Programowanie obiektowe

Wykorzystanie polimorfizmu

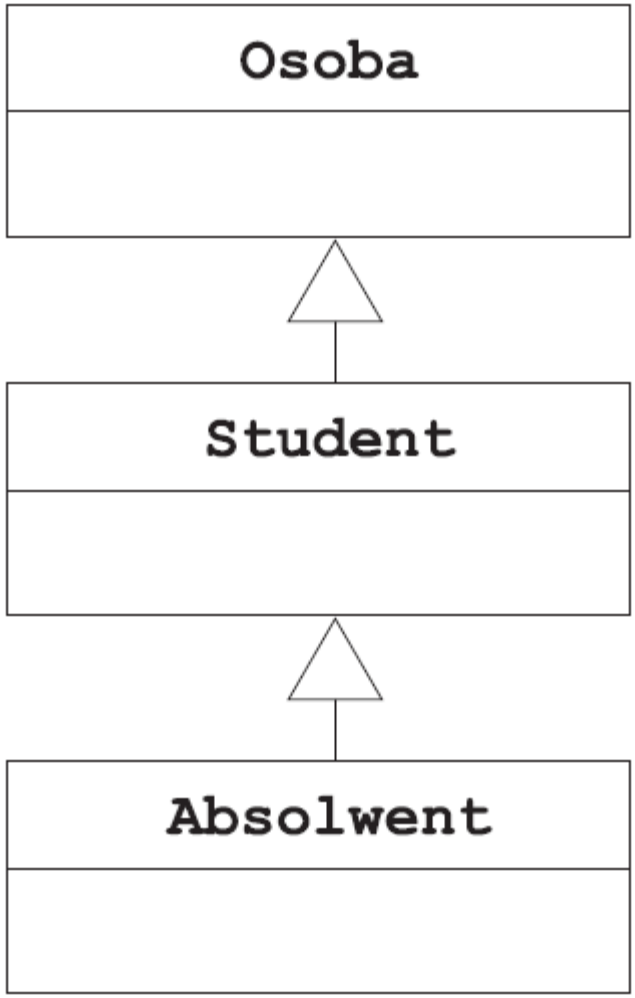
Odwołanie do obiektu przez wskaźnik

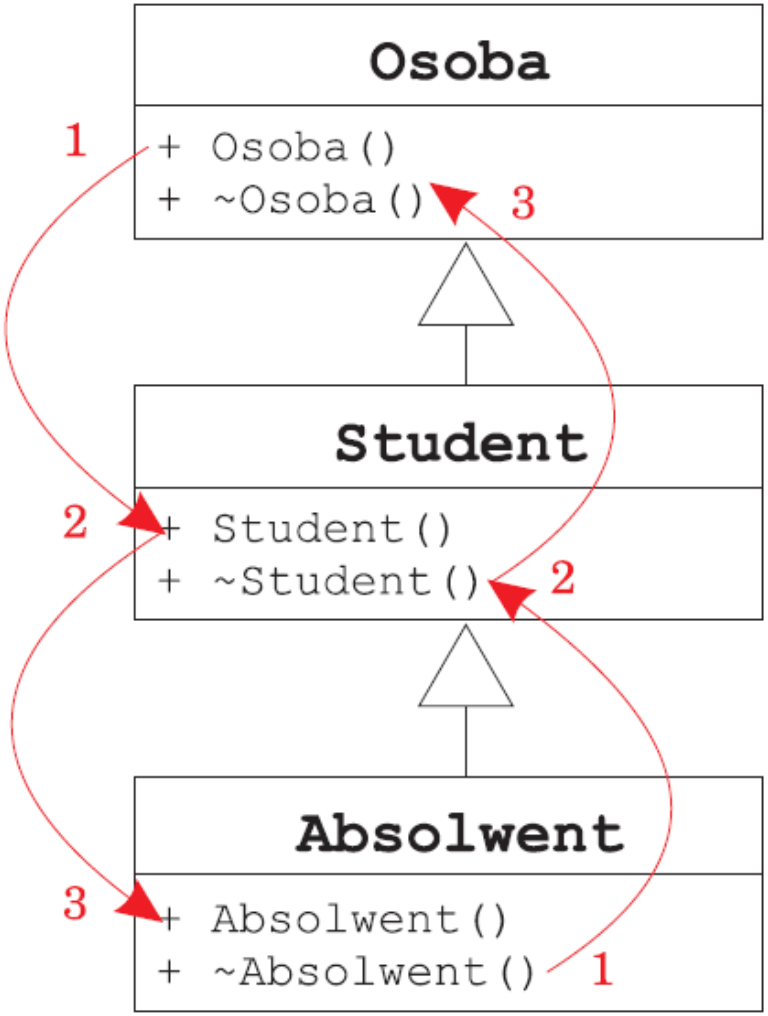
```
Kwadrat kwadr( 5 );
...
wypiszDane( &kwadr );
...
void wypiszDane( Kwadrat * k )
{
cout << "\nDługosc boku : " << (*k).podajDlBoku();
cout << "\nPole kwadratu: " << (*k).obliczPole();
}
//ALBO LEPIEJ:
void wypiszDane( Kwadrat * k )
{
cout << "\nDługosc boku : " << k->podajDlBoku();
cout << "\nPole kwadratu: " << k->obliczPole();
}
```

