Aus einer anderen Klasse, aus einem anderen Thread in MainWindow schreiben

Möchte man aus einer anderen Klasse, aus einem anderen Thread etwas in die Mainklasse Steuerelemente schreiben, stößt man auf 2 Probleme:

1. Man kann aus Thread 2 nicht in Thread 1 schreiben

2. Man kann nicht, ohne ein Objekt angelegt zu haben nicht in die Steuerelemente schreiben.

Abhilfe schafft ein kleiner Trick.

MainWindow.cs:
[crayon-6767e44ada35e790710448/]
[crayon-6767e44ada365157154258/]

meineAndereKlasse.cs:
[crayon-6767e44ada366430531797/]
Kleine Erweiterung, selbes Prinzip um ein Imagecontrol zu ändern:

MainWindow.cs:
[crayon-6767e44ada368222431366/]
meineAndereKlasse.cs:
[crayon-6767e44ada369770487862/]

***Quelle**: Stackoverflow*

Extension Method Time zu Decimal und zurück

wiederum aufbauen auf den letzten Beitrag möchte ich hier ein Snippet vorstellen, mit dem man Zeit in Decimal und Decimal in Zeit umwandeln kann.

Beispiel 12:45 -> 12.75 oder eben 12.75 -> 12:45

[crayon-6767e44ada7a8383743679/]

ExtensionMethods – ToInt

Im vorherigen Beitrag habe ich etwas zu der Extension Method ToString erklärt. ToInt() gibt es von Haus aus nicht. Aber das ist nicht schlimm, denn diese können wir uns selber basteln:

[crayon-6767e44ada91b101564040/]

Das Ausschlaggebende ist, dass wir als Rückgabewert von der Methode ToInt einen Datentypen haben. Nämlich den Int32 Datentyp.

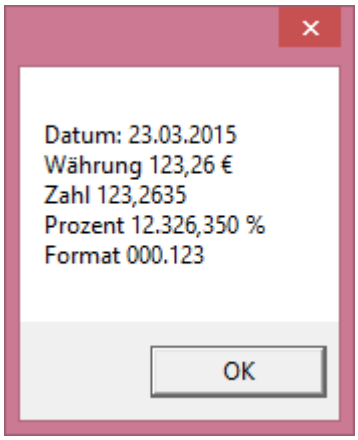
Weiter kann diese Methode jeden Datentyp implementieren, weil diese Methode generischer Natur ist. Leider ist aber auch hier ein try/catch Notwendig, denn wenn wir versuchen einen String „Apfel“ in ein Integer zu konvertieren, ist dies nicht möglich und es folgt die Ausnahme.

ToString

Möchte man einen Wert in String konvertieren, gibt es neben Convert.ToString(…) auch einfach .ToString(). Die 2. Möglichkeit ist weit mächtiger als dass diese einfach nur konvertieren kann. Damit lassen sich Datumsformate individuell darstellen, zahlen Runden usw. Mal ein paar Beispiele:

[crayon-6767e44adaacb249310990/]

als Ausgabe erhalten wir:



Dies war nur ein Bruchteil an Möglichkeiten. Weitere Formatzeichen und eine größere Erklärung gibt es hier:

http://openbook.rheinwerk-verlag.de/csharp/kap30.htm

Datum nach darauffolgenden Tagen gruppieren

Ehrlich gesagt habe ich etwas gebraucht um den Titel richtig zu wählen. Aber ich möchte einmal aufzeigen, welches Szenario ich meine:

Man hat eine Collection mit folgenden Datumelementen:

31.12.2014
01.01.2015
02.01.2015
16.02.2015
19.02.2015
20.02.2015

und möchte sie so sortieren:

Von – Bis

31.12.2014	02.01.2015
16.02.2015	16.02.2015
19.02.2015	20.02.2015

Als Quelle steht uns eine gefüllte List zur Verfügung:
[crayon-6767e44adac53884749115/]
Das heißt, wenn der nächste Eintrag nicht der darauffolgende Tag ist, wird eine neue Gruppe erstellt.
[crayon-6767e44adac57309485006/]
Möchte man nun noch die von und bis Werte auswerten, kann man folgendes nutzen:
[crayon-6767e44adac59046308385/]

Quelle: http://stackoverflow.com/questions/27393626/in-c-what-is-the-best-way-to-group-consecutive-dates-in-a-list

---

ADODB Recordset to Arraylist

Ich weiß, ADODB ist veraltet und sollte nicht verwendet werden. Wer jedoch doch damit arbeiten muss, kann hier mal weiter lesen.

Namespace:
[crayon-6767e44adae9a695717357/]
Typen:
[crayon-6767e44adae9f284659962/]
Connections:
[crayon-6767e44adaea1733272812/]

Methode zu ArrayList
[crayon-6767e44adaea3192704498/]

---

generische Listen /

Collection

Zwar braucht die Dictionary<T, K> weniger Zeit beim hinzufügen/ löschen von Werten, braucht die SortedList doch weniger Ramkapazitäten und ist im großen und ganzen schneller. Deutlich langsamer ist die Hashtable und SortedDictionary.

