

Języki i paradygmaty programowania

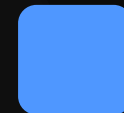
cz. II

Odc. 3

W dzisiejszym odcinku



Zaawansowany OOP



- Dziedziczenie
- Polimorfizm
- Metody wirtualne
 - „zwykłe”
 - pure virtual
- Deklaracje przyjaźni
- Przeładowanie/przeciążenie funkcji i operatorów

Dziedziczenie *ang. inheritance*



```
class Ssak {
public:
    void oddychaj() {
        cout << "Oddycham powietrzem." << endl;
    }
    void karmMlekiem() {
        cout << "Karmię młode mlekiem." << endl;
    }
};
```

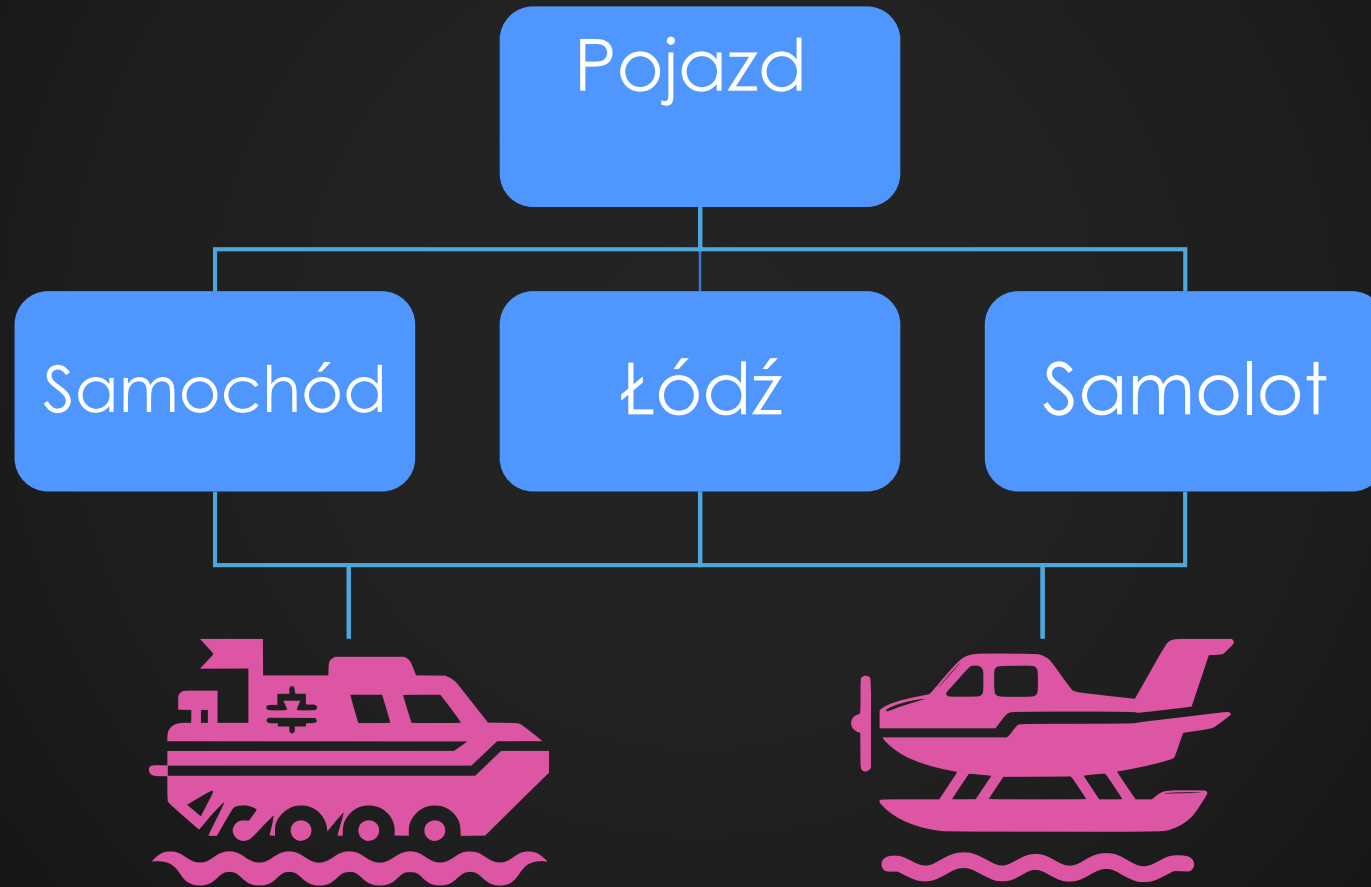
```
class Kotowate : public Ssak {
public:
    void liczbaZebow() {
        cout << "Mam 30 zębów." << endl;
    }
};
```

```
class Psowate : public Ssak {
public:
    void liczbaZebow() {
        cout << "Mam 42 zęby." << endl;
    }
};
```

```
class Lew : public Kotowate {
public:
    void grzywa() {
        cout << "Mam grzywę." << endl;
    }
};
```

```
class Tygrys : public Kotowate {
public:
    void prazki() {
        cout << "Mam prążki." << endl;
    }
};
```

