Politechnika Wrocławska

Programowanie obiektowe

Wykład 10

Delegaty

Funkcje Anonimowe

Wyjątki i ich obsługa

Prowadzący
dr inż. Paweł Maślak

Plan wykładu

- Delegaty
- Funkcje Anonimowe
- Wyjątki
- Obsługa wyjątków

Delegaty

- **Delegaty** to obiekty, które wskazują na dowolne metody znajdujące się w programie i definiowane są za pomocą słowa kluczowego ***delegate***.
- Po co są właściwie potrzebne delegaty?
- Problem używania wskaźników na funkcje istnieje od zawsze jeszcze od czasów świetności języków C/C++. Projektując różne rozbudowane systemy istniała potrzeba, aby funkcja wywoływana mogła komunikować się z funkcją wywołującą. Wskaźniki na funkcje mogą być argumentami metod, dzięki temu do funkcji można przesłać adresy różnych funkcji zależnie od wymaganej sytuacji lub wyboru użytkownika.

Delegaty

- Temat delegatów jest ściśle związany z tematem funkcji wywołań zwrotnych (ang. callbacks), które są po prostu wskaźnikami na funkcję. Ideologia używania callbacków jest odwrotną do ideologii używania zwykłych funkcji, ponieważ posiadając dowolną metodę, programista zajmuje się jej wywołaniem względem instancji danej klasy.
- Dzięki użyciu callbacków sytuacja się odwraca. Programista nie wywołuje funkcji manualnie, a jedynie definiuje funkcję o odpowiedniej nazwie i parametrach, do późniejszego wywołania przez inne elementy systemu niezależne od naszego programu. Jest to bardzo elastyczne rozwiązanie.

Delegaty

- Delegaty są obiektami dziedziczącymi bezpośrednio niejawnie z ***MulticastDelegate***. Ta z kolei dziedziczy z klasy ***Delegate*** oraz ***Object***. Deklarując nowy delegat kompilator automatycznie przekształca go na klasę, która jest automatycznie zapieczętowana (ang. *sealed*) a więc nie można z niej dziedziczyć.
  Automatycznie są także generowane metody:
  - ***Invoke***() – wywołuje metody podpięte do delegata (jedną lub listę metod). Przyjmuje takie same argumenty jak metoda związana z delegatem. Działa synchronicznie względem głównego wątku programu.
  - ***BeginInvoke()*** – wywołuje metody związane z delegatem w sposób asynchroniczny. Wywołana metoda zostaje wykonana w osobnym wątku.

Tworzenie delegatów

- Delegat wskazujący na dowolne metody, możemy stworzyć bardzo łatwo. W tym celu należy posłużyć się słowem kluczowym ***delegate***. Aby delegat mógł wskazywać na metodę, musi posiadać taki sam typ zwracany oraz takie same argumenty. Przykładowy delegat:

```
public delegate int Action(int x, int y);
```

- Delegat ten może wskazywać na dowolną funkcję przyjmującą dwa argumenty typu int oraz zwracającą wartość typu int. Mając zadeklarowany delegat, możemy podpiąć do niego dowolną metodę. Ważne jest to, aby zgodne były argumenty oraz typ zwracany.

Tworzenie delegatów

- Delegaty w C# obsługują tzw. **multicasting**. Dzięki temu możliwe jest podpięcie wielu metod do jednego delegata.
- Podpinanie metod pod delegat odbywa się za pomocą:
  - pierwsza metoda podpinana jest poprzez konstruktor, podczas tworzenia instancji delegata
  - każdą następną metodę możemy dodać/usunąć za pomocą operatorów += oraz -=

Delegaty - przykład

- Użycie delegata Action()

```
using System;
public delegate void Action(int x, int y);

public class Math
{
    public void Add(int a, int b)
    {
        Console.WriteLine(a + b);
    }
    public void Subtract(int a, int b)
    {
        Console.WriteLine(a - b);
    }
}
```

```
class Program
{
    static void Main()
    {
        Math math = new Math();

        Action action = new Action(math.Add);
        // wywoła math.Add(7,5)
        action(7,5);
        action += math.Subtract;
        // wywoła math.Add(3,4)
        // wywoła math.Subtract(3,4)
        action(3, 4);
        action.Invoke(3,4);
        // to samo, ale z metodą Invoke()
    }
}
```