<T> = Typenparameter, erwartet wird ein Datentyp als Parameter. Beispiel string, int, object,…

<V, K> = siehe <T>, jedoch wegen besserer Lesbarkeit stellt dieser Typenparameter den Wert des **V**alue bzw. **K**ey da.

List<T>
[crayon-6767e44adb0af239808858/]
SortedList<V, K>
[crayon-6767e44adb0b3010926420/]
Beinhaltet einen Key und Value. Über den Key lässt sich das Value herausfinden. Beispiel:
[crayon-6767e44adb0b5808266082/]
liefert den Wert Boolean Wert True

Dictionary<V, K>
[crayon-6767e44adb0b7683486930/]
ArrayList (Object)
[crayon-6767e44adb0b9035426828/]
Hashtable (Object K, Object V)
[crayon-6767e44adb0ba871197363/]
ObservableCollection
[crayon-6767e44adb0bc281704031/]
Im Gegensatz zur List<T> nutzt die ObservableCollection die INotifyCollectionChanged Schnittstelle. Diese gibt eine Meldung, sobald sich in der Collection etwas geändert hat. Sehr Sinnvoll, wenn man diese Collection an ein Steuerelement per WPF binden möchte, aktualisiert sich das Element so

automatisch. Ein Nachteil ist jedoch, dass man die Liste nicht aus der Liste suchen/sortieren kann. Hier kann man nachlesen, wie man dies doch mit einbauen kann -> Link

List<Tuble<T,A,B>

Bisher hatten wir immer nur die Möglichkeit über Key / Value ein Pärchen vom Eintrag zu bilden. Manchmal kommt man aber in die Situation wo man nicht das Key/Value Prinzip haben möchte, oder mehr als 2 Argumente übergeben möchte. Da kommt das seit .NET 4.0 eingeführte Tuble ins Spiel. Die einzelnen Einträge werden dann als Item1, Item2 usw. innerhalb des Eintrags geführt.
[crayon-6767e44adb0bd469827536/]

Quelle: http://blog.bodurov.com/Performance-SortedList-SortedDictionary-Dictionary-Hashtable/

Delegaten und Events

Delegaten sind in C# aufgebaut wie normale Methoden, jedoch besitzen sie keinen Code der ausgeführt wird, sondern weisen lediglich auf eine Methode mit Code hin.

Das bedeutet, dass die Delegaten genau so wie Methoden

einen Zugriffsmodifikator (private, public), einen Rückgabe wert und Parameter haben. Die Parameter müssen aber in der Anzahl und Datentyp den Methoden identisch sein.

[crayon-6767e44adb2fc977308013/]

Interessant werden Delegaten in Verbindung mit Events auf Deutsch Ereignisse. Events werden ausgelöst. Wenn man einen Button klickt, löst man das Click-Event aus. In WPF und Windows Forms haben die Steuerelemente z.B. viele vordefinierte Events. So kann man ein Mouseover Event auslösen lassen, wenn man z.B. mit der Maus über ein Steuerelement fährt. Sobald nun ein Event ausgelöst wird, versucht das an Ihm hängende Delegat eine Methode auszulösen mit den Rückgabewerten und Parameter, die dem Delegat zur Verfügung stehen. Das Interessante ist, dass ein Delegat unabhängig vom Zugriffsmodifikator arbeitet. An beide wird eine Methode durch überladung angehängt " += " oder abgezogen " -= „.

Mal ein kleines Beispiel, wie wir mithilfe eines Delegaten und Event von einem neuen Window in das Mainwindow Label etwas schreiben können:

MainWindow:
[crayon-6767e44adb301546121989/]
Das neue Window1:
[crayon-6767e44adb303340949330/]
Hier sehen wir, dass zunächst im MainWindow dem Event vom Objekt w1 die Methode wnd_MyCustomEvent hinzugetan wurde. Jedesmal also, wenn das Event ausgeführt wird, wird auch diese Methode ausgeführt.

Im Window1 wird nun die Methode durch das klicken auf den cmd_w1_Button das Event MyEvent ausgelöst und dem dazugehörigen Delegaten wird der Parameter von 100 übergeben.