Dziedziczenie wielokrotne



```
class Pojazd {
public:
    // Brak metod w klasie bazowej Pojazd
};
```


```
class Samochod : public Pojazd {
public:
    void jazda() {
        cout << "Jadę" << endl;
    }
};
```

```
class Lodz : public Pojazd {
public:
    void plywanie() {
        cout << "Płynę" << endl;
    }
};
```

```
class Samolot : public Pojazd {
public:
    void lot() {
        cout << "Lecę" << endl;
    }
};
```

```
class Amfibia : public Samochod, public Lodz {
};
```

```
class Hydroplan : public Samolot, public Lodz {
};
```

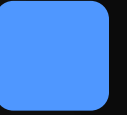


```
int main() {  
    Amfibia amfibia;  
    amfibia.jazda();  
    amfibia.plywanie();  
    return 0;  
}
```

```
Jadę  
Płynę
```

```
...Program finished with exit code 0  
Press ENTER to exit console. 
```


Jak jeszcze utworzyć obiekt klasy?



```
#include <iostream>
#include <string>

class Samochod {
private:
    std::string marka;
    std::string model;
    int rokProdukcji;

public:
    // Konstruktor
    Samochod(std::string m, std::string mo, int r) {
        marka = m;
        model = mo;
        rokProdukcji = r;
        std::cout << "Stworzono obiekt klasy Samochod: " << marka << " " << model << ", rok produkcji: " << rokProdukcji << std::endl;
    }

    // Metoda do wyświetlania informacji o samochodzie
    void pokazInformacje() {
        std::cout << "Marka: " << marka << ", Model: " << model << ", Rok produkcji: " << rokProdukcji << std::endl;
    }
};

int main() {
    // Tworzenie obiektu klasy Samochod
    Samochod mojSamochod("Toyota", "Corolla", 2020); //LUB

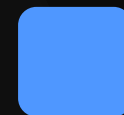
    Samochod* mojSamochod = new Samochod("Toyota", "Corolla", 2020);

    // Wywołanie metody pokazInformacje
    mojSamochod.pokazInformacje();

    delete mojSamochod;

    return 0;
}
```

Polimorfizm



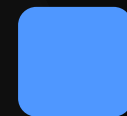
Przemiana w potwora, dzięki której zyskujemy obrażenia zadawane przeciwnikom. Aby użyć umiejętności potrzebna jest **Kula Polimorfii**.

Szybkość ruchu i ataku jest zależna od używanej kuli.



~Metin 2 Wiki

Polimorfizm



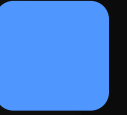
Polimorfizm w programowaniu to taka funkcja lub właściwość kodu, która pozwala na korzystanie z jednego "narzędzia" (np. metody, funkcji) do różnych zadań w zależności od typu rzeczywistego obiektu.

Metody wirtualne



```
class Bazowa {  
public:  
    virtual void metoda() { std::cout << "Metoda klasy Bazowa\n"; }  
};  
  
class Pochodna : public Bazowa {  
public:  
    void metoda() override { std::cout << "Metoda klasy Pochodna\n"; }  
};
```

Metody „pure virtual”



```
class Bazowa {  
public:  
    virtual void metoda() = 0; // Metoda "pure virtual"  
};  
  
class Pochodna : public Bazowa {  
public:  
    void metoda() override { std::cout << "Metoda klasy Pochodna\n"; }  
};
```

```
class Zwierze {
public:
    virtual void dzwiek() = 0
    // Nie możemy wydać dźwięku tutaj,
    //bo nie ma jednego wspólnego dźwięku
    //dla wszystkich zwierząt
};
```

```
class Kot : public Zwierze {
public:
    void dzwiek() override{
        std::cout << "Miau! Miau!" << std::endl;
    }
};
```

```
class Pies : public Zwierze {
public:
    void dzwiek() override{
        std::cout << "Hau! Hau!" << std::endl;
    }
};
```

```
int main() {
    Zwierze* zwierze1 = new Pies();
    Zwierze* zwierze2 = new Kot();

    zwierze1->dzwiek(); // dzwiek() z klasy Pies
    zwierze2->dzwiek(); // dzwiek() z klasy Kot

    delete zwierze1;
    delete zwierze2;

    return 0;
}
```

```
Hau! Hau!
Miau! Miau!
```

```
...Program finished with exit code 0
Press ENTER to exit console. □
```



