

Zawartość:

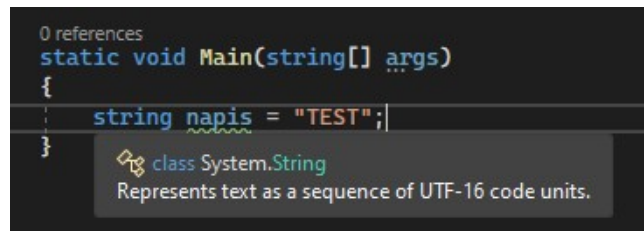
- klasy bazowe w C#
- hermetyzacja
- polimorfizm
- abstrakcja

Zaawansowane programowanie obiektowe w C#:

Programowanie obiektowe opiera się na klasach i obiektach.

Co to jest klasa?

Klasa to konstrukcja, która pozwala zdefiniować własny typ. Weźmy za przykład typ „string” z języka C#.



Jak widać na powyższym obrazku odpowiedź podpowiada nam, iż typ „String” znajduje się w przestrzeni nazw „System” oraz to, że jest to klasa.

Jeżeli klikniemy na typie „String” prawym przyciskiem myszy możemy za pomocą opcji „Go to definition” zobaczyć plik z definicją tegoż typu. Możemy tak zrobić na każdym typie, nie ważne czy my go zdefiniowaliśmy czy pochodzi on zestawu oferowanego przez język.

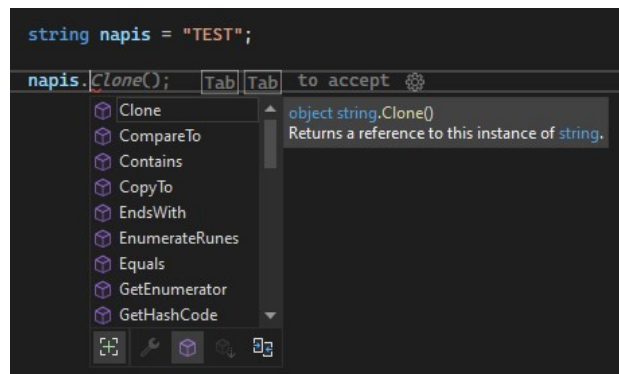
```
...public sealed class String : IEnumerable<char>, IEnumerable, ICloneable, IComparable,
{
    ...public static readonly String Empty;

    ...public String(char* value);
    ...public String(char[] value);
    ...public String(ReadOnlySpan<char> value);
    ...public String(sbyte* value);
    ...public String(char c, int count);
    ...public String(char* value, int startIndex, int length);
    ...public String(char[] value, int startIndex, int length);
    ...public String(sbyte* value, int startIndex, int length);
    ...public String(sbyte* value, int startIndex, int length, Encoding enc);

    ...public char this[int index] { get; }
    ...public int Length { get; }
```

```
...public object Clone();
...public int CompareTo(String? strB);
...public int CompareTo(object? value);
...public bool Contains(String value, StringComparison comparisonType);
...public bool Contains(char value, StringComparison comparisonType);
...public bool Contains(char value);
...public bool Contains(String value);
...public void CopyTo(int sourceIndex, char[] destination, int destinationIndex, int count);
...public bool EndsWith(char value);
...public bool EndsWith(String value);
...public bool EndsWith(String value, bool ignoreCase, CultureInfo? culture);
...public bool EndsWith(String value, StringComparison comparisonType);
...public StringRuneEnumerator EnumerateRunes();
...public override bool Equals(object? obj);
...public bool Equals(String? value);
...public bool Equals(String? value, StringComparison comparisonType);
...public CharEnumerator GetEnumerator();
...public int GetHashCode(StringComparison comparisonType);
...public override int GetHashCode();
...public ref readonly char GetPinnableReference();
```

Co tutaj widzimy? Parę przeciążonych konstruktorów, niezmiernie popularną właściwość Length (swoją drogą jest ona tylko getterem jak widać na obrazku) oraz wiele metod, które pojawiają się w podpowiedziach VS gdy operujemy na obiekcie klasy string.



