Kommentare und Regionen

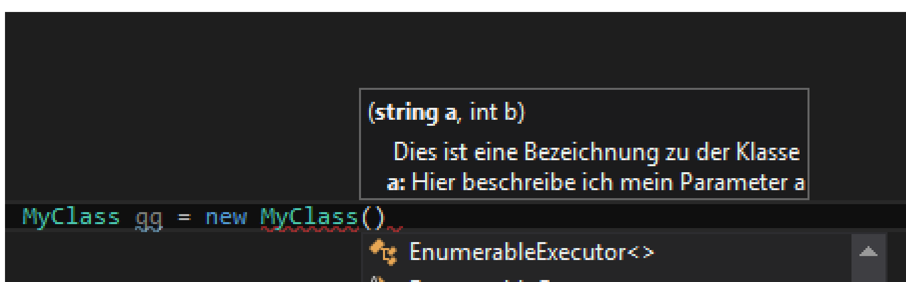
Gewöhnlich verwendet man Kommentare zweierlei Weise. Zum einen kommentiert man einen Code aus, wenn man sich eine Änderung gemacht hat, aber immer die Option haben möchte es rückgängig machen zu können, zum anderen eben um z.B. eine bestimmte Funktion mit eigenen Worten zu beschreiben, damit ein anderer 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-6767a5bc5a9d4491966839/]
 2. Kommentar über mehrere Zeilen:
[crayon-6767a5bc5a9db183372402/]
3. Kommentieren von Methoden/Klassen über <summery>:

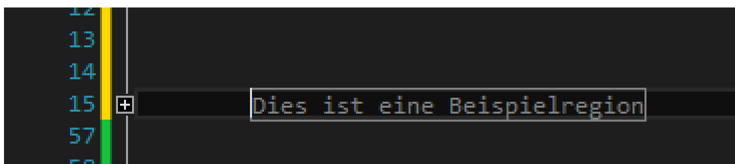
Nachdem man eine Klasse/Methode geschrieben hat, kann man genau darüber einfach 3 Schrägstriche /// machen und ein summery-Kommentar wird generiert. Die param Felder geben Auskunft über die Parameter, die gesetzt wurden.
[crayon-6767a5bc5a9dd592392857/]



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-6767a5bc5a9de073678279/]
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-6767a5bc5af83848626714/]
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-6767a5bc5af87771601252/]
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.

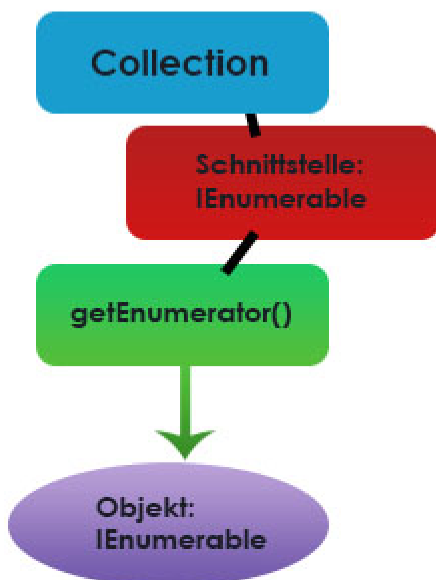
Quelle: http://www.dotnetperls.com/generic

Iteratoren

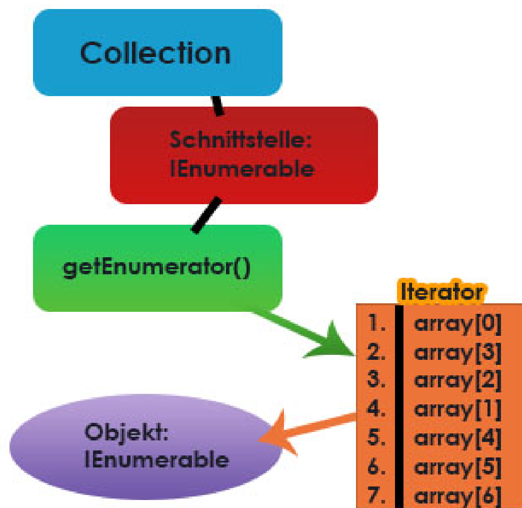
Das Verb iterieren bedeutet aus einer Kollektion (engl. Collection) nacheinander durch die enthaltenen Elemente durch zulaufen. Nichts anderes tut ja auch die foreach Schleife.

Doch was ist dazu Notwendig, damit eine Iteration stattfinden kann? Die Collection muss die IEnumerable Schnittstelle unterstützen.

IEnumerable besitzt einen Inhaltsverzeichnis (engl. Index) welcher alle Elemente der Liste durchnummeriert beinhaltet. Weiter hat die Schnittstelle eine Methode getEnumerable() um diesen Index auszugeben. Genau dies tut auch die foreach Schleife. Sie ruft die Methode getEnumerable() auf und durchläuft vom ersten bis zum letzten Enumerator.



Glücklicherweise beinhalten die meisten Collections in .NET diese Schnittstelle (Array, List, DataRow usw.)



