

# Programowanie Komputerowe

Zajęcia 11

# Klasy pochodne

- Dla klasy  $K$  (zwanej dalej klasą podstawową) możemy zdefiniować klasę pochodną  $L$ .
- Klasa pochodna dziedziczy pola i metody klasy podstawowej, tj. są one dostępne w obiektach klasy pochodnej – nie trzeba ich na nowo deklarować.
- W klasie pochodnej można dodać nowe pola i metody. Będą one dostępne w klasie pochodnej; w podstawowej oczywiście nie.
- Można zmienić działanie metody w klasie pochodnej.
- Klasy pochodnej można użyć jako klasy podstawowej dla innej klasy.
- Klasa może mieć wiele klas podstawowych, ale rodzi to różne problemy i nie należy nadużywać tego mechanizmu.

# Po co stosować klasy pochodne?

- Jeśli potrzebujemy stworzyć klasę, której działanie jest podobne do już istniejącej.
- Jeśli potrzebujemy stworzyć wiele podobnych do siebie klas, które w dużym stopniu działają podobnie. Wtedy tworzymy klasę podstawową, w której definiujemy pola i metody działające tak samo we wszystkich klasach, a te, które są specyficzne dla poszczególnych klas, umieszczamy w klasach pochodnych.
- Jeśli potrzebujemy tworzyć zbiory złożone z różnych obiektów. Przykład: interfejs graficzny programu składa się z wielu "okienek" – prostokątnych obszarów leżących jedno na drugim, z których każdy może działać w inny sposób.

# Deklaracja klasy pochodnej

```
class K {      // klasa podstawowa
    ...
};
class L: public K { // klasa pochodna do klasy K
    // dodatkowe pola i metody
};
```

Istnieje dodatkowy specyfikator dostępu: `protected`:

Oznacza on, że składniki klasy są widoczne w danej klasie i jej klasach pochodnych i nie są widoczne w pozostałej części programu.

# Przykład klasy pochodnej

```
class Punkt {
public:
    void drukuj() { cout << "(" << x << "," << y << ")"; }
    double x;
    double y;
};

class KolorowyPunkt: public Punkt {
public:
    void drukuj() { cout << "(" << x << "," << y << ":kolor-"
<< kolor << ")"; }
    int kolor;
};
```

## Przykład klasy pochodnej (2)

Po zadeklarowaniu klas z poprzedniej strony:

- klasa `Punkt` będzie miała pola `x`, `y` oraz metodę `drukuj`, która drukuje te pola
- klasa `KolorowyPunkt` będzie miała trzy pola:
  - `x`, `y` pochodzące z klasy `Punkt`
  - `kolor` zadeklarowane w klasie `KolorowyPunkt`
- metoda `drukuj` w klasie `KolorowyPunkt` będzie drukować zarówno `x`, `y` jak i `kolor`

# Konstruktory

Konstruktor z klasy pochodnej wywołuje jeden z konstruktorów klasy podstawowej. Jeśli ma być to konstruktor inny niż domyślny, dodajemy wywołanie tego konstruktora na liście inicjującej. Po dodaniu konstruktora:

```
class Punkt {  
public:  
    Punkt(double _x, double _y): x(_x), y(_y) {}  
    void drukuj() { cout << "(" << x << "," << y << ")"; }  
    double x;  
    double y;  
};
```

## Konstruktory (2)

W klasie `KolorowyPunkt` musimy dodać konstruktor, np. tak

```
class KolorowyPunkt: public Punkt {
public:
    KolorowyPunkt(double _x, double _y, double _kolor)
        : Punkt(_x, _y), kolor(_kolor) {}
    void drukuj() {
        cout << "(" << x << ", " << y << ":kolor-"<<kolor<<")";
    }
    int kolor;
};
```



# Funkcje wirtualne

```
int main() {  
    Punkt p(1,2);  
    KolorowyPunkt k(3,4,5);  
    p=k;  
    p.drukuj(); // drukuje bez koloru - p jest typu Punkt  
}
```

Jest sposób, aby działanie metody uzależnić od typu jakiego jest obiekt, mimo, że nie jest to wiadome w momencie pisania programu.

## Funkcje wirtualne (2)

```
int main() {  
    Punkt* w=new Punkt(1,2); // tworzymy dynamicznie obiekt typu Punkt  
    KolorowyPunkt* c=new KolorowyPunkt(3,4,5);  
    (*c).drukuj(); // kolor się drukuje  
    w=c;  
    (*w).drukuj(); // jak to powinno działać?  
}
```

Zamiast `(*w).drukuj()` można napisać `w->drukuj()` – wygodny skrót.

# Funkcje wirtualne (3)

- Działanie powyższego przykładu zależy od tego, czy funkcja została zadeklarowana jako wirtualna.
- Deklarację funkcji wirtualnej poprzedzamy słowem `virtual`, np.  

```
virtual void drukuj() { ... }
```
- Jeśli funkcja jest wirtualna, to wszystkie odpowiedniki tej funkcji w klasach pochodnych też muszą być wirtualne.
- Żeby korzystać z dynamicznego wiązania, tj. wyboru odpowiedniej funkcji w trakcie działania programu, musimy korzystać ze wskaźników, bo mogą one wskazywać na obiekty różnych typów.

# Metody i klasy abstrakcyjne

- W klasach można umieścić metody abstrakcyjne, tzn. takie których działanie nie jest określone.
- Robimy to przypisując 0, np. tak

```
virtual void f()=0;
```
- Klasa, która zawiera taką metodę jest abstrakcyjna i nie możemy tworzyć obiektów tej klasy.
- Metody abstrakcyjne należy przeddefiniować w klasach pochodnych. Jeśli klasa pochodna definiuje wszystkie metody abstrakcyjne, jej obiekty można tworzyć.

# Klasy abstrakcyjne - przykład

```
class Figura {  
    virtual void rysuj()=0; // nie wiemy jak narysować  
dowolną figurę  
};  
class Kolo: public Figura {  
    virtual void rysuj() { ... } // koła rysujemy przy pomocy  
tej metody  
};  
class Kwadrat: public Figura {  
    virtual void rysuj() { ... } // a kwadraty rysujemy tak  
};
```

# Ćwiczenia

1. Zaimplementować klasę `Obrazek`. Obiekt tej klasy reprezentuje kwadrat; każdemu punktowi tego kwadratu przyporządkowany jest numer koloru tego punktu. Metody publiczne:

```
Obrazek(double a, char tlo);
```

tworzy obrazek z punktami o współrzędnych od 0 do `a`, wszystkie punkty mają kolor `tlo`.

```
void malujKolo(double posX, double posY, double r, char kolor);
```

maluje koło o środku `(posX, posY)`, promieniu `r` i kolorze `kolor`.

```
void malujProstokat(double x1, y1, x2, y2, char kolor);
```

maluje prostokąt (wypełniony) o wierzchołkach `(x1,y1)`, `(x1,y2)`, `(x2,y1)`, `(x2,y2)` i podanym kolorze.

```
int kolor(double x, double y);
```

zwraca kolor punktu o podanych współrzędnych.

```
void drukuj();
```

drukuje obrazek na ekranie.

## Ćwiczenia (2)

2. Zaimplementować grę w szachy. Należy użyć klas odpowiadających poszczególnym rodzajom bierek (tj. figur i pionków), które są klasami pochodnymi abstrakcyjnej klasy `Bierka`.