Kliknięcie „Go to definition” na typie „int” również przenosi nas do jego definicji.

```
public readonly struct Int32 : IComparable, IComparable<Int32>, IConvertible, IEquatable<Int32>, IFormattable
{
    public const Int32 MaxValue = 2147483647;
    public const Int32 MinValue = -2147483648;

    public static Int32 Parse(string s, IFormatProvider? provider);
    public static Int32 Parse(string s, NumberStyles style, IFormatProvider? provider);
    public static Int32 Parse(string s);
    public static Int32 Parse(ReadOnlySpan<char> s, NumberStyles style = NumberStyles.Integer, IFormatProvider? provider = null);
    public static Int32 Parse(string s, NumberStyles style);
    public static bool TryParse(string? s, NumberStyles style, IFormatProvider? provider, out Int32 result);
    public static bool TryParse(ReadOnlySpan<char> s, NumberStyles style, IFormatProvider? provider, out Int32 result);
    public static bool TryParse(ReadOnlySpan<char> s, out Int32 result);
    public static bool TryParse(string? s, out Int32 result);
    public Int32 CompareTo(object? value);
    public Int32 CompareTo(Int32 value);
    public override bool Equals(object? obj);
    public bool Equals(Int32 obj);
    public override int GetHashCode();
    public TypeCode GetTypeCode();
    public override string ToString();
    public string ToString(IFormatProvider? provider);
    public string ToString(string? format);
    public string ToString(string? format, IFormatProvider? provider);
    public bool TryFormat(Span<char> destination, out Int32 charsWritten, ReadOnlySpan<char> format = default, IFormatProvider? provider = null);
}
```

Widzimy tu metody zwykłe (przypisane do każdego obiektu) oraz elementy statyczne. Widać słynną konstrukcję Int32.Parse(string s);

```
public readonly struct Boolean : IComparable, IComparable<Boolean>, IConvertible, IEquatable<Boolean>
{
    public static readonly string FalseString;
    public static readonly string TrueString;

    public static Boolean Parse(ReadOnlySpan<char> value);
    public static Boolean Parse(string value);
    public static Boolean TryParse(ReadOnlySpan<char> value, out Boolean result);
    public static Boolean TryParse(string? value, out Boolean result);
    public int CompareTo(Boolean value);
    public int CompareTo(object? obj);
    public Boolean Equals(Boolean obj);
    public override Boolean Equals(object? obj);
    public override int GetHashCode();
    public TypeCode GetTypeCode();
    public override string ToString();
    public string ToString(IFormatProvider? provider);
    public Boolean TryFormat(Span<char> destination, out int charsWritten);
}
```

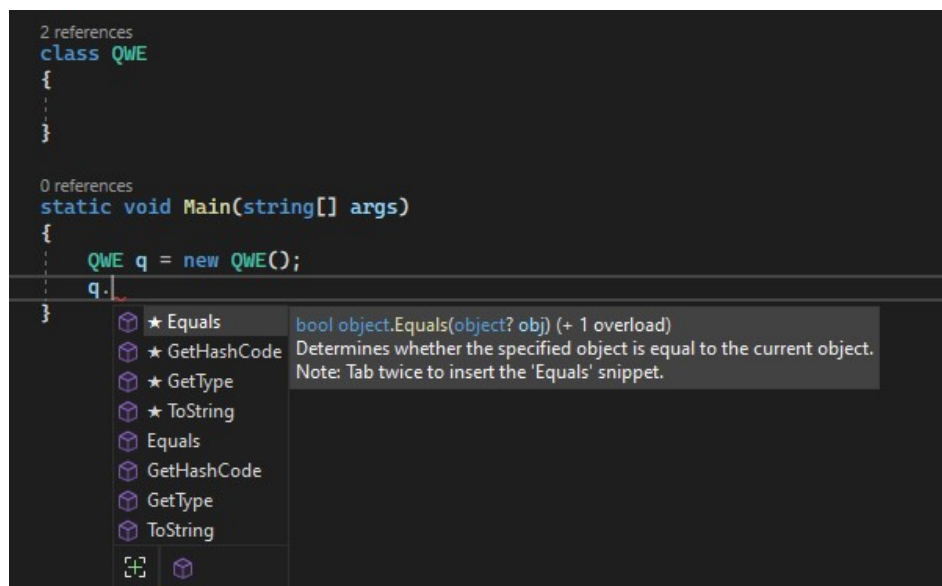
Na powyższych trzech przykładach widać pewną różnicę. String to klasa. Boolean oraz Int32 to struktura. Czemu? Trzeba się tutaj odnieść do typów wartościowych i referencyjnych. Int, Double, Float, Bool to typy wartościowe a jak wiadomo typy wartościowe są definiowane przez struktury. Typy referencyjne jak string przez klasy.