Möchte man nun, dass eine Klasse nach einem bestimmten Muster iteriert wird, bietet es sich an, eine Metode in dieser Klasse zu erstellen welche vom Typ
[crayon-6767a5bc5b0f9348093506/]
ist und die einzelnen Collection Elemente neu setzt.
Beispiel:
[crayon-6767a5bc5b0fd160367957/]
[crayon-6767a5bc5b0fe954975327/]
durch einen Aufruf von
[crayon-6767a5bc5b100579712975/]
wird auch nur die ersten 3 durchiteriert

Quelle: http://www.s-line.de/homepages/trac/wissen/dot-net/csharp-2.html

C# und MySQL /MariaDB

Als erstes benötigt man die MySql.Data.dll, die man als Verweis hinzufügen muss.
Dazu öffnen wir die Packet-Manager-Konsole durch den Tastenkürzel **ALT + T + N + O** und geben dort ein:
[crayon-6767a5bc5b2c7377913343/]
und bestätigen das Konsolenfenster mit Enter.

2. Namespace hinzufügen:
[crayon-6767a5bc5b2ca446179240/]
3. StringBuilder erzeugen. Alternativ geht natürlich ein gewöhnlicher String:
[crayon-6767a5bc5b2cb953460126/]
4. Connection String erzeugen. Port kann evtl. ein anderer sein:
[crayon-6767a5bc5b2cd851452472/]

> *In der Regel umschließt man Datenbankverbindungen in einem try/catch. Dies wird hier von meiner Seite aber aus Gründen der besseren Übersicht nicht getan. Solltet Ihr das so verwenden wollen, empfiehlt es sich definitiv einen try/catch Block um rum zu verwenden*

Methode um ein SQL Befehl auszuführen
[crayon-6767a5bc5b2ce422342795/]
Beispiel:
[crayon-6767a5bc5b2cf973641693/]
Methode um einen Befehl aus der Datenbank zu erhalten
[crayon-6767a5bc5b2d1038041149/]

Beispiel:
[crayon-6767a5bc5b2d2725663013/]

Google Chrome – Benutzername und Passwort auslesen

stand heute funktioniert die genannte Methode noch.

Ich möchte mit dieser Anleitung nicht erreichen, dass du damit die Passwörter anderer ausliest, sondern lediglich für eigene Zwecke einsetzt. Ich übernehme daher auch keine Haftung was ihr damit macht. Bitte, tue dir selbst den Gefallen und mache das nicht, denn das kann dein ganzes Leben verändern. Negativ natürlich!

Ich stelle hier 2 Klassen zur Verfügung. Die erste, von mir geschriebene für die Prozesse, die andere ist die Decodierung des Passwortes von einem Autor, vor dem ich meinen Hut ziehe.

Die Datei, welche sämtliche gespeicherte Anmeldedaten enthält findet ihr hier:
[crayon-6767a5bc5b5d0001062328/]
Dabei lässt sich diese Datei „Login Data“ mit Sqlite öffnen und bearbeiten. Lediglich die Passwortfelder sind als BLOB gekennzeichnet, lassen sich aber wie oben bereits erwähnt dekodieren.

Sobald man ein Objekt der Klasse Chrome erzeugt, muss man als Parameter, einen Dateinamen .db angeben, worauf diese mit denselben Dateinamen mit [Objekt].removeTemp(„Dateiname.db“) gelöscht werden kann.

Das Objekt der Klasse Chrome enthält als Eigenschaft unter anderem 3 ausgelesene Listen: Site, Username, Password

chrome Download

Tastenkürzel

Strg + K, Strg + C = Zeile auskommentieren

Strg + K, Strg + U = auskommentierte Zeile wieder aktivieren

Strg + K, Strg + D = Zeilen wieder richtig ausrichten

Strg + M + L = kompletten Codeabschnitt reduzieren

Strg + Umschalt. + L = Zeile löschen

Strg + X = Zeile Ausschneiden

Revidiert am 2015-05-17 (Billige Tastenkürzel raus und wirklich interessante rein)

Strg + Alt + B = Breakpointfenster mit allen verfügbaren Breakpoints anzeigen

---

Backgroundworker

Wenn man aus einem Thread heraus eine Zeitaufwändige Operation durchführt, kann dies dazu führen, dass man den Eindruck bekommt, das Fenster wäre eingefroren. Der Grund ist ganz einfach, weil diese Aufgabe im ersten Thread stattfindet und solange dauert, bis es fertig ist. Jetzt stellt .Net den Threading Namespace zur Verfügung womit man für solche Aufgaben einen neuen Thread aufmachen könnte. Eben genau dies

tut auch der Backgroundworker nur in einer stark vereinfachter Form.

Backgroundworker einbinden:

1. Namespace hinzufügen:
[crayon-6767a5bc5b7e1688990321/]
2. Backgroundworker deklarieren:
[crayon-6767a5bc5b7e5445854752/]
3. diesem Objekt events zuweisen:
[crayon-6767a5bc5b7e7009261275/]
4. die beiden Methoden implementieren:
[crayon-6767a5bc5b7e9775553431/]
5. worker asynchron ausführen lassen:
[crayon-6767a5bc5b7eb195585435/]

wünscht man einen Bericht über den derzeitigen Prozess, so muss man auch das event **ProgressChanged** hinzufügen, **ReportProgress(Int32)** in DoWorks ausführen lassen und in **WorkerReportsProgress** das Resultat bekommen

Häufigste Problematik die man Anfangs hat ist, dass man nicht auf die Steuerelemente aus dem 1. Thread zugreifen kann. Abhilfe schafft dort
[crayon-6767a5bc5b7ed364308129/]