

## SPIS TREŚCI

---

<b>Wstęp Do Programowania Obiektowego:</b>	<b>2</b>
Klasa A Obiekt:	2
Istota Programowania Obiektowego:	2
Tworzenie Klasy:	3
<b>Metody, Atrybuty I Konstruktory Klas:</b>	<b>8</b>
Metoda Klasy (Czym Jest I Do Czego Służy):	8
Deklarowanie Metody:	9
Atrybuty Klasy:	12
Definiowanie Atrybutów:	13
Konstruktory:	13
Deklaracja Konstruktora:	15
<b>Przestrzeń Nazw, Modyfikatory Dostępu:</b>	<b>16</b>
Istota Przestrzeni Nazw:	16
Modyfikatory Dostępu:	16
Dostępne Modyfikatory Dostępu:	17
<b>Wstęp Do Dziedziczenia:</b>	<b>19</b>
przykład Dziedziczenia	20
Ilustracja Mechanizmu Dziedziczenia	22
<b>Polimorfizm:</b>	<b>23</b>
Przykład Zastosowania Mechanizmu Polimorfizmu:	24
<b>Wzorzec RTTI, Operator As I IS</b>	<b>27</b>
Przykład Zastosowania RTTI:	27
<b>Klasy Abstrakcyjne</b>	<b>31</b>
Przykład Zastosowania Klasy Abstrakcyjnej:	31
<b>Typy Wyliczeniowe:</b>	<b>33</b>
Przykład Zastosowania Typu: Wyliczeniowego:	33
<b>Klasa Wyjątku, Zgłaszanie Wyjątku I Jego Obsługa:</b>	<b>34</b>
Definiowanie Własnego Wyjątku:	34
Obsługa Wyjątku:	37

# WSTĘP DO PROGRAMOWANIA OBIEKTOWEGO:

---

## KLASA A OBIEKT:

---

**Klasa** – Opis nowej struktury danych

**Obiekt** – Struktura danych stworzona zgodnie z opisem klasy

## ISTOTA PROGRAMOWANIA OBIEKTOWEGO:

---

Programowanie obiektowe umożliwia opisanie cech oraz funkcjonalności elementów, na przykład drukarki, monitora, klawiatury, samochodu itd. za pomocą abstrakcyjnego pojęcia zwanego klasą, które ułatwia implementację danego problemu.

Dzięki podejściu obiektowemu możliwy jest logiczny podział projektu na komponenty (obiekty), w celu ułatwienia implementacji rozwiązania a w szczególności ułatwienia rozbudowy projektu.

## TWORZENIE KLASY:

---

W celu utworzenia klasy należy umieścić poniższy kod, w tym przypadku będzie to klasa BasicUser:

```
public class BasicUser  
{  
    |  
}  
}
```

*RYSUNEK 1 UTWORZENIE KLASY*

W celu utworzenia obiektu klasy BasicUser należy umieścić poniższy kod:

```
namespace Cash_dispenser
{
    2 usages
    public class BasicUser
    {
    }

    public class Program
    {
        public static void Main(string[] args)
        {
            BasicUser basicUser = new BasicUser();
        }
    }
}
```

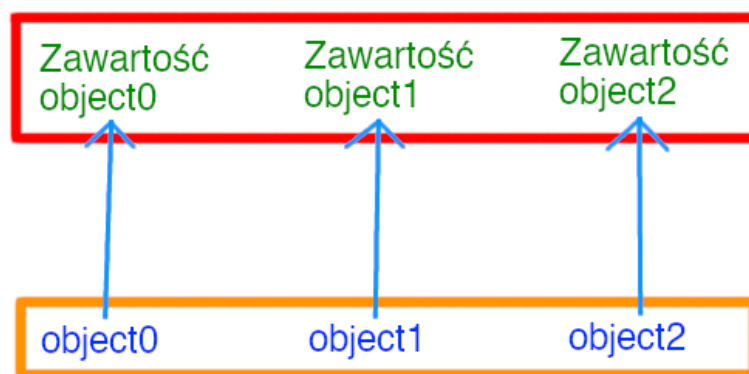
RYSUNEK 2 UTWORZENIE OBIEKTU KLASY

Obiekt klasy (typu) BasicUser jest tworzony i umieszczany w zasięgu metody Main.

Słowo kluczowe **namespace** będzie omawiane w następnych rozdziałach, tyczy się to także modyfikatora dostępu **public**, słowa kluczowego **static** oraz wyjaśnienia pojęcia **metod** w programowaniu obiektowym.