Abschließend wird nun im MainWindow dieser Wert angezeigt

Der nächste Schritt, den man gehen könnte ist, noch ein neues Window zu erstellen und dort eine Methode hinzufügen:
[crayon-6767e44adb305612588486/]
Dem MainWindow fügen wir hinzu:
[crayon-6767e44adb306332632346/]

Ausgabe ist eine Messagebox mit dem Wert 100 aus der Klasse in Window2 heraus

---

Kommentare und Regionen

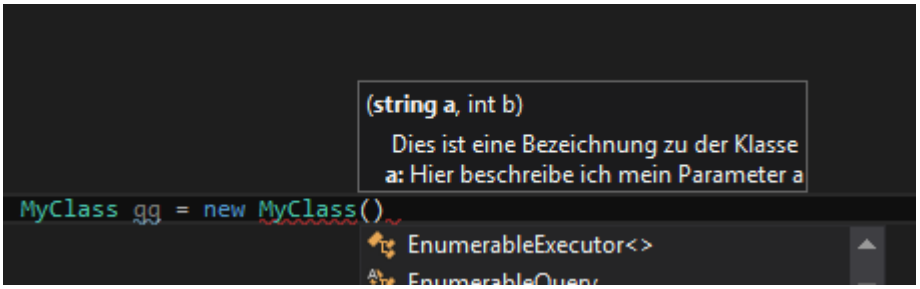
Gewöhnlich verwendet man Kommentare zweierlei Weise. Zum einen kommentiert man einen Code aus, wenn man sich eine Änderung gemacht hat, aber immer die Option haben möchte es rückgängig machen zu können, zum anderen eben um z.B. eine bestimmte Funktion mit eigenen Worten zu beschreiben, damit ein anderer Entwickler sich schnell damit zurecht finden. Andererseits sei es nicht unterschätzt, dass man selbst eigenen Code nach einigen Wochen wieder verstehen lernen muss.

Ich habe bereits Codes gesehen, die eher wie ein Fußballfeld von oben gesehen aussehen, als nach einer Programmierkunst. Eine komplette grüne Landschaft ist natürlich irgendwo kontraproduktiv.

Unter C# hat man aber viel mehr Möglichkeiten etwas zu kommentieren als durch das bekannte //.

1. Der klassische Kommentar:
[crayon-6767e44adb4f8482009789/]
2. Kommentar über mehrere Zeilen:
[crayon-6767e44adb4fe538578612/]
3. Kommentieren von Methoden/Klassen über <summery>:

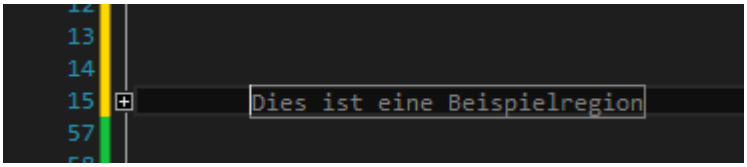
Nachdem man eine Klasse/Methode geschrieben hat, kann man genau darüber einfach 3 Schrägstriche /// machen und ein summery-Kommentar wird generiert. Die param Felder geben Auskunft über die Parameter, die gesetzt wurden.
[crayon-6767e44adb500647377867/]



4. Regionen schaffen

Zugegeben handelt es sich nicht um einen Kommentar, sondern um eine Möglichkeit bestimmte Teile in Regionen zu teilen um eine bessere Übersicht zu schaffen.

dies geschieht, indem man einfach
[crayon-6767e44adb502753145251/]
Nun kann man die Region schließen und hat einen bestimmten Codeteil sauber verpackt



---

generische Klassen

Mit generischen Klassen kann man Datentyp unabhängigen Aufbau einer Klasse erreichen. Das bedeutet, man kann dann ein Objekt der Klasse erzeugen und sagen, dass die Methoden dort eben mit diesen Datentyp arbeiten soll, dem wir dem Objekt übergeben. Auch eine dort deklarierte Variable ist eben der Typ, der

übergeben wurde.

Dabei arbeitet man mit einem sogenannten **T**yp-Parameter, welcher ähnlich einem Platzhalter für einen Datentyp stehen soll. Deutlicher wird das vielleicht durch das folgende Beispiel:

[crayon-6767e44adb6f1013632254/]
Die Klasse erhält einen Typenparameter T. Man könnte auch einen anderen Buchstaben nehmen (i.d.R. ist der erste immer ein T), oder durch Komma getrennt weitere Datentypen anfügen.

Etwas ungewöhnlicher sieht da die Klassen-Eigenschaft in der 2. Zeile aus. Denkt man sich das T weg, könnte dort ein int, string[], double oder sonstwas stehen.

Dann folgt ein Konstruktor, der dieser Eigenschaft nun einen Wert zuweist

und zum Schluss noch eine Methode, die diese Eigenschaft ausgibt.

Contraints

im oberen Beispiel hätte man rein theoretisch die Möglichkeit alle möglichen Datentypen zu setzen. Möchte man dies aber auf eine bestimmte Art von Datentypen beschränken, so gibt es eine where Klausel. Das Muster erinnert dann ein wenig an SQL.

Dann nimmt die Klasse folgende Bezeichnung ein:
[crayon-6767e44adb6f5396579649/]
Dabei kann man daraus den Satz machen: „Filtere alle, die von der Elternklasse IComparable erben“. Dabei macht man sich am besten im Objektexplorer schlau, welche Einschränkungen man treffen möchte.

Möchte man ein Objekt erstellen und dieser Datentyp passt nicht unter das Gefilterte, wird bereits zur Entwicklungszeit ein Fehler angezeigt.

Quelle: http://www.dotnetperls.com/generic