Hierarchia klas:

Jedną z właściwości programowania obiektowego jest dziedziczenie. Nie każdy jednak wie, że w c# każda klasa dziedziczy w sposób **NIEJAWNY** z klasy „Object”. Podobnie każda struktura w c# dziedziczy w sposób **NIEJAWNY** z klasy (tak klasy) „ValueType”.

Określenie „niejawny” bierze się z tego, że w nagłówku danej klasy czy struktury nie pisze się jawnie, że dziedziczy ona z „object” lub „ValueType”. Jak widać na powyższych obrazkach, w żadnym przypadku po nazwie klasy/struktury i dwukropku nie pada nazwa „object” czy „ValueType”.

Obie klasy dodają do klas/struktur kilka podstawowych metod z prostymi implementacjami:



Jak widać klasa QWE jest pusta a jednak VS podpowiada cztery virtualne (do napisania) metody:

- Equals() – metoda działa jak operator porównania, możemy ją dowolnie zaimplementować zależnie od tego jak chcemy porównywać nasze obiekty
- ToString() – nie jest to wbrew powszechnie błędnej opinii metoda konwertująca do typu string mimo, że w przypadku np. inta tak ją zaimplementowano, metoda ta służy do reprezentacji obiektu w formie tekstu
- GetType() – metoda zwracająca obiekt klasy Type, który przechowuje informacje o danym typie, przydatne dla mechanizmu refleksji
- GetHashCode() – zwraca identyfikator obiektu, który może zostać kiedyś użyty do identyfikacji obiektu w kolekcjach typu Dictionary czy HashSet.

Poniżej znajduje się budowa klasy „object” oraz „ValueType”.

```

...public class Object
{
    ...public Object();
    ...~Object();

    ...public static bool Equals(Object? objA, Object? objB);
    ...public static bool ReferenceEquals(Object? objA, Object? objB);
    ...public virtual bool Equals(Object? obj);
    ...public virtual int GetHashCode();
    ...public Type GetType();
    ...public virtual string? ToString();
    ...protected Object MemberwiseClone();
}

```

```

...public abstract class ValueType
{
    ...protected ValueType();

    ...public override bool Equals(object? obj);
    ...public override int GetHashCode();
    ...public override string? ToString();
}

```

Jako, że typy referencyjne są trudniejsze w porównywaniu i kopiowaniu klasa „object” dostarcza jakieś proste API i implementacje do wykonywania kopii oraz porównań zarówno wartości jak i samego adresu.