# MECHANIZM GENEROWANIA OBIEKTÓW KLAS:

## Pamięć Komputera



## Kod Źródłowy

RYSUNEK 3 KOD ŹRÓDŁOWY A PAMIĘĆ KOMPUTERA

```
public class Object
{
}

Michał *
public class Program
{
    Michał *
    public static void Main(string[] args)
    {
        Object object0 = new Object();
        Object object1 = new Object();
        Object object2 = new Object();
    }
}
```

RYSUNEK 4 KOD ŹRÓDŁOWY DO RYSUNKU3

Możliwe jest utworzenie kilku obiektów tej samej klasy. Istotny jest fakt, że każdy obiekt jest niezależny względem pozostałych obiektów, które powstały i tych, które mogą później zostać zdefiniowane.

W procesie definiowania nowych obiektów ważnym aspektem jest umieszczenie słowa kluczowego **new**, które sprawia, że tworzona jest nowa, niezależna instancja obiektu określonej klasy.

Odwołując się do rysunku 3 można zauważyć, że każdy element zdefiniowany w kodzie jako `object0`, `object1`, `object2` jest w swej istocie jedynie aliasem (referencją) obiektu, który jest umieszczony w określonym miejscu pamięci komputera.

Biorąc ten fakt pod uwagę poniższy kod powoduje, że istnieją dwa odniesienia do zawartości obiektu `object1`, co daje możliwość operowania na zawartości obiektu1 za pomocą dwóch aliasów, w tym przypadku `object11`.

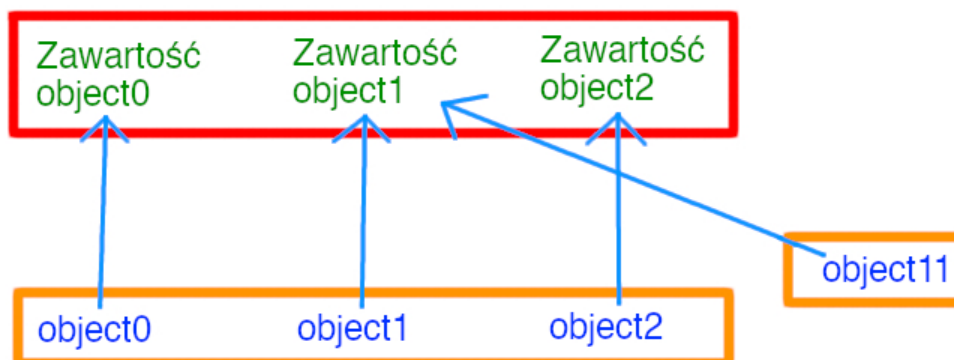
Michał \*

```
public static void Main(string[] args)
{
    Object object0 = new Object();
    Object object1 = new Object();
    Object object2 = new Object();

    Object object11 = object1;
}
```

RYSUNEK 5 KOD ŹRÓDŁOWY Z DODATKOWĄ REFERENCJĄ

## Pamięć Komputera



## Kod Źródłowy

*RYSUNEK 6 UWZGLĘDNIENIE NOWEJ REFERENCJI*

# METODY, ATRYBUTY I KONSTRUKTORY KLAS:

---

## METODA KLASY (CZYM JEST I DO CZEGO SŁUŻY):

---

Metoda jest odpowiednikiem funkcji w podejściu strukturalnym. Metoda wchodzi w skład klasy, umożliwiając operowanie na danych będących:

- Elementami składowymi klasy
- Danymi, przekazywanymi do metody za pomocą argumentów formalnych
- Danymi zaalokowanymi statycznie (dostępnych w obrębie całej aplikacji, bądź określonej przestrzeni nazw)
- Danymi zaalokowanymi w zasięgu bloku metody



## DEKLAROWANIE METODY:

---

W celu deklaracji należy stworzyć klasę i zdefiniować metodę, tak jak na przykład ilustruje to poniższy kod:

```
public class Class0
{
    public void Method()
    {
    }
}
```

*RYSUNEK 7 DEKLARACJA METODY TYPU VOID*

Słowo kluczowe **public** zostanie omówione w następnym rozdziale.

Istotne jest w tym przypadku słowo kluczowe **void**, które informuje, że metoda będzie przetwarzać dane, nie zwracając żadnej wartości po zakończeniu jej wykonywania w miejscu jej wywołania.

