

Többnyelvűség – Multilanguage

Minden szoftvernek, mely tömeggyártásra kerül, lényeges részét képezi a többnyelvűség támogatása. Nem elég, ha csupán egy nyelven érhetőek el a program egyes funkciói, gondolnunk kell arra, ha más országokban is sikerül bevezetni. Ennek megfelelően legalább egy világnyelvet – jelen esetben az angolt – is támogatnia kell, a magyar mellett. Emellé nem lenne hátrány, ha német, illetve orosz nyelvet is beépítenénk, és persze ahhoz, hogy ezt megvalósítsuk, már csak az adott nyelvre való fordítás szükséges, mely már nem szoftvertervezési probléma. Ez a lehetőség azért áll rendelkezésünkre, mert a szoftverünk dinamikus nyelvváltást támogat. A dinamikusság alapját a WPF alól elérhető Resource Dictionary komponens biztosítja. Ez a megoldás egyrészt lehetőséget biztosít, hogy dinamikusán bővíthessük termékünket új nyelvekkel. Ez alatt azt értjük, hogy nem statikus módon előre definiálva építünk bele nyelveket, hanem külön fájlba emelünk ki minden egyes nyelvet, majd ezeket a fájlokat használjuk a megfelelő rutinokkal, e módon új nyelv hozzáadása során csak egy új fájlal kell bővítenünk a rendszert. A másik lehetőség az előbbiekből következik, mert a nyelvek váltásakor csak a megfelelő nyelvet leíró fájlt kell használnunk, mely igen könnyen cserélhető. Megoldásunk szükséges feltétele, hogy minden egyes szövegrészlet, melynek állítani kívánjuk a nyelvét, ki legyen emelve string-ként egy Resource Dictionary fájlba. Itt minden egyes objektumot egy kulcs és egy érték jelképez, a kulccsal hivatkozhatunk a szövegerőforrás értékére, magára a szövegre. Az alábbi XAML kódrészletben látható, hogy a különféle nyelvi erőforrásfájlok hogyan épülnek fel, és láthatjuk, hogy minden fájlban ugyanazon kulcsok találhatók meg, de más és más nyelvű szöveget rendelünk hozzá, ezáltal biztosítva a többnyelvűséget, és cserélhetőséget:

Magyar nyelvi erőforrás fájl részlete:

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:system="clr-namespace:System;assembly=mscorlib">
    <system:String x:Key="newvarheader">Új változó</system:String>
    <system:String x:Key="name">Név</system:String>
    <system:String x:Key="add">Hozzáad</system:String>
    <system:String x:Key="newprojectheader">Új Projekt</system:String>
</ResourceDictionary>
```

Angol nyelvi erőforrás fájl részlete:

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:system="clr-namespace:System;assembly=mscorlib">
    <system:String x:Key="newvarheader">Variable</system:String>
    <system:String x:Key="name">Name</system:String>
    <system:String x:Key="add">Add</system:String>
    <system:String x:Key="newprojectheader">New Project</system:String>
</ResourceDictionary>
```

Látható, hogy a nyelvi erőforrás XAML kódban íródik, mely itt nagyon praktikusan alkalmazható. Vörössel aláhúzva láthatjuk a szövegerőforrás azonosítóját, a kulcsot, az angol és magyar fájlrészletben is, illetve zölddel aláhúzva az erőforrás értékét, melyet a kulcs azonosít, és amely majd megjelenik a programunk menüiben, beállításában, stb.

Példaként az előző kódrészletben leírt erőforrás használatára, a szoftverünk „Új projekt létrehozása” ablakának leíró részéből vett kódrészletet mutatom be:

```
Title="{DynamicResource newprojectheader}" Height="300" Width="300">
```

A Title attribútum itt az ablak címét hivatott beállítani, itt értéknek megadhatnánk egy egyszerű stringet is, azonban kapcsos zárójelbe zárva a DynamicResource kulcsszót használva, és a vörössel aláhúzott kulcsot megadva az erőforrásunkra hivatkozunk, ami dinamikusan változik.

Erőforrások – Resource Dictionary

Látható, hogy az erőforrások használata majdhogynem elengedhetetlen, nemcsak a többnyelvűség biztosításának céljából, de később meg fogjuk látni, hogy a szintaxis felismerésében is nagy szerepet játszik. Épp ezért az alábbiakban ennek a remek komponensnek a tulajdonságairól, és funkcionalitásáról ejtenék néhány szót.

