlGre` – wskazuje, że z przestrzeni nazw klasy `WyswietlanieGrafiki`, użyjemy metody `wyswietlGre`,
- `&grafika` – wskazuje na adres wcześniej zdeklarowanego obiektu klasy `WyswietlanieGrafiki`,
- `&waz, &plansza` – są argumentami dostarczanymi do metody `wyswietlGre()`, czyli odpowiednio: `Waz* waz` oraz `Plansza* plansza`.

Dla optymalnego działania programu spowolnimy działanie tego wątku.

W tym celu skorzystamy z następującego fragmentu kodu:

```
std::this_thread::sleep_for(std::chrono::milliseconds(200));
```

Kod ten zatrzymuje działanie danego wątku na 200 milisekund, czyli na dwie dziesiąte sekundy. Możemy to sobie skonfigurować pod nasze potrzeby.

> **Dla zainteresowanych**
>
> Jeśli chcesz sprawdzić, czy podany fragment kodu rzeczywiście stopuje działanie wątku na tyle milisekund, ile deklaruje, to zobacz, w jaki sposób zadziałałaby metoda `wyswietlGre()` w następującej postaci:

```
void wyswietlGre(Waz* waz, Plansza* plansza)
{
    while (true)
    {
        system("CLS");
        for (int i = 0; i < plansza->rozmiarYPlanszy; i++)
        {
            for (int j = 0; j < plansza->rozmiarXPlanszy; j++)
            {
                std::cout << plansza->pole[i][j].wartoscPola <<
                std::this_thread::sleep_for(std::chrono::millis
            }
            std::cout << std::endl;
        }
    }
}
```

Metoda – z użyciem wcześniej wspominanego fragmentu – wyglądałaby następująco:

```
void wyswietlGre(Waz* waz, Plansza* plansza)
{
    while (true)
    {
        system("CLS");
        for (int i = 0; i < plansza->rozmiarYPlanszy; i++)
        {
            for (int j = 0; j < plansza->rozmiarXPlanszy; j++)
            {
                std::cout << plansza->pole[i][j].wartoscPola << "
            }
            std::cout << std::endl;
        }
        std::this_thread::sleep_for(std::chrono::milliseconds(200
```

```
    }
}
```

Powodując zatrzymanie wątku na dwie dziesiąte sekundy po każdorazowym narysowaniu planszy.

> **Ważne!**
>
> Jeżeli po przetestowaniu, okazało się, że program zużywa wciąż zbyt dużo zasobów twojego komputera, to rozważ zwiększenie opóźnienia do – np. – 1000 milisekund lub więcej.

```
class ObslugaWejscia
{
public:
    void zinterpretujWcisnietyPrzycisk()
    {
        while (true)
        {
            char przycisk = getch();
            switch (przycisk)
            {
            case 'a':
                std::cout << "Nacisnieto a\n";
                break;
            case 'd':
                std::cout << "Nacisnieto d\n";
                break;
            case ' ':
                std::cout << "Nacisnieto spacje\n";
                break;
            }
        }
    }
};
```

W tym przypadku ponownie zdecydowaliśmy się zostawić, przynajmniej na razie, pusty konstruktor. Funkcja `getch()` czyta każdy kliknięty przez nas w konsoli znak od razu, bez potrzeby potwierdzania enterem.

Funkcja ta dostępna jest w ramach pliku nagłówkowego `conio.h`.

> **Ważne!**

> Jeżeli korzystasz z Visual Studio, to możliwe, że otrzymujesz błąd o treści: „'getch': The POSIX name for this item is deprecated. Instead, use the ISO C and C++ conformant name: _getch".
>
> W takim przypadku należy dopisać na samym początku kodu:
>
> ```
> #define _CRT_NONSTDC_NO_DEPRECATE
> ```
>
> Lub korzystać z funkcji `_getch()`.

Teraz sprawdźmy, czy obsługa wejścia działa – w tym celu zakomentujemy linie, które uruchamiają wątek obsługujący grafikę, oraz utworzymy obiekt klasy `ObslugaWejscia`, wraz z odpowiednim wątkiem.

```
int main()
{
    Plansza plansza = Plansza(5, 5);
    WyswietlanieGrafiki grafika = WyswietlanieGrafiki();
    Waz waz = Waz();
    //std::thread watekGrafiki (&WyswietlanieGrafiki::wyswietlGre
    ObslugaWejscia wejscie;
    std::thread watekWejscia(&ObslugaWejscia::zinterpretujWcisnie
    //watekGrafiki.join();
    watekWejscia.join();
}
```

Po skompilowaniu i użyciu klawiszy: `a`, `d`, `spacja` oraz `n` wyświetlą się następujące komunikaty.

```
Nacisnieto a
Nacisnieto d
Nacisnieto spacje
```

Przycisk `n` nie wyświetla żadnego komunikatu, ponieważ nie zdefiniowaliśmy dla tego przycisku żadnego zachowania. Kombinacja klawiszy `shift` + `a`, oraz `shift` + `d` również nie daje nam żadnego komunikatu.