Delegaty jako funkcje wywołań zwrotnych

- W dużej ilości przypadków delegaty używane są do tworzenia funkcji zwrotnych, czyli ***callbacków***. Deklaracja delegata wewnątrz jakiejkolwiek klasy zostanie niejawnie rozwinięta przez kompilator do postaci klasy zagnieżdżonej.
- ***Callback*** Jest to funkcja zwrotna, której idea używania jest przeciwna do standardowego używania funkcji. Zamiast wywoływać interesujące nas funkcje, nasz obiekt wywołuje delegat. To na co będzie wskazywał delegat w przyszłości, zależy już od innych obiektów i programistów pracujących na naszej klasie.

Delegaty jako funkcje wywołań zwrotnych

- Umieszczając delegat wewnątrz jakiejś klasy trzymajmy się następujących zasad:
- definicja delegata musi być publiczna – zostanie niejawnie rozwinięta do definicji zagnieżdżonej klasy
- zgodnie z zasadą hermetyzacji, zmienna delegata musi być prywatna, obudowana „setterem” w celu zapewnienia dostępu z zewnątrz

System logowania zaimplementowany za pomocą callbacków

```
using System;
public class SignInSystem
{
   public delegate void Logs(string message);
   private Logs sendLogs;

   public void AddCallback(Logs function)
   {
      sendLogs += function;
   }

   public void DeleteCallback(Logs function)
   {
      sendLogs += function;
   }

   public bool SignIn(string user, string
password)
   {
      sendLogs("Logowanie użytkownika: " +
user);
      return true;
   }
}

class Program
{
   static void Main()
   {
      SignInSystem system = new SignInSystem();
      system.AddCallback(CallbackLogs);
      system.SignIn("user", "password");
      Console.ReadKey();
   }

   static void CallbackLogs(string message)
   {
      Console.WriteLine(message);
      Console.WriteLine("Wywołano funkcję
zwrotną!");
   }
}
```

Użycie delegatów jako parametrów funkcji

```
using System;
using System.Collections;

public class Numbers
{
   public delegate void ActionDelegate(ref ArrayList
list);
   ArrayList numbersList = new ArrayList();

   public Numbers(params int[] numbers)
   {
      numbersList.AddRange(numbers);
   }

   public void Action(ActionDelegate action)
   {
      action(ref numbersList);
   }
}

public static class Action
{
   public static void Print(ref ArrayList list)
   {
      foreach (var i in list)
      {
         Console.Write(i);
      }
      Console.WriteLine();
   }
   public static void Reverse(ref ArrayList list)
   {
      list.Reverse();
   }
}

class Program
{
   static void Main()
   {
      Numbers numbers = new Numbers(1, 2, 3, 4, 5,
6);
      numbers.Action(Action.Print);
      numbers.Action(Action.Reverse);
      numbers.Action(Action.Print);
      Console.ReadKey();
   }
}
```

Komentarz do slajdu wcześniej

- Klasa ***Numbers***, która wbrew pozorom nie udostępnia żadnej funkcjonalności i wygląda bardziej jak niestandardowa kolekcja obiektów, to dzięki callback'om i delegatom można bardzo rozszerzać jej funkcjonalność.
- Dzieje się tak dlatego, że programista tworzący klasę ***Numbers*** pomyślał, że warto będzie zaimplementować w niej callback o nazwie Operacja. Wywołuje ona funkcję zwrotną z referencją do prywatnej kolekcji, do której nie dałoby się dostać winny sposób.

Wbudowane delegaty – ***Func***

- Delegat ***Func*** jest delegatem generycznym, podczas jego deklaracji należy określić typ zwracany oraz typ jego parametrów. Należy pamiętać, że w nawiasach ostrych (< >) końcowy parametr jest zawsze tym zwracanym, a wszystkie jakie dopiszemy wcześniej będą wejściowymi, na przykład:

```
Func<out> // brak parametrów wejściowych
Func<in, out> // jeden parametr wejściowy
Func<in, in, out> // dwa parametry wejściowe
Func<in, in, in, out> // trzy parametry wejściowe
```

Przykład użycia delegata *Func*

- Podpięcie funkcji Add() do delegata generycznego Func:

```
public class Math
{
    public int Add(int a, int b)
    {
        return a + b;
    }
}
class Program
{
    static void Main()
    {
        Math m = new Math();
        Func<int, int, int> dodawanie = m.Add;
        Console.WriteLine(dodawanie(5, 2));
    }
}
```

Wbudowane delegaty – ***Action***

- Jego działanie jest identyczne, jednak używamy go w przypadku procedur. Definicja formalna może wyglądać następująco:
- `public delegate void Action<in T>(T arg);`
- Jedyna różnica występuje podczas określania parametru generycznego. Wszystkie typy wymienione w kwadratowych dzióbkach będą parametrami.