Szoftverünk felépítése során erőforrásokat használunk. Erőforrások alatt itt azt értjük, hogy ha egyfajta komponenst, beállítást többször, illetve több helyen is használunk, akkor ezeket kiemelhetjük egy resource fájlba, majd ezt töltjük be és használjuk, ahelyett, hogy statikusan, a program minden pontján leírnánk ezeket. Ezzel egyrészt sok felesleges kódot spórolunk meg, másrészt dinamikussággal vétezzük fel az egyes funkciókat. Mivel az erőforrásokat egy helyen definiáltuk, ezért ha változtatni kell egy funkción, elég egy helyen átírni azt, és ez által mindenhol érvényesülni fog a változás. Az erőforrás egy .xaml kiterjesztésű fájl, és XAML kódban íródik. Természetesen bármi lehet erőforrás, amit használni kívánunk. Jelen esetben ezek alatt string, azaz szöveg típust értünk, de lehetnek például stílusok, ami ugyancsak hasznos, ha bonyolult stílust szeretnénk leírni, ebben az esetben már csak azért is érdemes kiemelni erőforrásként, hogy a kódot szeparáljuk, ezzel növelve az átláthatóságot, és az egyszerűbb kódolhatóságot. Akár összetett, komplex objektumokat is alkalmazhatunk erőforrásként. Minden erőforráshoz tartozik egy kulcs, amely azonosítja, és ami alapján elérhetjük azt, segítségével történik a betöltés, melyet az alábbi kódrészekben tekinthetünk meg:

Az erőforrásokat az App.xaml fájlban kell definiálni. Itt statikusan töltjük be az erőforrást, azaz a program indulásakor ezt használja alapértelmezettként:

```

<Application.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="Languages/Hungarian.xaml" />
      <ResourceDictionary Source="SyntaxKeywords/Hungarian.xaml" />
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Application.Resources>

```

Amint láthatjuk a MergedDictionaries tulajdonság kihasználásával egyszerre több erőforrást is betölthetünk alapértelmezettként. Egyértelmű, hogy a ResourceDictionary tag-et kell használnunk, illetve a Source attribútumnak az erőforrás fájl projekten belüli elérési útvonalát kell megadni.

Azonban ahhoz, hogy ezek a fájlok, erőforrások dinamikusan cserélhetőek legyenek, szükségünk van C# kódra. Példaképp az alábbi kódrészletben látható a hozzárendelés:

```

switch (logolang)
{
  case "langhu":
    App.Current.Resources.MergedDictionaries[1].Source =
    new Uri("SyntaxKeywords/Hungarian.xaml", UriKind.Relative);
    break;
  case "langen":
    App.Current.Resources.MergedDictionaries[1].Source =
    new Uri("SyntaxKeywords/English.xaml", UriKind.Relative);
    break;
}

```

Itt egy switch elágazás szerkezetben váltunk a nyelvek között, attól függően, hogy a felhasználó mit választott. A betöltés logikailag hasonló a XAML kódban való betöltéshez, a MergedDictionaries tömb az applikációnk erőforrásait tartalmazza. Kiválasztjuk a cserélni kívánt erőforrást a megfelelő indexszel, majd beállítjuk az elérési útvonalát egy Uri objektum segítségével, az elérési útvonalat megadva.

Szintaxis Kiemelés – Syntax Highlight

A fejlesztői nyelv, esetünkben a LOGO szintaxisának kiemelése kulcsfontosságú szerepet kap a szoftverben. Programozók lévén jól tudjuk, hogy nagyban megkönnyíti a kódolást a megfelelő és igényes fejlesztői környezet, ennek része, hogy a kód nem csak szemantikusan átlátható, de könnyen felismerhetőek benne a nyelv kulcsszavai. Ez által a program egyes részei jobban átláthatóak, könnyebben elkülöníthetőek, részben pedig úgy működik, mint egy szintaktikai elemző, mivel csak a helyesen begépett kulcsszavakat emeli ki.

