Zawartość

- delegaty C#: Func, Action, Predicate

- zastosowanie delegatów

Delegaty w C#

Delegat – jest to pewien typ danych przeznaczony do przechowywania referencji na funkcję/metodę. Oczywiście jest to całkiem duże uproszczenie.

Rodzaje:

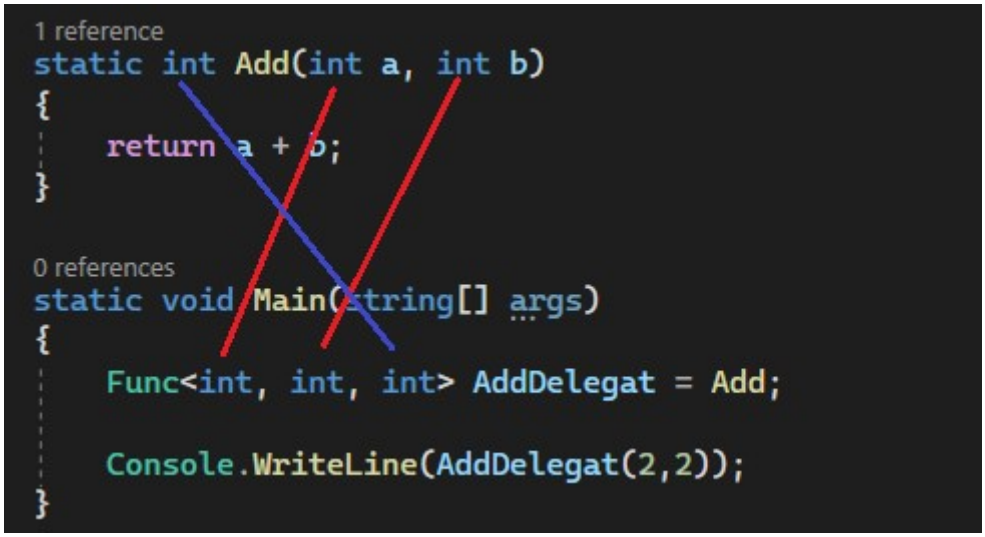
- **Func<T1, T2, T3,…,TR>** - delegat przeznaczony do przechowywania funkcji, które coś zwracają (mają jakikolwiek inny typ zwracany niż **void**)
- **Action<T1, T2, T3,…>** - delegat przeznaczony do przechowywania funkcji, które nie zwracają (mają typ zwracany ustawiony na **void**)
- **Predicate<T1>** - delegat przeznaczony do przechowywania funkcji, które zwracają typ **bool**

Gdzie:

T1, T2, T3 – typy kolejnych argumentów funkcji

TR – typ zwracany (ma zastosowanie tylko w przypadku delegata Func)

Kilka przykładów powinno rozjaśnić sytuację:



Tworzymy zmienną typu Func. Precyzujemy, że można do niej przypisać referencję na funkcję, która przyjmuje dokładnie dwa parametry (oba typu int) oraz zwraca typ int.

Przypisanie funkcji, która nie pasuje skończy się błędem kompilacji.

Na obrazku widać także wywołanie delegata.

Inny przykład

```
1 reference
static string Substring(string t, int s, int e)
{
    return t.Substring(s, e-s);
}

0 references
static void Main(string[] args)
{
    Func<string, int, int, string> SubStrD = Substring;

    string result = SubStrD("Ala ma kota", 0, 3);
    Console.WriteLine(result);
}
```

Tworzymy delegata typu Func, który przyjmuje odpowiednio stringa oraz dwa inty. Zwraca natomiast typ string. Przypisujemy do niego funkcję, która pasuje do założeń.