Kolejne funkcje gry zaimplementujemy w następnych e-materiałach. Stworzony do tego momentu kod można pobrać, korzystając z linku:

Kod źródłowy C++
Plik o rozmiarze 975.00 B w języku polskim

## Wielowątkowość i sterowanie w języku Python

Zanim zaczniemy omawiać program, przypomnimy sobie, jak uruchomić program w konsoli systemu Windows.

> **Już wiesz**
>
> Z racji tego, że piszemy program dla systemu Windows, otwieramy wiersz poleceń, czyli `cmd`.
>
> Następnie w wierszu poleceń wpisujemy:
>
> ```
> python ścieżkaBezpośredniaDoPliku\plik.py
> ```
>
> Przykład:
>
> ```
> python C:\Users\Sebastian\PycharmProjects\rename\scratch.py
> ```
>
> Wtedy program uruchomi się z konsoli. To, jak stworzyć program, który będzie uruchamiał się jak typowy `plik.exe`, omówimy w kolejnych e-materiałach. Na potrzeby testów na razie to wystarczy.
> Z racji tego, że odczytujemy wszystkie klawisze, niestety nie możemy użyć kombinacji `ctrl` + `c`, żeby zamknąć proces. Dlatego za każdym testem programu włączamy go ponownie. Aby przyspieszyć proces, utwórz plik `.bat` z następującą treścią:
>
> ```
> start python *scieżka*
> ```
>
> Np.:
>
> ```
> start python C:\Users\Sebastian\PycharmProjects\rename\scratch.
> ```
>
> Wtedy będziemy mogli na swoim komputerze, na którym zainstalowany jest Python, bezproblemowo uruchamiać pisany przez nas skrypt do testów w konsoli systemu Windows. Jak stworzyć plik `.exe`, omówimy w kolejnych e-materiałach.

Następnie, tak jak w przypadku dwóch poprzednich języków, zaczniemy od stworzenia wszystkich klas, zgodnie z diagramem klas.

```
class Waz:
    liczbaPunktow = 0
    pozycjaWezaX = 0
    pozycjaWezaY = 0
    kierunekRuchu = 0
    def __init__(self):
        self.liczbaPunktow = 0
        self.pozycjaWezaX = 0
        self.pozycjaWezaY = 0
        self.kierunekRuchu = 0

class Owoc:
    liczbaPunktow = 0
    def __init__(self):
        self.liczbaPunktow = 0
    def sprawdzPunkty(self):
        return
    def ustawPunkty(self):
        return

class Plansza:
    rozmiarXPlanszy = 0
    rozmiarYPlanszy = 0
    pole = None
    def __init__(self, rozmiarXPlanszy, rozmiarYPlanszy):
        self.rozmiarXPlanszy = rozmiarXPlanszy
        self.rozmiarYPlanszy = rozmiarYPlanszy
        #self.pole = #tu musimy dopisac tworzenie sie samej dwuwy
        return
    def losujPlansze(self):
        return
    def zresetujPlansze(self):
        return
    def zaaktualizujOgonWeza(self):
        return

class Pole:
    owoc = None
    wartoscPola = 0
    def __init__(self):
        self.owoc = None
```

```
        self. wartoscPola = 0

class WyswietlanieGrafiki:
    def wyswietlGre(self):
        return

class ObslugaWejscia:
    def zinterpretujWcisnietyPrzycisk(self):
        return
```

W kolejnym kroku stworzymy obsługę wejścia – z racji tego, że tworzymy program na system Windows, to skorzystamy z biblioteki: `msvcrt`

```
class ObslugaWejscia:
    def zinterpretujWcisnietyPrzycisk(self):
        import msvcrt
        while 1:
            x = msvcrt.getch().decode('ASCII')
            print(x)
```

Przedstawiony fragment kodu w nieskończonej pętli wyświetla na wyjście to, co dostanie na wejściu. Metoda `getch()` zapewnia nam odczyt wejścia bajt po bajcie. Metoda `decode()` dekoduje wejście – a podany argument sprawia, że uznajemy znaki za takie, które występują w ASCII. Na razie zostawimy to w tej formie. Żeby przetestować, czy działa to u ciebie, spróbuj wywołać następujący kod:

```
x = ObslugaWejscia()
x.zinterpretujWcisnietyPrzycisk()
```