Fontos megjegyezni, hogy az eddigi LOGO szerkesztők nem rendelkeznek ilyen funkcióval, így sokkal több hibalehetőséget tartalmaznak, nehezebben átláthatóak az itt írt kódok. Szükségesnek láttuk, hogy többféle nyelven képes legyen a LOGO szintaxisának

felismerésére, így megoldásként itt is a Resource Dictionary komponenst alkalmaztuk, illetve a tényleges szövegformázás megvalósításához szükség volt még egy RichTextBox komponensre, mely a későbbiekben kerül bemutatásra. Mivel a szoftver magyar és angol nyelvet támogat jelenleg, a szintaxis kiemelés is ezt a két nyelvet ismeri fel, ez azonban dinamikusan bővíthető, akár csak a nyelvek. Ezen kívül opcionálisan biztosítja a case-sensitivity tulajdonságot, azaz kis- és nagybetű közti különbséget szükség esetén, illetve a személyre szabhatóság kedvéért a szövegkiemelés több színben is lehetséges. A szintaxis kiemelés alatt valójában a kulcsszavak kiemelését értjük. Minden programozási nyelvben vannak előre definiált kulcsszavak, ezeket különítjük el a kód többi részletétől, például konstansoktól, különleges karakterektől, stb. Dinamikus megoldásunk miatt az elkészített osztályok bármilyen más programozási nyelv kulcsszavainak felismerésére képes. Ehhez használjuk a Resource Dictionary-t. Minden programozási nyelvhez, illetve esetünkben a LOGO nyelv angol és magyar kulcsszavaihoz tartozik egy-egy erőforrás fájl, melyben a nyelvekhez hasonlóan, kulcsokkal azonosítva adhatjuk meg a programozási nyelv kulcsszavait. Ezt a megvalósítást láthatjuk az alábbi kódrészleten:

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                    xmlns:system="clr-namespace:System;assembly=mscorlib">
    <system:String x:Key="kw_elore">előre</system:String>
    <system:String x:Key="kw_elorer">e</system:String>
    <system:String x:Key="kw_hatra">hátra</system:String>
    <system:String x:Key="kw_hatrar">h</system:String>
    <system:String x:Key="kw_jobbra">jobbra</system:String>
    <system:String x:Key="kw_jobbrar">j</system:String>
    <system:String x:Key="kw_balra">balra</system:String>
    <system:String x:Key="kw_balrar">b</system:String>
</ResourceDictionary>
```

Látható, hogy az x:Key attribútum az erőforrás kulcsát tartalmazza, mellyel a nyitó és záró XAML tag közé zárt LOGO kulcsszóra hivatkozunk. Az angol LOGO szintaxis ugyanilyen formában adhatjuk meg, egy másik erőforrás fájlban.

Az erőforrásokban definiált szintaxis kulcsszavakat el kell juttatnunk az azt használó programrészekhez ezt már C# kód segítségével oldjuk meg. Mivel szoftverünk LOGO nyelvet használ, illetve azon belül többnyelvű felismerhetőséget is lehetővé tesz, szükségünk van egy string tömbre, mely tartalmazza azokat a kulcsokat, melyeket az előbbi kódrészletben bemutatott erőforrásfájl tartalmaz.

Ennek a string tömbnek egy részlete a következőképpen néz ki, a kapcsolat átláthatósága érdekében vörössel aláhúzva tekinthető meg az a kulcs, melynek a párja a Resource Dictionary-ben aláhúzva látható:

```

string[] kw_keys = new string[]
{
    "kw_alore",
    "kw_elorer",
    "kw_hatra",
    "kw_hatrar",
    "kw_jobbra",
    "kw_jobbrar",
    "kw_balra",
    "kw_balrar",
    ...
};

```

Az itt definiált kulcsok betöltését az alábbi metódus végzi el:

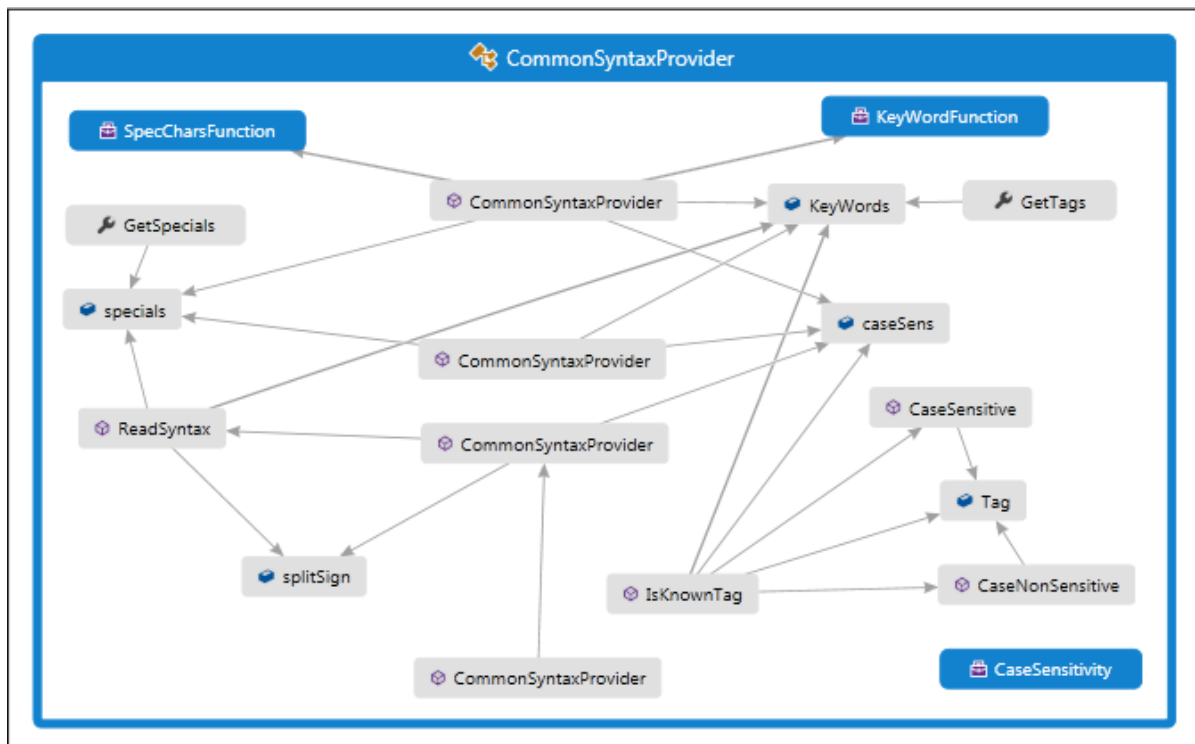
```