Przykładowa implementacja z metod, które często warto napisać po swojemu:

```

0 references
class Person
{
    1 reference
    public string Name { get; set; }
    1 reference
    public string Surname { get; set; }
    1 reference
    public int Age { get; set; }

    0 references
    public override string ToString()
    {
        return $"{Name} {Surname}: {Age}";
    }
}


```

Tak jak wspomniano wyżej metoda ToString() nie służy do konwersji obiektu w string. Zwraca ona reprezentację tekstową obiektu np. w celu wypisania go w konsoli.

```

Person person = new Person() { Name = "Adam", Surname = "Nowak", Age = 12 };
Console.WriteLine(person.ToString());

```



The screenshot shows the Visual Studio Debug Console with a single output line: "Adam Nowak: 12". The console window has a tab labeled "Microsoft Visual Studio Debug" and a close button (X). There are also plus (+) and minus (-) icons for window management.

```

4 references
class Person
{
    1 reference
    public string Name { get; set; }
    1 reference
    public string Surname { get; set; }
    3 references
    public int Age { get; set; }

    0 references
    public override bool Equals(object obj)
    {
        Person t = obj as Person;
        return t.Age == this.Age;
    }
}

```

Przykładowa implementacja metody Equals() wygląda tak. Przyjmujemy tutaj, że na potrzeby naszej logiki (naszej aplikacji) będziemy porównywać osoby względem ich wieku. Oczywiście jak wygląda porównanie to kwestia indywidualna dla każdego zdefiniowanego przez nas typu.

```

Person person1 = new Person() { Name = "Adam", Surname = "Nowak", Age = 13 };
Person person2 = new Person() { Name = "Michał", Surname = "Nowak", Age = 13 };
Console.WriteLine(person1.Equals(person2));
Console.WriteLine(person1==person2);

```

Microsoft Visual Studio Debug Console

True
False

Na powyższym obrazku widać dlaczego nie porównujemy obiektów typów referencyjnych za pomocą operatora ==. Porównał on adresy znajdujące się w zmiennych a są one oczywiście różne.

Hermetyzacja w C#:

Hermetyzacja w programowaniu obiektowym to niby najprostsza zasada. Jednak autorzy C# dodali trochę składniowego lukru, który skraca kod.

```
1 reference
class Person
{
    private string name;

    0 references
    public Person(string name)
    {
        this.name = name;
    }
}
```

Widzimy powyżej klasę, z jednym polem, które jest ukryte. Pole to ustawiane jest za pomocą konstruktora. Tradycyjna metoda hermetyzacji wyglądała by tak:

```
1 reference
class Person
{
    private string name;

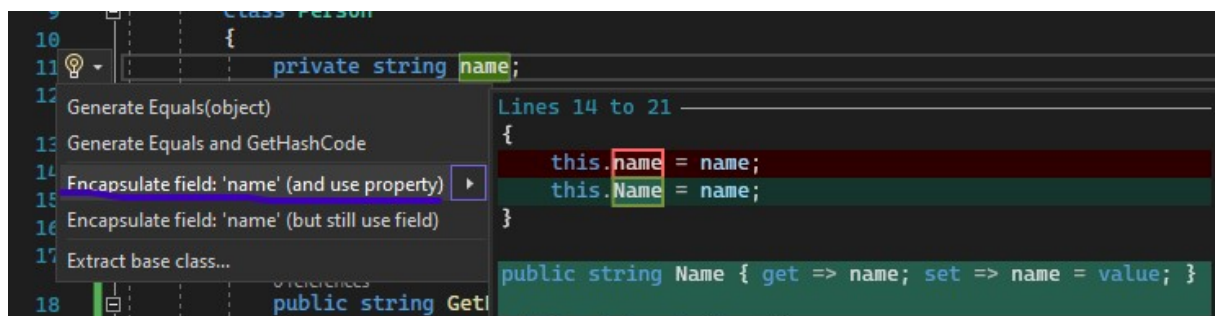
    0 references
    public Person(string name)
    {
        this.name = name;
    }

    0 references
    public string GetName()
    {
        return name;
    }

    0 references
    public void SetName(string name)
    {
        this.name = name;
    }
}
```

Standardowe podejście: metoda do pobierania wartości prywatnej i metoda do jej ustawiania (oczywiście pole może być read-only albo set-only co się w sumie rzadko zdarza).