Dynamiczny przydział pamięci

```
Kwadrat * kwadr;
kwadr = new Kwadrat( 5 );
if( kwadr != 0 )
{
cout << "\nDługosc boku : " << kwadr->podajDlBoku();
cout << "\nPole kwadratu: " << kwadr->obliczPole();
delete kwadr;
}
```

Hierarchia klas – przypomnienie





Nowa hierarchia klas

```
class Osoba
{
  public:
    Osoba() : nrDowodu(0){}
    void piszKimJestes()
    {
    cout << "Osoba\n";
    }
    int nrDowodu;
};
```

```
class Student : public Osoba
{
public:
    Student() : Osoba(),
    nrIndeksu(0) {}
    void piszKimJestes()
    {
        cout << "Student\n";
    }
    int nrIndeksu;
};
```

Nowa hierarchia klas c.d.

```
class Absolwent : public Student
{
  public:
    Absolwent() : Student(), nrDyplomu(0) {}
    void piszKimJestes()
    {
    cout << "Absolwent\n";
    }
    int nrDyplomu;
};
//gdzieś w main()
Osoba o;
Student s;
Absolwent a;
o.piszKimJestes();
s.piszKimJestes();
a.piszKimJestes();
```

Polimorfizm a hierarchia klas

```
int main()
{
  Osoba o;
  Student s;
  Absolwent a;
  o.piszKimJestes();
  o = s;
  o.piszKimJestes();
  o = a;
  o.piszKimJestes();
}
```

```
int main()
{
  Osoba * wo;
  wo = &o;
  wo->piszKimJestes();
  wo = &s;
  wo->piszKimJestes();
  wo = &a;
  wo->piszKimJestes();
}
```

```
Osoba
Osoba
Osoba
```

Ratunek – dynamiczne wiązanie

```
class Osoba
{
  public:
    . . .
    virtual void piszKimJestes()
    {
      cout << "Osoba\n";
    }
    . . .
};
```

```
Osoba
Student
Absolwent
```

Lepszy przykład wykorzystania polimorfizmu

```
class Osoba
{
public:
Osoba() : imie( "" ), nazwisko( "" ), nrDowodu( 0 ) {}
Osoba( string i, string n, int nr ) : imie( i ),
  nazwisko( n ), nrDowodu( nr ) {}
virtual void piszKimJestes() { cout << "\nOsoba\n"; }
virtual void wypiszDane()
{
  piszKimJestes();
  cout << imie << " " << nazwisko << endl;
  cout << "Nr dowodu: " << nrDowodu << endl;
}
int nrDowodu;
string imie;
string nazwisko;
};
```