public void UpdateKeywords()
{
    keyWords.Clear();
    for (int i = 0; i < kw_keys.Length; i++)
    {
        keyWords.Add(App.Current.TryFindResource(kw_keys[i]).ToString());
    }
}

```

Valójában a metódus végigiterál a kulcsokon, és kinyeri az erőforrásban hozzájuk tartozó szintaxis kulcsszavakat, ezeket pedig odaadja annak az osztálynak, mely elvégzi a szintaxis felismerés rutinját. A továbbiakban ezeknek az osztályoknak a bemutatásáról lesz szó.

CommonSyntaxProvider osztály



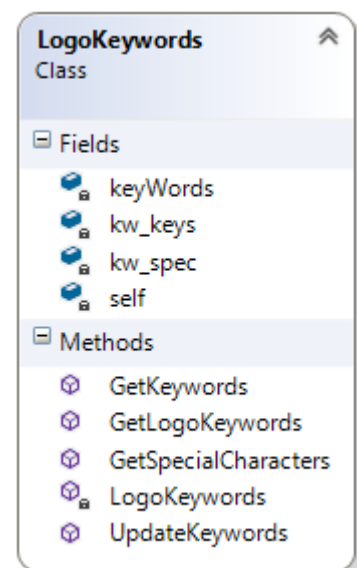
Az osztály szerkezetét remekül szemlélteti a gráf szerkezettel felépített ábra. Ez a funkció a Visual Studio 2012-es verziójában érhető el, nagyon hasznos, mivel megjeleníti az osztály referenciáit, a kapcsolatokat. Ahogyan a neve is mutatja, ez az osztály biztosítja majd a szintaxist, ez alatt pedig a kulcsszavakat értjük. Ennek tárolására egy *KeyWords* nevű string alapú listát használunk, ebbe kell beolvasnunk a kulcsszavakat, ezen kívül szükséges egy lista a speciális karakterek tárolására, ez a dolga a *specials*, karakter alapú listának.

Középen láthatóak az osztállyal azonos nevű konstruktorok, melyek többféle szintaxis beolvasási lehetőséget adnak. Erre azért van szükség, mert a megfelelő szintaxis beolvashatjuk egy egyszerű szövegfájlból is, azonban ez többnyire tesztelés céljából alkalmas, illetve kisebb projektek esetén célszerű. Ez esetben a szövegfájlból egyenként ki kell olvasnunk a kulcsszavakat, illetve elkülönítve a speciális karaktereket, és elhelyezni a megfelelő listákban, erre szolgál a *ReadSyntax* metódus. Lehetőség van arra, hogy rögtön listából kapjuk meg a kulcsszavakat, ez esetben nincs is szükség semmilyen konverzióra. Ezen belül is két lehetőség van, az egyik konstruktor már kész listát vár paraméterként, a másik, amelyet a programban is alkalmazunk megoldásként, hogy a konstruktor egy callback függvényt vár paraméterként, amely egy listát ad vissza. Ez azért hasznos, mert így a kulcsszavakat beolvashatjuk Resource Dictionary-ből az osztályon kívül, ezt függvénybe zárhatjuk, és magát a függvényt adjuk paraméterként a konstruktornak. A konstruktor meghívja a függvényt, és felhasználja az eredményt. Így nem szemeteljük tele az osztályt olyan teendőkkal, ami nem a feladata, és dinamikus is marad a módszer.

Az osztályban ezen kívül van két metódus, a *CaseSensitive* és *CaseNonSensitive*, amelyeket az *IsKnowTag* függvény használ, mely megmondja az ellenőrzött szóról, hogy kulcsszó-e vagy nem. Értelemszerűen, az egyik függvény figyelembe veszi a kis- és nagybetűk közötti különbséget, a másik nem.