Deklaracja każdej metody może być opatrzona słowem kluczowym **void** bądź **typem prostym** bądź **typem obiektu**. W dwóch ostatnich przypadkach oznacza to, że po zakończeniu wykonywania metody, zwróci ona dane typu zdefiniowanego w deklaracji metody.

```
public class Class0
{
    public int Method()
    {
        return 10;
    }
}
```

RYSUNEK 8 DEKLARACJA METODY TYPU PROSTEGO (INT)

```
public class Class0
{
    public decimal Method()
    {
        return 1_500_678.678M;
    }
}
```

RYSUNEK 9 DEKLARACJA METODY TYPU PROSTEGO (DECIMAL)

```
public class Class0
{
    public Class1 Method()
    {
        return new Class1();
    }
}
```

RYSUNEK 10 DEKLARACJA METODY TYPU OBIEKT

Każda metoda zwracająca wartość (**typ prosty**, bądź **typ obiektu**) musi posiadać instrukcję **return**, informującą jaką wartość, bądź obiekt będzie zwrócona/zwrócony.

Jak wcześniej opisano, metoda może przyjmować wartości zwane argumentami formalnymi. W celu zdefiniowania takiej metody można posłużyć się poniższym kodem:


```
public class Class0
{
    public void Method(int number0, int number1)
    {
        Console.WriteLine($"Wartość argumentu formalnego number0: {number0}");
        Console.WriteLine($"Wartość argumentu formalnego number1: {number1}");
    }
}
```

*RYSUNEK 11 DEKLARACJA METODY Z PARAMETRAMI FORMALNYMI*

W celu wyświetlenia przekazanych wartości do metody w terminalu użyto metody **WriteLine** klasy **Console** i interpolacji łańcuchów znakowych (**`$"{} {} {}"`**).

Możliwe jest zdefiniowanie metody przyjmującej dowolną ilość danych danego typu. W celu zdefiniowania takiej metody należy skorzystać ze słowa kluczowego **params**.

Przykładowa definicja metody korzystającej z wyżej opisywanego mechanizmu w celu wyświetlenia wszystkich przesłanych danych typu int została zilustrowana poniżej:

```
public class Class0
{
     1 usage
    public void Method(params int[] numbers)
    {
        foreach (var number:int in numbers)
        {
            Console.WriteLine(number);
        }
    }
}
```

RYSUNEK 12 DEKLARACJA METODY ZE ZMIENNĄ ILOŚCIĄ PARAMETRÓW

```
class Program
{
    static void Main(string[] args)
    {
        Class0 C0 = new Class0();

        C0.Method( params numbers: 2, 3, 4);
    }
}
```

RYSUNEK 13 PRZYKŁADOWE WYKORZYSTANIE METODY ZE ZMIENNĄ ILOŚCIĄ PARAMETRÓW

## ATRYBUTY KLASY:

---

Atrybuty są elementami składowymi klas służącymi do przechowywania danych związanych z daną klasą. Dane te mogą być:

- Typem prostym (int, long, double, decimal itd.)
- Typem złożonym (obiekt, struktura danych itp.)

## DEFINIOWANIE ATRYBUTÓW:

---

W celu zdefiniowania atrybutów klasy można posłużyć się poniższym przykładem:

```
public class Class0
{
    public int number0;
    public int number1;
    public int number2;
}
```

RYSUNEK 14 DEKLARACJA ATRYBUTÓW KLASY

```
public class Class0
{
    public Class1 class1;
    public Class2 class2;
}
```

RYSUNEK 15 DEKLARACJA ATRYBUTÓW KLASY  
JAKO OBIEKTÓW

## KONSTRUKTORY:

---

Konstruktor jest to metoda, wywoływana w chwili utworzenia nowej instancji klasy. Konstruktor może być:

- Bezparametryczny (sygnatura konstruktora jest pusta)
- Wieloargumentowy (sygnatura konstruktora składa się z przynajmniej jednego argumentu formalnego)
- Statyczny (Wywoływany w momencie pierwszej interakcji z klasą, na przykład utworzenie nowej instancji klasy, bądź odczytanie wartości statycznego atrybutu klasy)

## DEKLARACJA KONSTRUKTORA:

---

```
public class Class0
{
    1 usage
    public Class0()
    {
    }
}
```

RYSUNEK 16 DEKLARACJA KONSTRUKTORA  
BEZPARAMETRYCZNEGO

```
public class Class0
{
    1 usage
    public Class0(int number0, double number1)
    {
    }
}
```

RYSUNEK 17 DEKLARACJA KONSTRUKTORA  
WIELOARGUMENTOWEGO

```
public class Class0
{
    static Class0()
    {
        Console.WriteLine("Static Constructor");
    }
}
```

RYSUNEK 18 DEKLARACJA KONSTRUKTORA STATYCZNEGO

Blok konstruktora może posiadać instrukcje, które mają być wykonane w trakcie obsługi kodu konstruktora, jak na przykład: wyświetlenie określonego łańcucha znakowego.