Kolejny przykład:

```
Func<string, int, int, string> SubStrD = (t, s, e) => {
    return t.Substring(s, e - s);
};
```

Do delegatów naturalnie przypisywać można lambdy (paradoksalnie to się robi najczęściej nawet o tym nie wiedząc, ale o tym niżej).

```
Func<double> GetPi = () =>
{
    return 3.14;
};
```

Ten delegat przechowuje funkcję, która przyjmuje dokładnie zero argumentów. Double odnosi się do typu zwracanego.

```
1 reference
static double Pi()
{
    return 3.14;
}

0 references
static void Main(string[] args)
{
    Func<double> GetPi = Pi;
}
```

To samo co wyżej tylko w innej formie zapisu docelowej funkcji.

```
1 reference
static void Print(string message)
{
    Console.WriteLine(message);
}

0 references
static void Main(string[] args)
{
    Action<string> PrintD = Print;

    PrintD("Ala ma kota!");
}
```

Przykład delegatu Action stosowanego do metod, które nic nie zwracają.

```
0 references
static void Main(string[] args)
{
    Action<string> PrintD = (m) => Console.WriteLine(m);

    PrintD("Ala ma kota!");
}
```

To samo tylko, krócej zapisane za pomocą lambdy.

```
1 reference
static void Print()
{
    Console.WriteLine("Print");
}

0 references
static void Main(string[] args)
{
    Action PrintD = Print;
}
```

Action może nie przyjmować żadnego parametru. Tutaj akurat użyteczność takiego kodu jest mała, ale istnieje wiele casów gdzie takie coś się przyda.

Przykład dla Predicate

Predicate to typ delegatu, który zawsze zwraca bool i dla tego nie trzeba precyzować typu zwracanego. **To samo można uzyskać za pomocą Func podając typ zwracany jako bool tak więc:**

Func<int, bool> == Predicate<int>

```
1 reference
static bool isEven(int a)
{
    return !(a % 2 == 0);
}

0 references
static void Main(string[] args)
{
    Predicate<int> EvenP = isEven;
}
```

To samo uzyskamy tak:

```
1 reference
static bool isEven(int a)
{
    return !(a % 2 == 0);
}

0 references
static void Main(string[] args)
{
    Func<int, bool> EvenP = isEven;
}
```

Przykład z bardziej rozbudowanymi typami:

```
1 reference
static bool Equal(Point p, Point q)
{
    return p.Equals(q);
}

0 references
static void Main(string[] args)
{
    Func<Point, Point, bool> IsEqual = Equal;
}
```

Predicate oczekuje dokładnie jednego parametru generycznego (w praktyce oznacza to jeden argument funkcji). Co oznacza, że powyższy przykład mimo, że zwraca bool nie może być zapisany w formie Predicate. Trzeba użyć Func.

Typ Point zdefiniowałem jako klasę wyżej. Implementacja nie istotna dla przykładu.

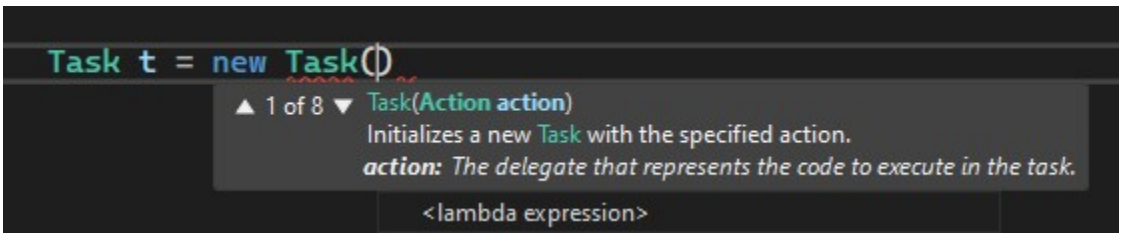
Po co te delegaty?

Zastosowanie:

- delegat to typ zmiennej, do której możemy przypisać funkcję więc
    - jeżeli chcemy zapisać funkcję w polu klasy to musimy posłużyć się delegatem
    - jeżeli chcemy podać funkcję jako parametr innej funkcji to musimy posłużyć się delegatem

Jeżeli ktoś podawał np. lambdy w parametrze funkcji w innych językach to wie o co chodzi. C# jest statycznie typowany, więc te parametry/zmienne muszą mieć jakiś typ.