Przykład użycia delegata *Action*

- Przykład delegata Action, który wskazuje na funkcje Console.WriteLine()

```
class Program
{
   static void Main()
   {
      Action<string> print = Console.WriteLine;
      print("Something...");
   }
}
```

- W powyższym przykładzie utworzono delegat o nazwie **print**, który wskazuje na funkcję **Console.WriteLine**() i określony został dla niego jeden argument typu **string**.

Funkcje anonimowe oraz wyrażenia lambda

- Nie wszystkie funkcje są tak ważne, że muszą mieć nazwę. Wyrażenia lambda nie funkcjonują na poziomie klas i dlatego nie posiadają nazw. Referencje do takich funkcji zazwyczaj przetrzymuje się w postaci zmiennych typu danej delegaty.
- Istnieją pewne ograniczenia dotyczące tego, gdzie anonimowe funkcje mogą się znajdować i jakie mają być. Na przykład nie mogą one być generyczne. Poza tym anonimowe funkcje mogą robić to samo, co normalne metody.

Funkcje anonimowe

- Nie trzeba określać typu zwracanego w metodzie anonimowej, wynika to z instrukcji return wewnątrz bloku kodu. Metody anonimowe deklarowane są z wykorzystaniem instancji delegata, ze słowem kluczowym delegate.

Funkcje anonimowe

- Przykład funkcji anonimowej **print**, utworzonej w oparciu o delegat **PrintNumber**

- ```
  class Program
  {
      delegate void PrintNumber(int n);

      static void Main()
      {
          PrintNumber print = delegate (int i)
          {
              Console.WriteLine("Liczba to: {0}", i);
          };
          print(6);
      }
  }
  ```
  ```

Wyrażenia Lambda

- Od wersji 3.0 języka C# wprowadzono wyrażenia ***Lambda***, będące uproszczeniem funkcji anonimowych. Takie wyrażenie jest krótsze, ponieważ nie ma potrzeby pisać słowa kluczowego **delegate** i całe ciało zostało sformatowane do jednej linijki. Wyrażenie lambda określa się operatorem **=>**. Po lewej stronie umieszczony jest parametr (gdy jest więcej parametrów to umieszcza się je w nawiasach), a po prawej ciało operacji.

Wyrażenia Lambda - przykład

- Wyrażenia lambda służące do wykonywania operacji matematycznych – dodawania, mnożenia i potęgowania

```
delegate int Action(int n, int m);
static void Main()
{
    Action add = (n, m) => { return n + m;
    };
    Console.WriteLine(add(1, 2));

    Action multiply = (n, m) => { return n * m; };
    Console.WriteLine(multiply(3, 2));

    Action pow = (n, m) => {
        var result = 1;
        for (int i = 1; i<=m; i++)
        {
            result *= n;
        }
        return result;
    };
    Console.WriteLine(pow(3, 2));
}
```

Wyrażenia Lambda - przykład

- W tym przykładzie utworzony został delegat **Action** z parametrami typu **int**, a co za tym idzie, musi on też zwracać wartość tego typu. Na podstawie tego utworzonego delegata, zaimplementowano trzy wyrażenia lambda, które wykonują pewne operacje matematyczne, takie jak dodawanie, mnożenie i podniesienie liczby n do m-tej potęgi. Można zauważyć że, wyrażenie lambda o nazwie pow, jest trochę bardziej rozbudowane, w porównaniu z pozostałymi - „jednolinijkowcami”.
- Istnieje zasada, żeby kod w ciele wyrażenia lambda, „kręcił się” wokół wartości zwracanej.

Wyjątki

- Każdy program musi mieć możliwość obsługi błędów występujących podczas wykonywania w spójny sposób. Język C# i platforma .NET udostępniają model do powiadamiania programu i wskazują na niepowodzenie przez zgłaszanie wyjątków.
- Wyjątek to dowolny warunek błędu lub nieoczekiwane zachowanie, które występuje przez program wykonujący. Mogą one być zgłaszane z powodu błędu w kodzie lub bibliotece lub innych nieoczekiwanych warunkach napotkanych przez środowisko uruchomieniowe.

Wyjątki

- W języku C# wyjątkiem jest obiekt dziedziczony po klasie ***System.Exception*** i jest on zgłaszany z obszaru kodu, w którym wystąpił problem.
- Tradycyjny model obsługi błędów polegał na unikatowym sposobie ich wykrywania i lokalizowania dla nich programów obsługi lub na mechanizmie obsługi błędów dostarczonym przez system operacyjny.

Wyjątki - **IndexOutOfRange**

- Jest on zgłaszany przez środowisko uruchomieniowe tylko wtedy, gdy tablica jest indeksowana nieprawidłowo, czyli indeks, do którego chcemy się dostać znajduje się poza prawidłowym zakresem tablicy