# PRZESTRZENIE NAZW, MODYFIKATORY DOSTĘPU:

---

## ISTOTA PRZESTRZENI NAZW:

---

Przestrzeń nazw jest abstrakcyjnym obszarem, służącym do grupowania elementów składowych projektu, jak na przykład:

- Pojedyncze Klasy
- Biblioteki DLL (Dynamic Link Library)
- Programy Wykonywalne

W celu zdefiniowania przestrzeni nazw i zawartych w niej elementów należy użyć słowa kluczowego **namespace**, co ilustruje poniższy kod:

```
namespace namespace0
{
    public class Class0
    {
    }

    public class Class1
    {
    }
}
```

*RYСУNEK 19 DEKLARACJA PRZESTRZENI NAZW*



## MODYFIKATORY DOSTĘPU:

---

Modyfikatory dostępu **służą do określenia dostępu** do elementów składowych klasy. Ograniczanie dostępu do elementów składowych stosuje się w celu hermetyzacji danych, która jest jedną z głównych zasad programowania obiektowego.

---

### DOSTĘPNE MODYFIKATORY DOSTĘPU:

---

**public** – Dostęp do elementów składowych jest nieograniczony

Dostęp Do Zasobu Mają:

- Metody składowe klasy
- Klasy dziedziczące po klasie
- Klasy korzystające z instancji klasy

**protected** – Dostęp do elementów składowych jest ograniczony

Dostęp Do Zasobu Mają:

- Metody składowe klasy
- Klasy dziedziczące po klasie

**private** – Dostęp do elementów składowych jest ograniczony

Dostęp Do Zasobu Mają:

- Metody składowe klasy

**internal** – Dostęp do elementów składowych jest ograniczony

Dostęp Do Zasobu Mają:

- Metody składowe klasy
- Klasy wchodzące w skład tej samej przestrzeni nazw

**protected internal** – Dostęp do elementów składowych jest ograniczony

Dostęp Do Zasobu Mają:

- Metody składowe klasy
- Klasy dziedziczące po klasie
- Klasy wchodzące w skład tej samej przestrzeni nazw

**private protected** – Dostęp do elementów składowych jest ograniczony

Dostęp Do Zasobu Mają:

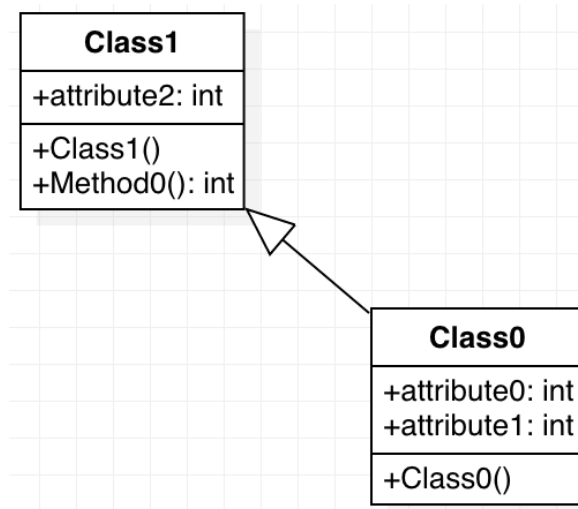
- Metody składowe klasy
- Klasy dziedziczące po klasie

# WSTĘP DO DZIEDZICZENIA:

---

Dziedziczenie jest mechanizmem umożliwiającym **implementowanie cech klas do innych klas**. Dzięki temu, możliwe jest łatwiejsze modelowanie programu, co wiąże się z krótszym czasem projektowania schematu projektu.

## PRZYKŁAD DZIEDZICZENIA



RYSUNEK 20 DIAGRAM KLAS W JĘZYKU UML

```
namespace namespace0
{
    1 usage 1 inheritor
    public class Class1
    {
        public int attribute2;

        public Class1()
        {
        }

        public int Method0()
        {
            return 102;
        }
    }

    public class Class0 : Class1
    {
        public int attribute0;
        public int attribute1;

        public Class0()
        {
        }
    }
}
```

RYSUNEK 21 ODWZOROWANIE  
DIAGRAMU KLAS W KOD ŹRÓDŁOWY

Powyższy kod źródłowy, pokazuje, że aby zastosować mechanizm dziedziczenia należy dopisać do nazwy klasy ‘: ,<nazwa klasy bazowej>’. Zabieg taki powoduje, że klasa staje się **klasą pochodną** (Class0), dziedzicząc po **klasie bazowej** (Class1).