C# dodaje jednak mechanizm właściwości:



The screenshot shows a code editor with a C# class `Person` containing a private field `name`. A context menu is open over the field, with the option `Encapsulate field: 'name' (and use property)` selected. To the right, a preview window titled 'Lines 14 to 21' shows the resulting code after refactoring: the private field `name` is replaced by a public property `Name` with `get` and `set` accessors. The `set` accessor is highlighted with a red box in the preview.

```
10 {
11     private string name;
12
13     Generate Equals(object)
14     Generate Equals and GetHashCode
15     Encapsulate field: 'name' (and use property)
16     Encapsulate field: 'name' (but still use field)
17     Extract base class...
18     public string GetI
```

```
Lines 14 to 21
{
    this.name = name;
    this.Name = name;
}

public string Name { get => name; set => name = value; }
```

Wystarczy zaznaczyć prywatne pole i kliknąć żarówkę lub skrót Ctrl + .

```

1 reference
class Person
{
    private string name;

    0 references
    public Person(string name)
    {
        this.Name = name;
    }

    1 reference
    public string Name { get => name; set => name = value; }
}

```

Oczywiście w tym przypadku możemy zostawić tylko get (właściwość będzie tylko getterem) lub tylko set (właściwość będzie tylko setterem).

Oczywiste jest także, że wyrażenia lambda tak jak i metody z przykładu wcześniej mogą wykonywać jakąś logikę a nie tylko zwracanie/ustawianie pola.

```

1 reference
class Person
{
    private string name;

    0 references
    public Person(string name)
    {
        this.name = name;
    }

    2 references
    public string Name
    {
        get
        {
            //some logic
            return this.name;
        }
        set
        {
            //some logic
            this.name = value;
        }
    }
}

```

Bardzo często właściwości w C# są mylone z polami. Mało tego, nawet oficjalne biblioteki tak robią. Można było by się kłócić, ale nie ma to sensu.

```

0 references
class Person
{
    public string name; //pole
    0 references
    public string Name { get; set; } //właściwość
}

```

Powyższa właściwość zachowuje się jako pole.

Polimorfizm:

Polimorfizm to chyba najtrudniejsza część programowania obiektowego, chociaż jak ktoś pisał np. aplikacje mobilne albo webowe za pomocą frameworków dostępnych dla konkretnych języków.

Polimorfizm pozwala przypisać obiekt klasy pochodnej do referencji (w innych językach wskaźnika) typu bazowego. Mało tego. Język jest w stanie zorientować się jakiego typu obiekt jest przypisany do tej referencji i uruchomić odpowiednią metodę.

Jest wiele przykładów użycia polimorfizmu:

- Co jeżeli chcemy zapisać kilka różnych typów wartości w jednej kolekcji
- Polimorfizm pozwala uniknąć sytuacji, w której przeciążamy funkcję n razy, żeby była ona dostępna dla każdego z n typów
- Polimorfizm jest stosowany we frameworkach o czym niżej

Przykład:

```
5 references
class Pracownik
{
    protected string name;

    2 references
    public Pracownik(string name)
    {
        this.name = name;
    }

    2 references
    public virtual string PrzedstawSie()
    {
        return $"Jestem {name}";
    }
}

1 reference
class Programista : Pracownik
{
    0 references
    public Programista(string name) : base(name)
    {
    }

    1 reference
    public override string PrzedstawSie()
    {
        return $"Jestem {name}. Pracuje jako programista";
    }
}

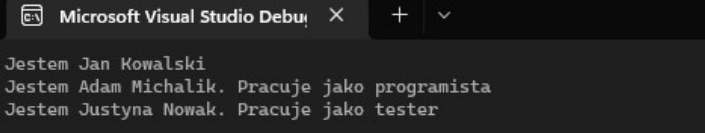
1 reference
class Tester : Pracownik
{
    0 references
    public Tester(string name) : base(name)
    {
    }

    1 reference
    public override string PrzedstawSie()
    {
        return $"Jestem {name}. Pracuje jako tester";
    }
}
```