```
1   using System;
2
3   namespace Wyk10_3
4   {
5       internal class Program
6       {
7           public static void Main(string[] args)
8           {
9               int[] arr = { 1, 2, 3, 4, 5, 6, 7, 8 };
10              Console.WriteLine(arr[arr.Length + 1]);
11          }
12      }
13  }
```

```
Wyjątek nieobsłużony: System.IndexOutOfRangeException: Indeks wy
kraczał poza granice tablicy.
   w Wyk10_3.Program.Main(String[] args) w H:\zajecia\Programowa
nie_obiektowe\Przykłady\Wyk10\Wyk10_3\Program.cs:wiersz 10

Process finished with exit code -532,462,766.
```

Wyjątki - **NullReferenceException**

- Jest zgłaszany przez środowisko uruchomieniowe wtedy, gdy obiekt o wartości null jest wywoływany
-
```
namespace wyk10
{
    internal class Wyk10_4
    {
        static void Main()
        {
            object o = null;
            String s = o.ToString();
            Console.WriteLine(s);
        }
    }
}
```

```
Unhandled exception. System.NullReferenceExcepti
on: Object reference not set to an instance of a
n object.
   at wyk10.Wyk10_4.Main() in C:\Users\Pawel\Des
ktop\ProgramowanieObiektowe\Laboratoria_2024\Prz
yklady\GR1\wyk10\wyk10\Program.cs:line 10

Process finished with exit code -1,073,741,819.
```

Wyjątki - **ArgumentNullException**

- Jest zgłaszany przez metody, które nie zezwalają na argument ma wartość **null**
-
```
namespace wyk10
{
    internal class Wyk10_5
    {
        static void Main()
        {
            String s = null;
            Console.WriteLine("calculate".IndexOf(s));
        }
    }
}
```

Wyjątki - **ArgumentNullException**

- ```
  internal class Wyk10_5
  {
      static void Main()
      {
          String s = null;
          Console.WriteLine("calculate".IndexOf(s));
      }
  }
  ```
- Zadaniem metody IndexOf() jest zwrócenie indeksu początku pierwszego wystąpienia frazy podanej jako parametr. W przypadku gdy ten parametr jest wartości null, zgłaszany jest wyjątek ArgumentNullException.
  ```

Wyjątki - **ArgumentNullException**

- Jest zgłaszany przez metody, które nie zezwalają na argument ma wartość **null**
-
```
namespace wyk10
{
    internal class Wyk10_5
    {
        static void Main()
        {
            String s = null;
            Console.WriteLine(
        }
    }
}
```

```
Unhandled exception. System.ArgumentNullException: Va
lue cannot be null. (Parameter 'value')
   at System.Globalization.CompareInfo.IndexOf(String
 source, String value, Int32 startIndex, Int32 count,
 CompareOptions options)
   at System.String.IndexOf(String value, Int32 start
Index, Int32 count, StringComparison comparisonType)
   at System.String.IndexOf(String value)
   at wyk10.Wyk10_5.Main() in C:\Users\Pawel\Desktop\
ProgramowanieObiektowe\Laboratoria_2024\Przyklady\GR1
\wyk10\Wyk10_5\Program.cs:line 8

Process finished with exit code -532,462,766.
```