Każda klasa dziedzicząca po innej klasie ma dostęp do **elementów składowych** instancji takiej klasy opatrzonych klauzulą **public**, **protected**, **protected internal** bądź **private protected**.

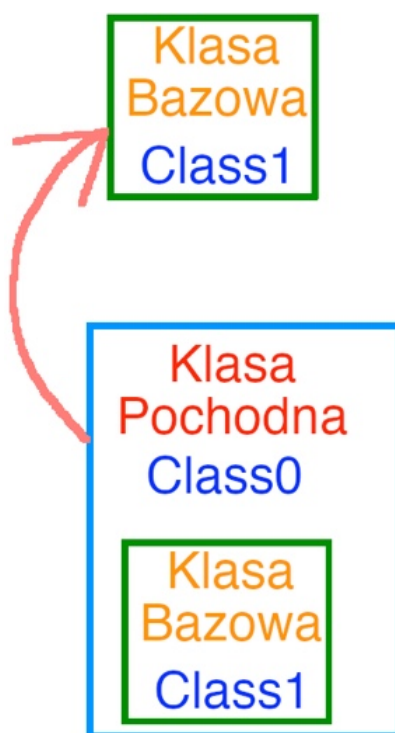
Poniższy kod ilustruje przykład użycia elementów składowych **klasy bazowej** w instancji **klasy pochodnej**:

```
public class Program
{
    public static void Main()
    {
        Class0 C0 = new Class0();
        int number0 = C0.Method0();
    }
}
```

RYSUNEK 22 PRZYKŁAD ZASTOSOWANIA DZIEDZICZENIA

## ILUSTRACJA MECHANIZMU DZIEDZICZENIA

---



RYSUNEK 23 ILUSTRACJA MECHANIZMU  
DZIEDZICZENIA

Zgodnie z powyższym przykładem kodu źródłowego, w momencie tworzenia klasy **Class0** tworzona jest dodatkowo instancja klasy **Class1**, która jest **zagnieżdżona** w instancji klasy **Class0**, dzięki czemu możliwe jest odwoływanie się do jej atrybutów bądź metod.

Przyswojenie tej właściwości może pomóc w zrozumieniu zagadnienia **polimorfizmu**, które będzie omawiane w kolejnych rozdziałach.

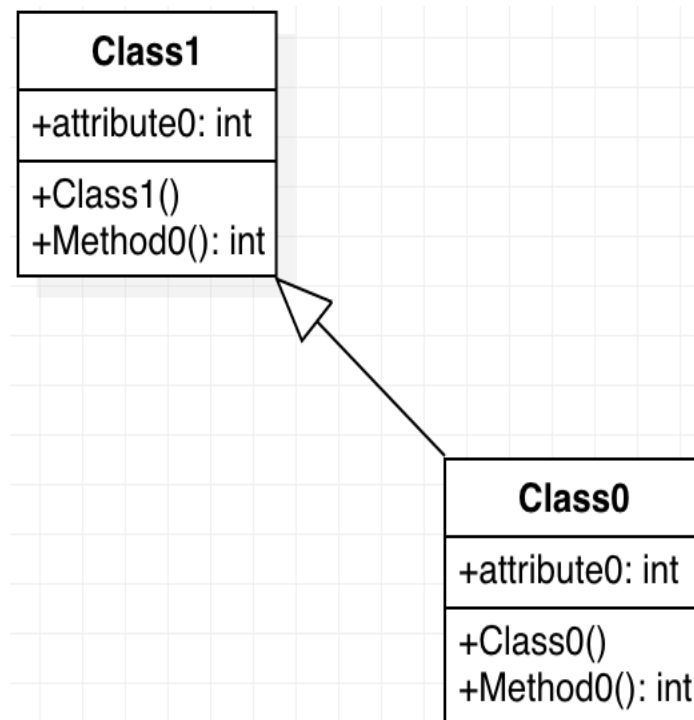
# POLIMORFIZM:

---

Polimorfizm jest mechanizmem pozwalającym na używanie atrybutów, metod itp. na kilka różnych sposobów, poprzez korzystanie z **referencji polimorficznej**.

