# Wykład 4
# Dziedziczenie i polimorfizm

**dr inż. Maciej Kusy**

Katedra Podstaw Elektroniki
Wydział Elektrotechniki i Informatyki
Politechnika Rzeszowska

Programowanie w języku C#

# Plan wykładu

- Dziedziczenie
- Polimorfizm – wprowadzenie, późne wiązanie
- Pojęcie specjalizacji i uogólniania
- Używanie dziedziczenia: wywołanie konstruktora klasy podstawowej (**`base`**)
- Implementacja polimorfizmu (**`virtual`**, **`override`**), proces przesłaniania
- Dziedziczenie i polimorfizm – praktyczne uwagi
- Klasy abstrakcji i ich cechy; klasy i metody zamknięte
- Nadpisywanie metod (**`new`**)
- Metody rozszerzające, przykład rozszerzenia
- Klasa główna **`Object`**

# Dziedziczenie

**Dziedziczenie** – technika pozwalająca na definiowanie nowej klasy (**pochodnej**) na podstawie – *na bazie* – klasy już istniejącej (**podstawowej**).

**Dziedziczenie** definiuje pewien *związek* między klasami: klasa pochodna jest *szczególnym przypadkiem* klasy bazowej.

**Cel** – rozszerzenie funkcjonalności typu pochodnego o dodatkowe możliwości.

**Efekt** – oszczędność pracy, minimalizacja kodu (wielokrotne wykorzystanie kodu) odzwierciedlenie rzeczywistości w kodzie programu.

# Polimorfizm – wprowadzenie

**Polimorfizm** – ***Poli*** oznacza wiele, ***morf*** to forma – wielość form. Możliwość używania typu w wielu formach. Poznanie typu obiektu w trakcie działania programu.

**Cel** – wykazywanie różnej funkcjonalności podczas wywoływania metody.

**Uwaga** – nie jest ważny typ obiektu, na rzecz którego wywoływana jest dana metoda.

**Efekt** – oszczędność pracy, wielokrotne wykorzystanie kodu przejrzystość, logika zapisu.

# Polimorfizm – późne wiązanie

**Polimorfizm** – inaczej: późne wiązanie (*ang. late binding*), wiązanie w czasie wykonania programu (*ang. runtime binding*), wiązanie dynamiczne (*ang. dynamic binding*).

**Sytuacja przeciwna** – decyzja o tym, która metoda zostanie wywołana podejmowana jest na etapie kompilacji i na podstawie zadeklarowanego typu obiektu – tzw. wczesne (statyczne) wiązanie (*ang. early (static) binding*).

# Specjalizacja i uogólnianie

**Specjalizacja** – relacja ***jest-czymś***: pies jest ssakiem (pies to **specjalny** rodzaj ssaka).

**Uogólnienie** – zależność odwrotna do specjalizacji, można określić poprzez relację: wybrane (nie wszystkie) cechy są wspólne, np.: kot i pies posiadają **ogólne** właściwości ssaków.

Relacje są hierarchiczne (tworzą drzewa): wyspecjalizowane typy to gałęzie odchodzące od bardziej ogólnych typów.

Im wyżej w hierarchii, tym większy stopień ogólności typów.

# Używanie dziedziczenia

Tworzenie klasy pochodnej:

```
class Button : Control
```

Klasa pochodna dziedziczy wszystkie składowe klasy bazowej: pola, metody, właściwości.

W języku C# nie jest możliwe dziedziczenie prywatne ani chronione (C++).

W języku C# nie jest możliwe dziedziczenie wielokrotne (C++) – język obsługuje dziedziczenie pojedyncze.

# Konstruktor klasy bazowej i pochodnej

```
class Figura
{
   private int x, y;
   public Figura(int x, int y)
   {
      this.x = x;
      this.y = y;
   }

}

class Kwadrat : Figura
{
   private int bok;
   public Kwadrat(int x, int y, int bok) : base(x,y)
   {
      this.bok = bok;
   }
}
```

# Implementacja polimorfizmu – 1

Do implementacji polimorfizmu używane są metody wirtualne (**`virtual`**) oraz mechanizm dziedziczenia.

```
class Figura
{
    public virtual void Rysuj(){ }
}
```

Do utworzenia nowej wersji metody w klasie pochodnej (przesłonięcia metody z klasy podstawowej) stosuje się słowo kluczowe **`override`** (*ang. uchylić, być ważniejszym*).

```
class Kwadrat : Figura
{
  public override void Rysuj()
  {
    base.Rysuj();
    //Kod rysowania dla kwadratu
  }
}
```

```
class Trojkat : Figura
{
  public override void Rysuj()
  {
    base.Rysuj();
    //Kod rysowania dla trójkąta
  }
}
```

# Implementacja polimorfizmu – 2

Dzięki takiej hierarchii można zapisać:

```
Figura f = null;
char znak = char.Parse(System.Console.ReadLine());
switch(znak)
{
   case 'a' : { f = new Kwadrat(1, 1, 10); break; }
   case 'b' : { f = new Trojkat(2, 2, 12, 6); break; }
   default : { f = new Figura(0, 0); break; }
}
f.Rysuj();
```

Która metoda ruszy do pracy?

Na tym etapie nie można tego określić. Zależy to od tego, czy użytkownik wciśnie znak **a**, **b** lub jakikolwiek inny znak.