Wyjątki - **ArgumentNullException**

- Wywołanie funkcji IndexOf() nie prowadzące do otrzymania wyjątku

```
namespace wyk10
{
   internal class Wyk10_5
   {
      static void Main()
      {
         String s = "late";
         Console.WriteLine("calculate".IndexOf(s));
      }
   }
}
```

Wyjątki - **ArgumentOutOfRangeException**

- Jest wyrzucany przez metody, które sprawdzają, czy argumenty znajdują się w danym zakresie.
-
```
namespace wyk10
{
   internal class Wyk10_6
   {
      static void Main()
      {
         String s = "late";
         s.Substring(s.Length + 1);
         Console.WriteLine(s);
      }
   }
}
```

Wyjątki - **ArgumentOutOfRangeException**

- Jest wyrzucany przez metody, które sprawdzają, czy argumenty znajdują się w danym zakresie.
-
```
namespace wyk10
{
    internal class Wyk10_6
    {
        static void Main()
        {
            String s = "late";
            s.Substring(s.Length + 1);
            Console.WriteLine(s);
        }
    }
}
```

```
Unhandled exception. System.ArgumentOutOfRangeExcepti
on: startIndex cannot be larger than length of string
. (Parameter 'startIndex')
   at System.String.ThrowSubstringArgumentOutOfRange(
Int32 startIndex, Int32 length)
   at System.String.Substring(Int32 startIndex)
   at wyk10.Wyk10_5.Main() in C:\Users\Pawel\Desktop\
ProgramowanieObiektowe\Laboratoria_2024\Przyklady\GR1
\wyk10\Wyk10_5\Program.cs:line 8

Process finished with exit code -532,462,766.
```

Wyjątki - **ArgumentOutOfRangeException**

- Jest wyrzucany przez metody, które sprawdzają, czy argumenty znajdują się w danym zakresie.
- ```
  namespace wyk10
  {
     internal class Wyk10_6
     {
        static void Main()
        {
           String s = "late";
           s.Substring(s.Length + 1);
           Console.WriteLine(s);
        }
     }
  }
  ```
- Wyjątek ArgumentOutOfRangeException spowodowany wywołaniem metody Substring() z wartością parametru większą niż rozmiar łańcucha tekstowego
  ```

Wyjątki - **ArgumentOutOfRangeException**

- Jest wyrzucany przez metody, które sprawdzają, czy argumenty znajdują się w danym zakresie.
- ```
  namespace wyk10
  {
     internal class Wyk10_6
     {
        static void Main()
        {
           String s = "late";
           s.Substring(s.Length + 1);
           Console.WriteLine(s);
        }
     }
  }
  ```
- Wyjątek ArgumentOutOfRangeException spowodowany wywołaniem metody Substring() z wartością parametru większą niż rozmiar łańcucha tekstowego
  ```

Wyjątki - **DivisionByZeroException**

- Wyjątek **DivideByZeroException** dzielenie wartości zmiennej „i” przez 0, co jest niedozwolone w matematyce
- ```
  internal class Wyk10_6
  {
      static void Main()
      {
          int i = 1000;
          Console.WriteLine(i/0);
      }
  }
  ```
  ```

Wyjątki - **DivisionByZeroException**

- Wyjątek **DivideByZeroException** dzielenie wartości zmiennej „i” przez 0, co jest niedozwolone w matematyce
- ```
  internal class Wyk10_6
  {
      static void Main()
      {
          int i = 1000;
          Console.WriteLine(i/0);
      }
  }
  ```

```
Unhandled exception. System.DivideByZeroException: At
tempted to divide by zero.
   at wyk10.Wyk10_7.Main() in C:\Users\Pawel\Desktop\
ProgramowanieObiektowe\Laboratoria_2024\Przyklady\GR1
\wyk10\Wyk10_5\Program.cs:line 8