## PRZYKŁAD ZASTOSOWANIA MECHANIZMU POLIMORFIZMU:

---



RYSUNEK 24 MODEL KLAS DO WYKORZYSTANIA  
MECHANIZMU POLIMORFIZMU



```

public class Class1
{
    public int attribute0;

    public Class1()
    {
    }

    1 usage 1 override
    public virtual int Method0()
    {
        return 102;
    }
}

```

RYSUNEK 25 DEKLARACJA KLASY  
BAZOWEJ

```

public class Class0 : Class1
{
    public int attribute0;

    1 usage
    public Class0()
    {
    }

    0+1 usages
    public override int Method0()
    {
        return 1002;
    }
}

```

RYSUNEK 26 DEKLARACJA KLASY  
POCHODNEJ

W celu zdefiniowania klas korzystających z właściwości **polimorfizmu**, należy:

- W klasie bazowej użyć słowa kluczowego **virtual**, które sprawia, że metoda staje się **wirtualna** (wywołanie jej za pomocą **referencji polimorficznej** umożliwia korzystanie z definicji metody klasy, na którą wskazuje referencja)
- W klasie pochodnej użyć słowa kluczowego **override**, które sprawia, że definicja metody będzie uwzględniana w przypadku użycia **referencji polimorficznej**

```
public class Program
{
    public static void Main()
    {
        Class1 C1 = new Class0();

        Console.WriteLine(C1.Method0());
    }
}
```

RYSUNEK 27 UŻYCIE KLAS OBSŁUGUJĄCYCH  
MECHANIZM POLIMORFIZMU

Zdefiniowanie referencji obiektu klasy bazowej (**referencja polimorficzna**) i przypisanie do niej obiekt klasy pochodnej z uprzednio zastosowanymi operacjami daje możliwość korzystania z elementów składowych klasy pochodnej za pomocą referencji klasy bazowej.

Dzięki takiemu rzutowaniu, możliwe jest ograniczenie obsługi atrybutów, metod itp. klasy pochodnej do elementów składowych zagnieżdżonej instancji klasy bazowej w obiekcie klasy pochodnej.

# WZORZEC RTTI, OPERATOR AS IS

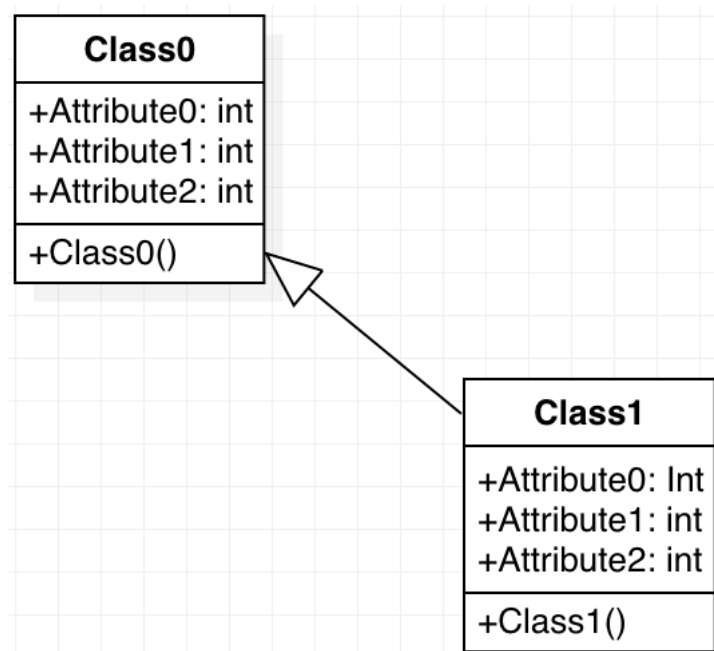
---

Wzorzec projektowy RTTI (runtime type identification) jest techniką stosowaną w nowoczesnych obiektowych językach programowania polegającą na identyfikowaniu danych na podstawie typu klasy instancji

## PRZYKŁAD ZASTOSOWANIA RTTI:

---

W celu zastosowania wzorca RTTI wymagane jest zdefiniowanie klas na przykład w taki sposób jak poniżej:



RYSUNEK 28 MODEL KLAS DO WYKORZYSTANIA  
WZORCA RTTI

```

public class Class0
{
    public int Attribute0;
    public int Attribute1;
    public int Attribute2;


    public Class0()
    {
    }
}

```

RYSUNEK 29 DEKLARACJA KLASY BAZOWEJ

```

public class Class1 : Class0
{
    public int Attribute0;
    public int Attribute1;
    public int Attribute2;

     1 usage
    public Class1()
    {
    }
}

```

RYSUNEK 30 DEKLARACJA KLASY POCHODNEJ

```

class Program
{
    static void Main(string[] args)
    {
        Class1 class1 = new Class1();

        if (class1 is Class1)
        {
            Console.WriteLine("Obiekt jest typu Class1");
        }
    }
}

```

RYSUNEK 31 WYKORZYSTANIE WZORCA RTTI  
ZA POMOCĄ SŁOWA KLUCZOWEGO IS

Dzięki słowu kluczowemu **is** możliwe jest sprawdzenie czy możliwe jest rzutowanie instancji klasy na określoną klasę.