Spróbujmy teraz stworzyć i wyświetlić planszę.

```
class Plansza:
    rozmiarXPlanszy = 0;
    rozmiarYPlanszy = 0;
    pole = None;
    def __init__(self, rozmiarXPlanszy, rozmiarYPlanszy):
        self.rozmiarXPlanszy = rozmiarXPlanszy
        self.rozmiarYPlanszy = rozmiarYPlanszy
        self.pole = [[Pole() for x in range(rozmiarXPlanszy)] for
```

```
        return
    def losujPlansze(self):
        return
    def zresetujPlansze(self):
        return
    def zaaktualizujOgonWeza(self):
        return
```

W tym fragmencie kodu, w konstruktorze klasy `Plansza`, umieściliśmy tworzenie dwuwymiarowej listy obiektów klasy `Pole`.

```
class WyswietlanieGrafiki():
    def wyswietlGre(self, plansza):
        for i in range(plansza.rozmiarYPlanszy):
            for j in range(plansza.rozmiarXPlanszy):
                print(plansza.pole[j][i].wartoscPola, end='\t')
            print("\n")
```

Następnie umieściliśmy wypisywanie wszystkich linii, umieszczonych w klasie plansza.

Przykładowe wywołanie:

```
plansza = Plansza(5, 5)
wyswietlanieGrafiki = WyswietlanieGrafiki()
wyswietlanieGrafiki.wyswietlGre(plansza)
```

Wynik ostatniego fragmentu kodu:

```
0   0   0   0   0

0   0   0   0   0

0   0   0   0   0

0   0   0   0   0

0   0   0   0   0
```

Na razie robimy to tylko raz, potem zmienimy w taki sposób, aby się odświeżało.

Nie możemy jednak jednocześnie odczytywać wejścia i rysować planszy, ponieważ obie czynności są w praktyce nieskończone. Skorzystamy w tym przypadku z omówionych we wcześniejszych e-materiałach wątków.

Możemy zrobić to na dwa sposoby – albo wywoływać metody klas w ramach wątków, albo odziedziczyć wątki w ramach poszczególnych klas.

My skorzystamy z pierwszej opcji.

Poprawne uruchomienie wątku wejścia wyglądałoby w następujący sposób:

```
obslugaWejscia = ObslugaWejscia()
watekWejscia = threading.Thread(target=ObslugaWejscia.zinterpretu
watekWejscia.start()
watekWejscia.join()
```

> **Ważne!**
>
> Do użycia klasy `threading` może być konieczne jej zaimportowanie.

Wskazujemy w nim `target` jako metodę danej klasy, a następnie – jako argument `self`– obiekt tej klasy.
Potem uruchamiany wątek i robimy `join()`, żeby program się nie zakończył przed końcem uruchomionego wcześniej wątku.

Analogicznie robimy z wątkiem grafiki:

```
plansza = Plansza(5, 5)
wyswietlanieGrafiki = WyswietlanieGrafiki()
wyswietlanieGrafiki.wyswietlGre(plansza)
watekGrafiki = threading.Thread(target=WyswietlanieGrafiki.wyswie
watekGrafiki.start()
watekGrafiki.join()
```

Po stworzeniu obu wątków wywołania wyglądałyby następująco:

```
plansza = Plansza(5, 5)
wyswietlanieGrafiki = WyswietlanieGrafiki()
watekGrafiki = threading.Thread(target=WyswietlanieGrafiki.wyswie
```

```
obslugaWejscia = ObslugaWejscia()
watekWejscia = threading.Thread(target=ObslugaWejscia.zinterpretu
watekWejscia.start()
watekGrafiki.start()
watekWejscia.join()
watekGrafiki.join()
```

Poprawiamy wyświetlanie grafiki, żeby ekran się odświeżał.

```
class WyswietlanieGrafiki():
    def wyswietlGre(self, plansza):
        import os
        import time
        while 1:
            os.system('CLS')
            for i in range(plansza.rozmiarYPlanszy):
                for j in range(plansza.rozmiarXPlanszy):
                    print(plansza.pole[j][i].wartoscPola, end='\t
                print("\n")
            time.sleep(1)
```

Cały poprzedni kod umieściliśmy w nieskończonej pętli oraz zaimportowaliśmy jeszcze klasę `time`, dzięki której spowolniliśmy odświeżanie ekranu, żeby mniej obciążać komputer. Zrobiliśmy tak, ponieważ – w przypadku tej gry – niepotrzebne jest odświeżanie częściej niż co sekundę.