Mamy tu klasę bazową Pracownik oraz dwie pochodne: Programista i Tester. Klasa bazowa implementuje metodę przedstaw się. Klasy pochodne nadpisują ją.

```
Pracownik pracownik = new Pracownik("Jan Kowalski");
Programista programista = new Programista("Adam Michalik");
Tester tester = new Tester("Justyna Nowak");

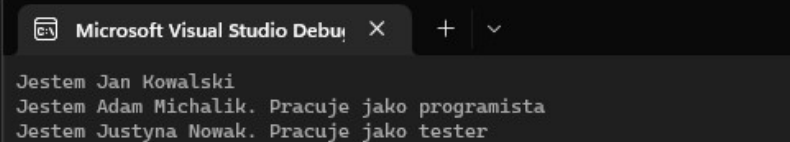
Console.WriteLine(pracownik.PrzedstawSie());
Console.WriteLine(programista.PrzedstawSie());
Console.WriteLine(tester.PrzedstawSie());
```



Tutaj sytuacja jest prosta. Klasy pochodne nadpisują swoją implementację bazową za pomocą swojej. Nie wykorzystujemy tutaj polimorfizmu. Ale zobaczmy ten przykład:

```
Pracownik pracownik = new Pracownik("Jan Kowalski");
Pracownik programista = new Programista("Adam Michalik");
Pracownik tester = new Tester("Justyna Nowak");


Console.WriteLine(pracownik.PrzedstawSie());
Console.WriteLine(programista.PrzedstawSie());
Console.WriteLine(tester.PrzedstawSie());
```



W każdym przypadku referencja jest typu Pracownik, ale metoda i tak wykonuje się ta co trzeba. Wszystko to dzięki polimorfizmowi i słowu kluczowemu „virtual”. Usuńmy je teraz i uruchommy powyższy program:

```
Pracownik pracownik = new Pracownik("Jan Kowalski");
Pracownik programista = new Programista("Adam Michalik");
Pracownik tester = new Tester("Justyna Nowak");

Console.WriteLine(pracownik.PrzedstawSie());
Console.WriteLine(programista.PrzedstawSie());
Console.WriteLine(tester.PrzedstawSie());
```



Teraz już tak fajnie nie jest. Co nam daje polimorfizm i słowo kluczowe „virtual”? **Najprościej zapamiętać, że gdy użyjemy virtual w metodzie klasy bazowej to metoda do wykonania dobierana jest na bazie typu, który jest po PRAWEJ stronie operatora new a nie po lewej.**

A teraz faktycznie po co nam to?

Przykład numer 1:

```
1 reference
static void PrzedstawPracownika(Pracownik p)
{
    Console.WriteLine(p.PrzedstawSie());
}

1 reference
static void PrzedstawProgramiste(Programista p)
{
    Console.WriteLine(p.PrzedstawSie());
}

1 reference
static void PrzedstawTestera(Tester p)
{
    Console.WriteLine(p.PrzedstawSie());
}

0 references
static void Main(string[] args)
{
    Pracownik pracownik = new Pracownik("Jan Kowalski");
    Programista programista = new Programista("Adam Michalik");
    Tester tester = new Tester("Justyna Nowak");

    PrzedstawPracownika(pracownik);
    PrzedstawProgramiste(programista);
    PrzedstawTestera(tester);
}
```

Musieliśmy przygotować trzy metody „PrzedstawX”. Dla każdego typu osobną, ponieważ różnią się one typem parametru. To jest sytuacja bez polimorfizmu. Zobaczmy teraz jak to wygląda z polimorfizmem:

```
3 references
static void PrzedstawPracownika(Pracownik p)
{
    Console.WriteLine(p.PrzedstawSie());
}

0 references
static void Main(string[] args)
{
    Pracownik pracownik = new Pracownik("Jan Kowalski");
    Programista programista = new Programista("Adam Michalik");
    Tester tester = new Tester("Justyna Nowak");

    PrzedstawPracownika(pracownik);
    PrzedstawPracownika(programista);
    PrzedstawPracownika(tester);
}
```

Mimo, że referencja jest typu Pracownik, zostanie wykonana dobra metoda „PrzedstawSie()” i wypisze się poprawny string.