Taka klasyfikacji umożliwia sprawdzenie, czy instancja jest określonego typu, bądź czy klasa instancji jest klasą pochodną względem innej klasy (sprawdzenie możliwości **rzutowania w górę**). W pozytywnym przypadku zwracana jest wartość **true**.

```

class Program
{
    static void Main(string[] args)
    {
        Class0 class1 = new Class0();
        Class1 class2 = class1 as Class1;
    }
}

```

RYSUNEK 32 WYKORZYSTANIE WZORCA RTTI ZA POMOCĄ SŁOWA KLUCZOWEGO AS

Słowo kluczowe **as** ma takie samo zastosowanie jak słowo kluczowe **is** (sprawdzenie możliwości rzutowania instancji klasy na określoną klasę), lecz różni się sposobem działania. Słowo **as** służy do bezpośredniego rzutowania instancji klasy na referencję określonego typu. W przypadku błędnego rzutowania (**rzutowania klasy bazowej na pochodną**) rezultatem rzutowania jest pusta referencja (**referencja z wartością null**).

# KLASY ABSTRAKCYJNE

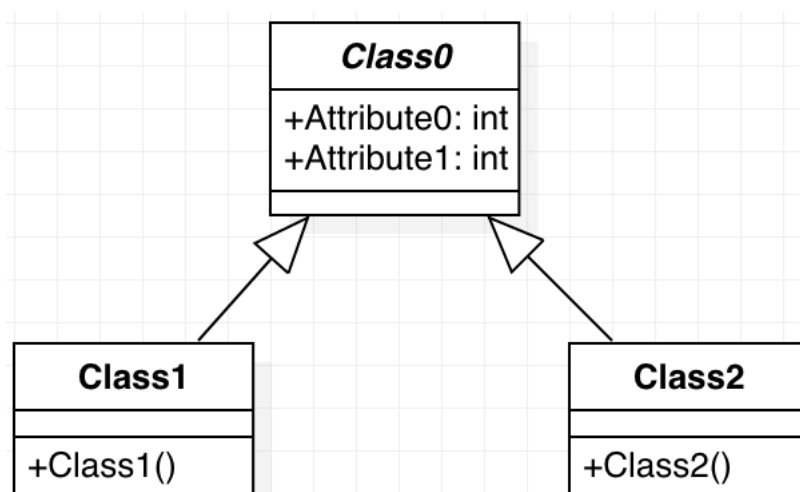
---

**Klasa abstrakcyjna** jest to klasa, która ułatwia modelowanie projektu, będąc **uogólnieniem** innych klas, które mogą dziedziczyć z niej cechy.

Nie jest możliwe utworzenie instancji klasy abstrakcyjnej a jedyne jak wyżej opisano dziedziczenie po niej przez inne klasy.

## PRZYKŁAD ZASTOSOWANIA KLASY ABSTRAKCYJNEJ:

---



RYSUNEK 33 MODEL KLAS DO WYKORZYSTANIA  
KLASY ABSTRAKCYJNEJ

```
public abstract class Class0
{
    public int Attribute0;
    public int Attribute1;
}
```

*RYSUNEK 34 DEKLARACJA KLASY ABSTRAKCYJNEJ*

```
public class Class1 : Class0
{
    public Class1()
    {
    }
}

public class Class2 : Class0
{
    public Class2()
    {
    }
}
```

*RYSUNEK 35 DEKLARACJA KLAS  
DZIEDZICZĄCYCH PO KLASIE ABSTRAKCYJNEJ*

W celu zdefiniowania klasy jako abstrakcyjnej należy wykorzystać słowo kluczowe **abstract**.

Dzięki takiej konstrukcji, możliwe jest oszczędzenie linii kodu wymaganych do modelowania klas.



## TYPY WYLICZENIOWE:

---

Typy wyliczeniowe służą do definiowania wartości, które w prosty sposób mogą być wykorzystane w programie jak i zinterpretowane przez programistę (większa czytelność danych).