# Realizacja polimorfizmu – 1

Decyzja, która z metod zostanie wywołana jest podejmowana w czasie wykonywania się programu (późne wiązanie) a nie w czasie kompilacji (wczesne wiązanie).

Jak to się dzieje?
Każda klasa, która ma jedną lub więcej metod wirtualnych, posiada jeden egzemplarz tablicy metod wirtualnych (**vtable**) wspólny dla wszystkich obiektów. Elementami tej tablicy są adresy tych metod.

Jeżeli typem obiektu jest klasa posiadająca metody wirtualne, to kompilator dodaje do tego obiektu ukrytą referencję powiązaną z tablicą metod wirtualnych.

# Realizacja polimorfizmu – 2

Podczas wywoływania metody wirtualnej, kod podąża śladem referencji do tablicy metod wirtualnych odpowiedniej klasy.

Następnie odnajduje stosowny wpis w tablicy metod wirtualnych dotyczący kodu właściwej metody.

W przypadku odziedziczenia klasy z metodami wirtualnymi tablica metod wirtualnych jest przesłaniana.

# Praktyczne uwagi

- Metoda o sygnaturze **`override`** jest też metodą wirtualną.
- Słowo **`virtual`** może również występować przy definicji właściwości i zdarzenia.
- Metoda wirtualna nie może być metodą prywatną.
- Metoda wirtualna nie może być metodą statyczną.
- Metodę należy definiować jako wirtualną tylko wtedy, gdy w klasach pochodnych będzie ona przesłonięta.
- Wywołanie metody wirtualnej jest wolniejsze niż metody zwykłej.
- Przeciążając operatory równy (**`==`**) i różny (**`!=`**) należy zawsze przesłonić metodę **`Equals`**.

# Klasy abstrakcyjne

Klasa abstrakcyjna, to klasa, która nie może mieć swoich przedstawicieli w postaci obiektów: stworzenie obiektu, którego typ jest określony przez klasę abstrakcyjną, spowoduję błąd kompilacji.

Jest klasą bazową w całkowitej hierarchii klas.

Zawiera zbiór pól oraz metod wspólnych dla wszystkich klas po niej dziedziczących.

Składnia: `abstract class NazwaKlasy {}`

Przykład: klasa **`Control`** powinna być abstrakcyjna, bo pozwala określić czym jest kontrolka, choć sama nie służy do tworzenia żadnej konkretnej kontrolki.

# Cechy klas abstrakcyjnych

Można w nich definiować metody abstrakcyjne (dodanie słowa kluczowego **`abstract`** po modyfikatorze dostępu).

Są to metody, które nie posiadają ciała.
Mogą one występować wyłącznie w klasach abstrakcyjnych.

Metodę deklaruje się jako abstrakcyjną wtedy, gdy w danej klasie nie jest możliwe podanie jej „sensownej” definicji.

Każda metoda abstrakcyjna jest wirtualna.

W klasach pochodnych konieczne jest przesłonięcie każdej metody abstrakcyjnej (słowo kluczowe **`override`**).

Klasa abstrakcyjna może posiadać implementację, tzn.: posiadać konstruktory, pola, właściwości, metody (wirtualne).

Można stworzyć obiekt klasy abstrakcyjnej za pomocą konstruktora klasy pochodnej.

# Nadpisywanie metod (**new**)

Metody wirtualne mogą być nadpisane. Służy do tego słowo kluczowe **new**. Przerywa to mechanizm wirtualności.

```
class NowaPochodna : Pochodna
{
   public new void Metoda() {}
}
```

Użycie słowa **new** powoduje stworzenie nowej metody nie związanej z metodą klasy bazowej.
Metoda ta może oczywiście być znowu wirtualna i być wykorzystana do „celów polimorficznych” w hierarchii klas.

# Klasy i metody zamknięte

Klasa zamknięta (**`sealed`**), to klasa, po której nie można dziedziczyć (nie można po niej tworzyć klas pochodnych). Metody wirtualne tej klasy nie mogą być zatem przesłonięte.

Przykładem klasy zamkniętej jest klasa **`String`**. Nie można po niej dziedziczyć. Można jedynie ją rozszerzyć.

Metoda zamknięta, to przesłonięta metoda wirtualna, której nie można już przesłonić w klasach pochodnych.

# Metody rozszerzające

W języku **C# 3.0** wprowadzono możliwość rozszerzania istniejących klas o nowe metody.

Metody rozszerzające definiowane są jako statyczne w statycznych klasach, ale mogą być wywoływane tak, jak metody składowe innych wcześniej zdefiniowanych klas.

Pierwszym argumentem metody rozszerzającej musi być referencja **`this`**. Kolejne argumenty są dowolnego typu.

# Przykład rozszerzenia

Dodanie do klasy **`String`** metody, która zlicza wystąpienia zadanego znaku w *stringu* (argument **`arg`**).

```
static class Rozszerzenie
{
    public static int Zlicz(this String arg, char znak)
    {
        return (arg.Split(znak)).Length - 1;
    }
}
```

Od tej pory powyższą metodę można wywoływać na rzecz każdego obiektu klasy **`String`**.

# Klasa główna **System.Object**

Klasa podstawowa dla wszystkich klas języka C#
(również typów skalarnych).

Jest klasą główną w hierarchii dziedziczenia.

Udostępnia wirtualne metody, które mogą być przesłaniane przez klasy pochodne, np.:

```
bool Equals(Object obj)
int GetHashCode()
Type GetType()
string ToString()
```