Przykład numer 2:

Kontenery. Co podać w miejscu pytańników, żeby upchnąć w liście wszystkich trzech pracowników?

```
Pracownik pracownik = new Pracownik("Jan Kowalski");
Programista programista = new Programista("Adam Michalik");
Tester tester = new Tester("Justyna Nowak");

List<????> lista = new List<????>();
```

Należy podać typ bazowy Pracownik. Jednak należy pamiętać, że spowoduje to zmianę typu referencji na typ Pracownik.

```
Pracownik pracownik = new Pracownik("Jan Kowalski");
Programista programista = new Programista("Adam Michalik");
Tester tester = new Tester("Justyna Nowak");

List<Pracownik> lista = new List<Pracownik>() { pracownik, programista, tester};
lista.ForEach(t => Console.WriteLine(t.PrzedstawSie()));
```

Dzięki polimorfizmowi program, wie którą metodę wykonać w Foreach. Mimo, że każda zmienna dodana do listy została zamieniona na typ Pracownik.

Przykład numer 3:

Wspomniane frameworki.

```
public partial class InsightView : ContentPage
{
    1 reference
    public InsightView()
    {
        InitializeComponent();
        BindingContext = new InsightViewVM(Navigation);
    }

    0 references
    public override SizeRequest GetSizeRequest(double widthConstraint, double heightConstraint)
    {
        return base.GetSizeRequest(widthConstraint+20, heightConstraint+20);
    }
}
```

Przykład z Xamarina. Przeciążam metodę GetSizeRequest(). Jest to metoda wirtualna pochodząca z klasy ContentPage. W momencie uruchomienia środowisko wie, że ma wykonać metodę przeciążoną w klasie InsightView a nie tą z ContentPage mimo, że gdzieś pod spodem obiekt klasy InsightView jest trzymany w referencji ContentPage.

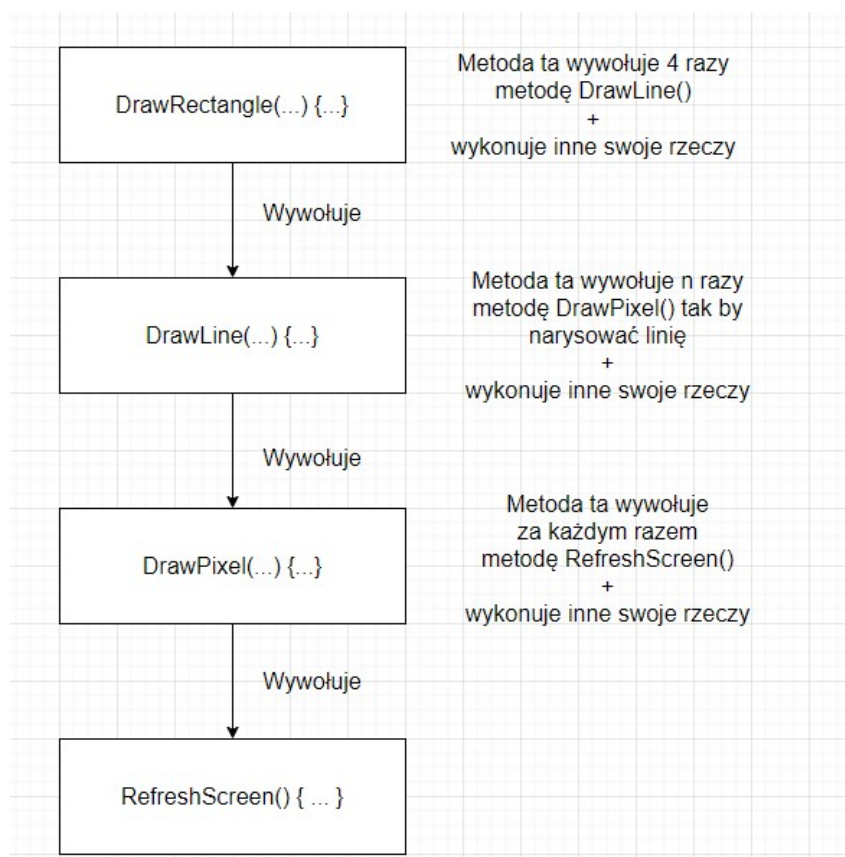
Takich przykładów jest multum.