Lepszy przykład wykorzystania polimorfizmu c.d.

```
class Student : public Osoba
{
public:
Student() : Osoba(), nrIndeksu( 0 ) {}
Student( string i, string n, int nr, int nrI )
: Osoba( i, n, nr ), nrIndeksu( nrI ) {}
void piszKimJestes() { cout << "\nStudent\n"; }
void wypiszDane()
{
  Osoba::wypiszDane();
  cout << "Nr indeksu: " << nrIndeksu << endl;
}
int nrIndeksu;
};
```

Lepszy przykład wykorzystania polimorfizmu c.d.

```
class Absolwent : public Student
{
public:
Absolwent() : Student(), nrDyplomu( 0 ) {}
Absolwent( string i, string n, int nr, int nrI, int
  nrD ): Student( i, n, nr, nrI ), nrDyplomu( nrD ){}
void piszKimJestes() { cout << "\nAbsolwent\n"; }
void wypiszDane()
{
  Student::wypiszDane();
  cout << "Nr dyplomu: " << nrDyplomu << endl;
}
int nrDyplomu;
};
```

Wykorzystanie stworzonych klas

```
Osoba * wo = new Student("Jan", "Nowak", 2222, 3333);
. . .
wo->wypiszDane();
. . .
delete wo;
```

```
Student
Jan Nowak
Nr  dowodu: 2222
Nr indeksu: 3333
```

```
. . .
Osoba * wo = new Absolwent("Jan", "Nowak", 2222, 3333, 4444);
. . .
wo->wypiszDane();
. . .
delete wo;
```

```
Absolwent
Jan Nowak
Nr  dowodu: 2222
Nr indeksu: 3333
Nr dyplomu: 4444
```

Metody czysto wirtualne

```
class KlasaAbstrakcyjna
{
public:
    virtual void MetodaDoNadpisania() = 0;
}
```

Jeżeli w klasie jest przynajmniej jedna metoda czysto wirtualna to nazywamy ją abstrakcyjną.

Wirtualny destruktor

```
class Figura
{
public:
    virtual obliczPole() = 0;
    virtual ~Figura();
};
```

Wirtualny destruktor często pozwala uniknąć problemów z wyciekami pamięci.

Ćwiczenia 1

1. Przekształć poprzednie klasy dotyczące figur geometrycznych tak, by dziedziczyły po klasie bazowej **Figura**. W ramach klasy **Figura** zdefiniuj czysto wirtualną metodę **obliczPole**(). Następnie zdefiniuj w programie tablicę 3 różnych figur geometrycznych (np. zawierającą obiekty Kwadrat, Prostokąt, Sześcian) oraz wypisz na ekranie sumę ich pól powierzchni (stosując odpowiednią iteracje na w/w tablicy).

Ćwiczenia 2

2. Zaimplementuj następujące klasy:
   1. Klasę abstrakcyjną **Funkcja** – posiadającą pole `x` reprezentujące liczbę rzeczywistą oraz funkcję `wartość()`.
   2. Klasę **FunkcjaLiniowa** – dziedziczącą po klasie Funkcja. Jej podprogram `wartosc()` powinien zwracać wynik wyrażenia `ax+b`.
   3. Klasę **FunkcjaKwadratora** – dziedziczącą po klasie Funkcja. Jej podprogram `wartosc()` powinien zwracać wynik wyrażenia $ax^2 + bx + c$.

   Przetestuj działanie stworzonych klas odwołując się do nich przez wskaźnik `Funkcja * test;`

Ćwiczenia 3

3. Napisz klasę **Towar**, posiadającą pola `nazwa`, `cena`, `liczba` i metodę `opis()`, wypisująca na ekran stan obiektu (wartości wszystkich pól).

   Następnie zaimplementuj klasę **Lozko**, dziedzicząca w klasy **Towar**, ale posiadająca dodatkowo pola `długość`, `szerokość` i `wysokość`.

   Ostatecznie stwórz funkcję `wypiszNaEkran(Towar * t[])`, która wypisuje informacje o wszystkich przekazanych w parametrze obiektach.