Polecenie `os.system()` powoduje wysłanie polecenia do konsoli. Polecenie `cls` powoduje wyczyszczenie okna konsoli.

Na koniec tego e-materiału poprawmy jeszcze obsługę wejścia, żeby powoli zaczynała interpretować kliknięcia odpowiednich klawiszy. Chwilowo zakomentujemy również działanie wątku generującego grafikę.

```
class ObslugaWejscia():
    def zinterpretujWcisnietyPrzycisk(self):
        import msvcrt
        while 1:
            x = msvcrt.getch().decode('ASCII')
            if x == 'a':
                print("Naciśnięto a")
```

```
            elif x == 'd':
                print("Naciśnięto d")
            elif x == ' ':
                print("Naciśnięto spacje")
```

Przedstawiliśmy edycję klasy, teraz zapiszmy przykładowe wywołanie wątku:

```
plansza = Plansza(5, 5)
wyswietlanieGrafiki = WyswietlanieGrafiki
#watekGrafiki = threading.Thread(target=WyswietlanieGrafiki.wyswi
obslugaWejscia = ObslugaWejscia
watekWejscia = threading.Thread(target=ObslugaWejscia.zinterpretu
#watekGrafiki.start()
watekWejscia.start()
watekWejscia.join()
#watekGrafiki.join()
```

Po użyciu kombinacji klawiszy `a`, `d`, `spacja`, `a` + `shift`, `d` + `shift` wyświetlą się w konsoli następujące komunikaty:

```
Naciśnięto a
Naciśnięto d
Naciśnięto spacje
```

Stąd wniosek, że program odróżnia wielkie litery od małych znaków.

Kolejne mechaniki rozgrywki zaimplementujemy w następnych e-materiałach.

Korzystając z zamieszczonego linku, możesz pobrać końcową wersję kodu źródłowego Python dla tego e-materiału.

Plik źródłowy Python
Plik o rozmiarze 824.00 B w języku polskim

## Słownik

> ***case sensitive***
>
> wrażliwy na wielkość liter, tj. odróżniający wielkość liter; jeśli coś jest *case sensitive*, to wtedy „aBc” != „ABC”; jeśli coś nie jest *case sensitive*, to wtedy „AbC” == „ABC”
> **grafika komputerowa**

dział informatyki zajmujący się generowaniem oraz przetwarzaniem obrazów

**wątek**

część programu (sekwencja instrukcji), która działa w ramach danego procesu; w jednym procesie może działać wiele wątków

# Animacja

**Polecenie 1**

Zapoznaj się z animacją. Następnie poszukaj informacji na temat innych problemów spotykanych podczas programowania obiektowego niż te, o których mowa w animacji.


Film dostępny pod adresem /preview/resource/R1drkZlwcyYvz



Film nawiązujący do treści materiału: Programowanie obiektowe.

---

**Polecenie 2**

Przygotuj notatkę, w której podsumujesz najważniejsze informacje zawarte w animacji.

# Sprawdź się

Pokaż ćwiczenia:

### Ćwiczenie 1

Zaznacz poprawną odpowiedź. Wskaż, czy program, który jest wykonywany wielowątkowo, zawsze będzie wykonywał się w taki sam sposób.

- ○ Nie, ponieważ kolejność wykonywania wątków zależy od tego, w jakim stopniu są zajęte zasoby komputera oraz od samego systemu operacyjnego.
- ○ Tak, ponieważ współczesne komputery są deterministyczne i nie ma w nich miejsca na losowość.

### Ćwiczenie 2

Dokończ zdanie, zaznaczając wszystkie poprawne odpowiedzi.
**Jeżeli program jest wielowątkowy, to...**

- ☐ można go synchronizować za pomocą zamków lub programowo.
- ☐ w praktyce nie możemy być nigdy pewni, że wykonywane przez niego obliczenia są poprawne.
- ☐ nie można go synchronizować w żaden sposób.

**Ćwiczenie 3**

Napisz program w wybranym przez siebie języku programowania, który będzie sczytywał i wyświetlał na ekranie informację o kliknięciu strzałek kierunkowych na klawiaturze. Następnie po użyciu przycisku spacja, program wypisze, jakie strzałki zostały użyte w kolejności od tej klikniętej najwcześniej, do tej klikniętej najpóźniej. Swoje rozwiązanie porównaj z odpowiedzią.