THE ANTAGONIST

**POWER OF
FRIENDSHIP**

Deklaracje
przyjaźni

```
#include <iostream>

class KlasaB; // Deklaracja wstępna klasy B

class KlasaA {
private:
    int sekretneDane;

public:
    KlasaA() : sekretneDane(123) {}

    // Deklaracja przyjaźni: KlasaB jest teraz przyjacielem KlasaA
    friend class KlasaB;
};

class KlasaB {
public:
    void pokazSekretneDane(KlasaA& a) {
        std::cout << "Sekretne dane KlasaA: " << a.sekretneDane << std::endl;
    }
};

int main() {
    KlasaA a;
    KlasaB b;
    b.pokazSekretneDane(a); // Wyświetla "Sekretne dane KlasaA: 123"

    return 0;
}
```



```

#include <iostream>
use namespace std;

class Tomek; // Deklaracja wstępna klasy Tomek

class PaczkaChipsow {
private:
    int iloscChipsow;
public:
    PaczkaChipsow() : iloscChipsow(123) {}

    // Deklaracja przyjaźni: Tomek jest teraz przyjacielem PaczkaChipsow
    friend class Tomek;
};

class Tomek {
public:
    void pokazIloscChipsow(PaczkaChipsow& p) {
        cout << "W paczce jest jeszcze: " << p.iloscChipsow << " chipsów." << endl;
    }

    void zabierzChipsa(PaczkaChipsow& p) {
        if (p.iloscChipsow > 0) {
            p.iloscChipsow--;
            cout << "Zabrano jednego chipsa." << endl;
        } else {
            cout << "Nie ma już więcej chipsów w paczce!" << endl;
        }
    }
};

class Franek {
public:
    void poprosTomekOChipsa(Tomek& t, PaczkaChipsow& p) {
        cout << "Franek prosi Tomka o jednego chipsa." << endl;
        t.zabierzChipsa(p);
    }
};

int main() {
    PaczkaChipsow paczka;
    Tomek tomek;
    Franek franek;

    franek.poprosTomekOChipsa(tomek, paczka);

    return 0;
}

```



```

class PaczkaChipsow {
private:
    int iloscChipsow;

public:
    PaczkaChipsow() : iloscChipsow(123) {}

    // Deklaracja przyjaźni: Tomek jest teraz przyjacielem PaczkaChipsow
    friend class Tomek;
};

```

```

class Tomek {
public:
    void pokazIloscChipsow(PaczkaChipsow& p) {
        std::cout << "W paczce jest jeszcze: " << p.iloscChipsow << " chipsów." << std::endl;
    }

    void zabierzChipsa(PaczkaChipsow& p) {
        if (p.iloscChipsow > 0) {
            p.iloscChipsow--;
            std::cout << "Zabrano jednego chipsa" << std::endl;
        } else {
            std::cout << "Nie ma już więcej chipsów w paczce!" << std::endl;
        }
    }
};

```

```

class Franek {
public:
    void poprosTomekOChipsa(Tomek& t, PaczkaChipsow& p) {
        std::cout << "Franek prosi Tomka o jednego chipsa." << std::endl;
        t.zabierzChipsa(p);
    }
};

```

```

int main() {
    PaczkaChipsow paczka;
    Tomek tomek;
    Franek franek;

    franek.poprosTomekOChipsa(tomek, paczka);
    return 0;
}

```



Zadanko



Stwórz hierarchię klas reprezentujących różne typy pracowników w firmie.

Wymagania:

1. Klasa bazowa **'Employee'** powinna zawierać informacje o imieniu, nazwisku i wynagrodzeniu.
2. Klasa pochodna **'Manager'** powinna dodatkowo zawierać informacje o **liczbie podwładnych**.
3. Klasa pochodna **'Intern'** nie powinna mieć możliwości dostępu do informacji o wynagrodzeniu.
4. Wszystkie klasy powinny mieć metodę **'IntroduceYourself'**, która jest metodą **wirtualną** w klasie bazowej.
5. Klasa **'Manager'** powinna mieć przyjaciela - funkcję **'ChangeSalary'**, która pozwala na zmianę wynagrodzenia.