Abstrakcja

Słynna abstrakcja. Co to jest ta abstrakcja? Jest to mega ogólne pojęcie. Najlepiej będzie to wytłumaczyć na przykładzie. Zobaczmy poniższy przykład kodu (na razie nie obiektowy):

```
void RefreshScreen(void);
void RefreshPartOfScreen(uint16_t sx, uint16_t sy, uint16_t ex, uint16_t ey);
void DrawPixel(uint16_t x, uint16_t y, COLORS color);
void DrawHorizontalLine(uint16_t x, uint16_t y, uint16_t len, COLORS color);
void DrawVerticalLine(uint16_t x, uint16_t y, uint16_t len, COLORS color);
void DrawLine(uint16_t sx, uint16_t sy, uint16_t ex, uint16_t ey, COLORS color);
void DrawWLine(uint16_t sx, uint16_t sy, uint16_t ex, uint16_t ey, uint16_t width, COLORS color);
void DrawRectangle(uint16_t sx, uint16_t sy, uint16_t ex, uint16_t ey, COLORS border_color, COLORS fill_color);
void DrawQuarterCircle(uint16_t x, uint16_t y, uint16_t r, uint8_t quarter, COLORS border_color, COLORS fill_color);
void DrawCircle(uint16_t x, uint16_t y, uint16_t r, COLORS border_color, COLORS fill_color);
void DrawWCircle(uint16_t x, uint16_t y, uint16_t r1, uint16_t r2, COLORS border_color, COLORS fill_color);
void DrawChar(uint16_t x, uint16_t y, char chr, uint8_t font_size, COLORS color, COLORS background);
void DrawString(int x, int y, const char* str, uint8_t font_size, COLORS color, COLORS background);
void ClearFullScreen(void);
void ClearPartScreen(uint16_t sx, uint16_t sy, uint16_t ex, uint16_t ey);
void FillScreen(COLORS color);
void DrawProgressBar(uint16_t sx, uint16_t sy, uint16_t ex, uint16_t ey, uint8_t percent, COLORS border_color, COLORS fill_color);
```

Mamy tu parę funkcji. Pozostają one w pewnej relacji:



Na rysunku widzimy kilka warstw abstrakcji. Wykonaliśmy warstwę abstrakcji, która odświeża ekran. Warstwę wyżej wykonaliśmy warstwę, która rysuje jeden pixel. Warstwa tak korzysta z elementów zaimplementowanych w warstwie niżej. I tak dalej i tak dalej. Proste?

Podobnie jest z językami.

Najpierw powstał assembler. W assemblerze napisali język C. Na bazie języka C powstał C#/Java/Python.

Teraz przykład obiektowy:

Wyobraźmy sobie, że mamy trzy klasy A, B i C. Każda z tych klas wykonuje jakieś swoje zadania. Na przykład klasa C wysyła wiadomości po sieci za pomocą jakiegoś protokołu. Klasa B służy do generowania wiadomości i przygotowaniu danych adresata dla klasy C. Klasa A zajmuje się obsługą Gui i zbiera dane dla klasy B.

Można więc powiedzieć, że klasa B jest zależnością klasy A. Ale również klasa C jest zależnością klasy B.

```
0 references
class A
{
    //jakieś funkcje które wywołują elementy B
    B b = new B();
}

2 references
class B
{
    //jakieś funkcje, które wywołują elementy C
    C c = new C();
}

2 references
class C
{
    //jakaś logika
}
```

Widzimy tu podział na trzy warstwy abstrakcji.

PS. Fajnie było by w takim przypadku stosować wstrzykiwanie zależności, ale to tylko przykład.