Wyobraźmy sobie, że chcemy otworzyć nowy wątek w programie za pomocą klasy Task.



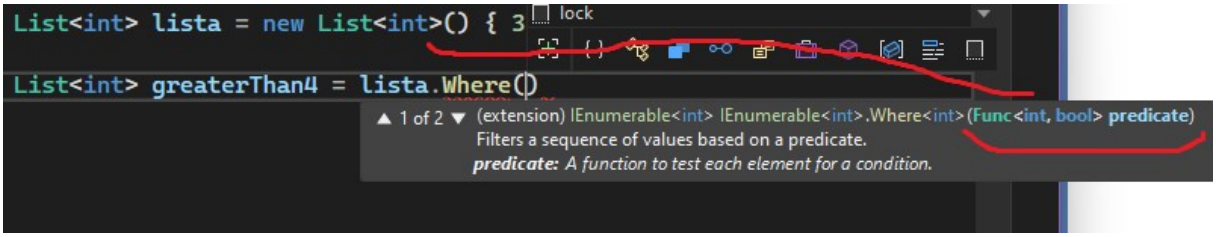
Jak widać w konstruktorze przyjmuje ona funkcję (Action – nic nie zwracamy, brak parametrów), która ma być wykonywana w nowym wątku.

```
1 reference
static void NewThreadFunc()
{
    //do something
}

0 references
static void Main(string[] args)
{
    Task t = new Task(NewThreadFunc);
    t.Start();
}
```

Oczywiście konstruktor Task ma 8 przeciążeń co widać w podpowiedzi. Inne przeciążenie pozwalają podać funkcję typu Action<object> czyli funkcję z jednym parametrem dowolnego typu (dowolnego, ponieważ wszystko dziedziczy w C# z object – kłania się także polimorfizm i dziedziczenie).

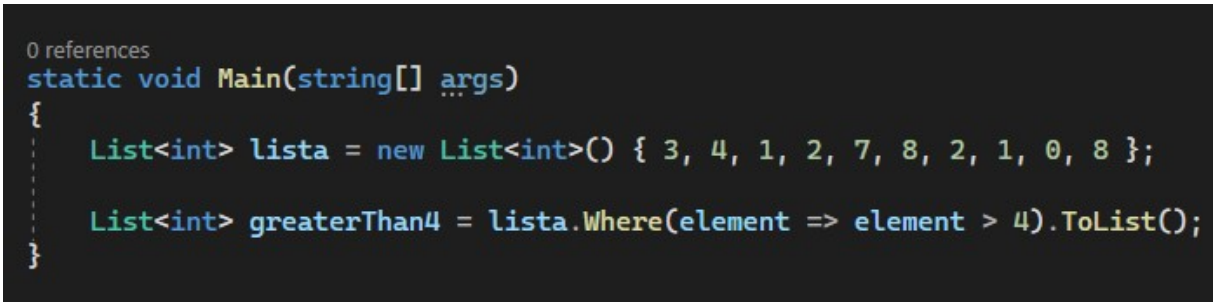
Innym świetnym przykładem jest LINQ.



Funkcja Where() oczekuje w parametrze funkcji, która zwraca bool. Jeżeli dla danego elementu funkcja ta zwróci prawdę jest on zwracany przez Where(). Widać to na obrazku.

Warto wiedzieć, że podpowiedź jest generowana dynamicznie. Widać w niej, że argument funkcji powinien być typu int bo lista przechowuje elementy typu int. Jeżeli lista przechowywała by stringi wyświetliło by się Func<string, bool>.

Podpowiedzi w Visual Studio dają więcej niż to na pierwszy rzut oka wygląda.



Tak to wygląda po uzupełnieniu.

Każdy tego używa niekoniecznie będąc świadomym tego co robi.

Oczywiście przykładów jest jeszcze ogrom.

Jak to jest w innych językach?

- W C i C++ trzeba posłużyć się wskaźnikami na funkcję, type std::function albo zostawić typ auto do automatycznego wypełnienia
- W Pythonie, JSie jeżeli nie stosujemy typów to nie ma tego problemu