**Specyfikacja:**

*Dane:*

- `lista` – zmienna typu tablicowego

*Wynik:*

Program na wyjściu standardowym zwróci informację, jakie przyciski zostały wciśnięte przez użytkownika.

```
1
```

# Dla nauczyciela

**Autor:** Maurycy Gast
**Przedmiot:** Informatyka

**Temat: Programowanie obiektowe – projekt, etap VI**

**Grupa docelowa:**

Liceum ogólnokształcące i technikum, liceum ogólnokształcące, technikum, zakres podstawowy i rozszerzony

**Podstawa programowa:**

Cele kształcenia – wymagania ogólne

I. Rozumienie, analizowanie i rozwiązywanie problemów na bazie logicznego i abstrakcyjnego myślenia, myślenia algorytmicznego i sposobów reprezentowania informacji.

Treści nauczania – wymagania szczegółowe

I. Rozumienie, analizowanie i rozwiązywanie problemów.

Zakres podstawowy. Uczeń:

2) stosuje przy rozwiązywaniu problemów z różnych dziedzin algorytmy poznane w szkole podstawowej oraz algorytmy:

b) na tekstach: porównywania tekstów, wyszukiwania wzorca w tekście metodą naiwną, szyfrowania tekstu metodą Cezara i przestawieniową,

**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 wątki w praktyce.
- Stworzysz pierwsze klasy na podstawie diagramu.
- Napiszesz proste sterowanie gry.

**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);
- oprogramowanie dla języka Python 3 (lub nowszej wersji), w tym PyCharm lub IDLE;
- oprogramowanie dla języka C++11, w tym kompilator GCC/G++ 4.8.1 (lub nowszej wersji) i Code::Blocks 16.01 (lub nowszej wersji), Orwell Dev-C++ 5.11 (lub nowszej wersji) lub Microsoft Visual Studio 2019 16.8.

**Przebieg lekcji**

**Przed lekcją:**

1. **Przygotowanie do zajęć.** Nauczyciel loguje się na platformie i udostępnia e-materiał: „Programowanie obiektowe – projekt, etap VI". Nauczyciel prosi uczniów o zapoznanie się z treściami w sekcji „Przeczytaj".

**Faza wstępna:**

1. Wyświetlenie przez nauczyciela tematu i celów lekcji. Określenie wiążących dla uczniów kryteriów sukcesu.
2. **Rozpoznanie wiedzy uczniów.** Uczniowie tworzą pytania dotyczące tematu zajęć, na które odpowiedzą w trakcie lekcji.

**Faza realizacyjna:**

1. **Praca z multimedium.** Nauczyciel wyświetla zawartość sekcji „Animacja". Uczniowie zapoznają się z animacją a następnie indywidualnie wykonują program przedstawiony na filmie.
2. **Ćwiczenie umiejętności.** Uczniowie realizują indywidualnie ćwiczenia nr 1 i 2, po ich wykonaniu porównują otrzymane wyniki z inną osobą.

**Faza podsumowująca:**

1. Wybrany uczeń podsumowuje zajęcia, zwracając uwagę na nabyte umiejętności, omawia ewentualne problemy podczas rozwiązania ćwiczeń.

**Praca domowa:**

1. Uczniowie wykonują ćwiczenie 3 z sekcji „Sprawdź się". Ich zadaniem jest napisanie programu w wybranym języku programowania, który będzie sczytywał i wyświetlał na ekranie informację o kliknięciu TYLKO strzałek kierunkowych na klawiaturze. Następnie po użyciu przycisku spacja, program ma za zadanie wypisanie, jakie strzałki zostały użyte w kolejności od tej klikniętej najwcześniej, do tej klikniętej najpóźniej.

**Materiały pomocnicze:**

- Oficjalna dokumentacja techniczna dla języka Java SE 8 (lub nowszej wersji).
- Oficjalna dokumentacja techniczna dla oprogramowania Eclipse 4.4 (lub nowszej wersji).
- Oficjalna dokumentacja techniczna dla języka Python 3 (lub nowszej wersji).
- Oficjalna dokumentacja techniczna dla oprogramowania PyCharm lub IDLE.
- Oficjalna dokumentacja techniczna dla oprogramowania GCC/G 4.8.1 (lub nowszej wersji) i Code::Blocks 16.01 (lub nowszej wersji), Orwell Dev-C 5.11 (lub nowszej wersji) lub Microsoft Visual Studio 2019 16.8.
- Oficjalna dokumentacja techniczna dla języka C++.

**Wskazówki metodyczne:**

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