Process finished with exit code -1,073,741,676.
```
```

Wyjątki - **DivisionByZeroException**

- ```
  internal class Wyk10_6
  {
      static void Main()
      {
          int i = 1000;
          Console.WriteLine(i/0);
      }
  }
  ```
- Jako argument funkcji Console.WriteLine() podano operację matematyczną, która jest niedozwolona, bo nie da się dzielić przez 0. Tak więc wyjątek ten został zgłoszony właśnie dlatego. W tym przypadku doprowadzono do tego celowo, ale zdarza się to czasami przypadkiem, na przykład poprzez dzielenie w pętli.
  ```

Obsługa wyjątków – blok try, catch, finally

- Funkcje obsługi wyjątków języka C# ułatwiają radzenie sobie z nieoczekiwanymi lub wyjątkowymi sytuacjami, które występują, gdy program jest uruchomiony.
- Obsługa wyjątków w języku C# używa trzech słów kluczowych: try, catch oraz finally. Służą one do wypróbowania akcji, które mogą się nie powieść i do obsługi ich błędów, gdy programista zdecyduje, że jest to uzasadnione.
- Wyjątki mogą być generowane przez środowisko uruchomieniowe lub tworzone przy użyciu słowa kluczowego throw.

Obsługa wyjątków – blok try, catch, finally

- W wielu przypadkach wyjątek może zostać zgłoszony nie przez metodę wywoływaną bezpośrednio przez kod, ale przez inną metodę, która jest umieszczona dalej na stosie wywołań.
- Gdy zgłaszany jest wyjątek, na stosie wywołań zostaje wyszukana metoda z blokiem catch dla tego określonego, zgłaszanego typu wyjątku i wykonany zostaje pierwszy taki blok catch, który zostanie znaleziony. Jeśli niestety, nie znajdzie się odpowiedni blok catch w dowolnym miejscu w stosie wywołań, zakończy się proces i wyświetli użytkownikowi stosowny komunikat.

Obsługa wyjątków – przykład

- Obsługa wyjątku DivideByZeroException, za pomocą bloku try … catch

```
using System;
class Wyk10_3
{
static double Division(double x, double y)
{
   if (y == 0)
   {
      throw new DivideByZeroException();
   }
   return x / y;
}

public static void Main()
{
   double a = 14, b = 0;
   double result;
    try
   {
      result = Division(a, b);
      Console.WriteLine("{0} : {1} = {2}", a,
         b, result);
   }
   catch (DivideByZeroException)
   {
      Console.WriteLine("Niedozwolone
dzielenie przez 0!");
   }
}
}
```

Obsługa wyjątków – przykład

- Wyjątki mogą być generowane jawnie przez program przy użyciu słowa kluczowego throw, w taki sposób jak na slajdzie wczesniej. W obsłudze wyjątków istnieje jeszcze jeden blok, który również jest bardzo często używany – finally. Kod umieszczony w bloku finally jest wykonywany niezależnie od tego, czy jest zgłaszany wyjątek.

Obsługa wyjątków – przykład

- Obsługa wyjątku DivideByZeroException, za pomocą bloku try … catch… finally

```
using System;
class Wyk10_4
{

   static double Division(double x, double y)
   {
      if (y == 0)
      {
         throw new DivideByZeroException();
      }

      return x / y;
   }

   public static void Main()
   {
      double a = 14, b = 0;
      double result;

      try
      {
         result = Division(a, b);
         Console.WriteLine("{0} : {1} = {2}", a, b,
result);
      }
      catch (DivideByZeroException)
      {
         Console.WriteLine("Niedozwolone dzielenie
przez 0!");
      }
      finally
      {
         Console.WriteLine("Koniec Programu");
      }
   }
```

Obsługa wyjątków – przykład

- Obsługa wyjątku DivideByZeroException, za pomocą bloku try … catch… finally

```
using System;
class Wyk10_4
{

   static double Division(double x, double y)
   {
      if (y == 0)
      {
         throw new DivideByZeroException();
      }

      return x / y;
   }

   public static void Main()
   {
      double a = 14, b = 7;
      double result;

      try
      {
         result = Division(a, b);
         Console.WriteLine("{0} : {1} = {2}", a, b,
result);
      }
      catch (DivideByZeroException)
      {
         Console.WriteLine("Niedozwolone dzielenie
przez 0!");
      }
      finally
      {
         Console.WriteLine("Koniec Programu");
      }
   }
```

Obsługa wyjątków – podsumowanie

- Pamiętaj na przyszłość, że w tej samej instrukcji try – catch można użyć więcej niż jednej konkretnej klauzuli catch. W takim przypadku kolejność klauzul catch jest ważna, ponieważ są one badane po kolei. Należy więc przechwycić bardziej szczegółowe wyjątki przed mniej konkretnymi wyjątkami.

Politechnika Wrocławska

Dziękuje bardzo za uwagę

Dr inż. Paweł Maślak