### PRZYKŁAD ZASTOSOWANIA TYPU: WYLICZENIOWEGO:

---

W celu zdefiniowania typu wyliczeniowego należy skorzystać ze słowa kluczowego **enum**.

```
public enum EnumType
{
    Attribute0,
    Attribute1,
    Attribute2
}
```

RYSUNEK 36 DEKLARACJA TYPU  
WYLICZENIOWEGO

## KLASA WYJĄTKU, ZGŁASZANIE WYJĄTKU I JEGO OBSŁUGA:

---

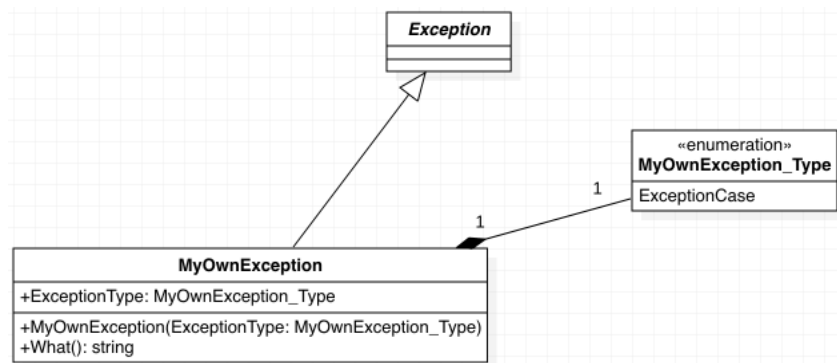
Wyjątek jest to **nieprzewidywalna sytuacja** względem oferowanej funkcjonalności programu. W zależności od typu oraz okoliczności wystąpienia takiej sytuacji może wpłynąć negatywnie na działanie programu bądź nawet go całkowicie zablokować (deadlock w przypadku programowania wielowątkowego itp.)

Wbudowaną klasą obsługującą wyjątki jest klasa **Exception**, która może rozszerzać funkcjonalność nowo definiowanych klas (klasy takie mogą dziedziczyć po klasie **Exception**, dzięki czemu, możliwe jest definiowanie własnych wyjątków)

## DEFINIOWANIE WŁASNEGO WYJĄTKU:

---

W celu zdefiniowania własnego wyjątku jak wyżej opisano, należy utworzyć klasę własnego wyjątku i dziedziczyć po klasie **Exception**.



RYSUNEK 37 MODEL WŁASNEGO WYJĄTKU

```

public enum MyOwnException_Type
{
    ExceptionCase
}
  
```

RYSUNEK 38 DEKLARACJA TYPU WYLICZENIOWEGO  
PRZYPADKÓW WŁASNEGO WYJĄTKU

```

public class MyOwnException : Exception
{
    public MyOwnException_Type ExceptionType;

    public MyOwnException(MyOwnException_Type ExceptionType)
    {
        this.ExceptionType = ExceptionType;
    }

    public string What()
    {
        switch (ExceptionType)
        {
            case MyOwnException_Type.ExceptionCase:
            {
                return "!!! Wystąpił Wyjątek !!!";
            }
        }
    }
}

```

RYSUNEK 39 DEKLARACJA KLASY WŁASNEGO WYJĄTKU CZĘŚĆ 1

```

        break;
    default:
        throw new ArgumentOutOfRangeException();
    }
}
}
}

```

RYSUNEK 40 DEKLARACJA KLASY WŁASNEGO WYJĄTKU CZĘŚĆ 2

W celu zdefiniowania własnego wyjątku wykorzystano już opisany w poprzednim rozdziale **typ wyliczeniowy**, który definiuje przypadki wyjątków obsługiwanych przez **modelowaną klasę**.

Klasa własnego wyjątku posiada konstruktor, który przyjmuje parametr informujący o typie zgłaszanego wyjątku. Metoda **What()** zwraca informację na temat przypadku wyjątku w postaci komunikatu zapisanego w postaci łańcucha znakowego.

## OBSŁUGA WYJĄTKU:

---

W celu obsługi wyjątku należy skorzystać z klauzuli **try catch**, gdzie blok **try** zawiera kod, w którym może zostać zgłoszony wyjątek, zaś blok **catch** zawiera kod, który ma obsłużyć wyjątek określonego typu.

```
try
{
    throw new MyOwnException(MyOwnException_Type.ExceptionCase);
}
catch (MyOwnException ex)
{
    Console.WriteLine(ex.What());
}
```

*RYSUNEK 41 OBSŁUGA WYJĄTKU*

Dzięki słowu kluczowemu **throw** możliwe jest zgłoszenie wyjątku. Obiekt nowo utworzonego wyjątku obsługiwany jest przez blok **catch**, gdzie dodatkowo wyświetlany jest w terminalu komunikat o zaistniałym błędzie.

created by Michał Kopiel

Kopiel

---