LogoKeywords osztály

Szükség van egy olyan osztályra, mely a LOGO nyelv kulcsszavait kigyűjti az erőforrás fájlokból. Ezt a problémát a *LogoKeywords* osztály oldja meg. Itt definiáljuk a már korábban bemutatott *kw_keys* tömböt, mely a LOGO szintaxis kulcsszavainak erőforrásbeli kulcsait tartalmazza stringekként. Helyet kap az a két függvény, melyet a *CommonSyntaxProvider* osztály konstruktorának adunk oda, ezekben írjuk meg azt a rutint, mely az erőforrásokból a kulcsok alapján listába gyűjti ki a kulcsszavakat és a speciális karaktereket, így ez a két függvény egy string és egy karakter listát ad vissza.

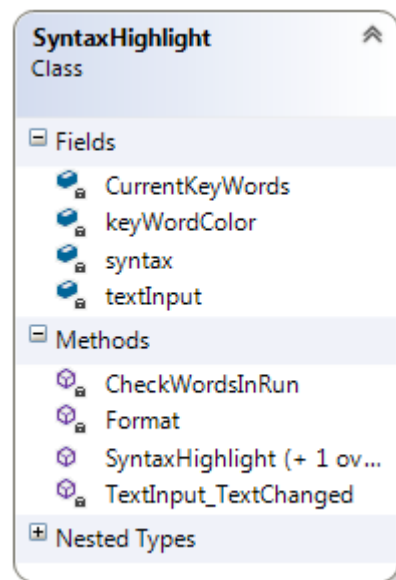


Amikor a szoftverünkben átállítjuk a szintaxis kiemelésének nyelvét, szükség van a kulcsszavak frissítésére, ezt a feladatot végzi el az *UpdateKeywords* metódus. Az osztály maga is tárolja a kulcsszavakat egy listában.

SyntaxHighlight osztály

Ahogy a nevéből is látszik, ez az osztály építi fel és kezeli valójában a szintaxis felismerését, kiemelését, formázását. A kulcsszavak kiemelését végző folyamat a következő műveletekből áll:

Szükség van egy komponensre, melybe a kódot írhatjuk, ennek pedig rendelkeznie kell a szöveg szerkesztését, formázását biztosító tulajdonságokkal, függvényekkel. Esetünkben ez a RichTextBox komponens. A kulcsszavak felismerését természetesen minden egyes karakter beírása után ellenőrizni kell, mert ha a bevitt karakter egy kulcsszó utolsó karaktere, a kulcsszót már is ki kell emelni. Tehát figyelni kell eseményként a karakterbevitelt, és minden változásra meghívni az ellenőrző rutint. Ekkor ki kell nyernünk a kulcsszavakat, elmentve a pozíciójukkal, majd a kulcsszavakon elvégezni a formázást a megfelelő paraméterekkel.



A *SyntaxHighlight* osztályunknak két konstruktort biztosítottunk, az egyik az alapértelmezett formázási paramétereket tartalmazza, a másikban opcionálisan állíthatjuk a szövegi kiemelés színét. A konstruktornak két kötelező paramétere van, az egyik egy RichTextBox példány, a másik pedig a CommonSyntaxProvider osztályunk egy példánya, osztályunk maga is tartalmaz egyet-egyet mindkét típusból, ezáltal aggregációs kapcsolatot biztosítva az osztályok között. Ily módon az osztályunknak lesz egy referenciája arra a komponensre, melyen el kell végezni a megfelelő műveleteket. A RichTextBox komponens TextChanged eseményére feliratkoztatjuk az általunk definiált *TextInput_TextChanged* függvényt, így a referencia által, ha az eredeti komponensnek megváltozik a szövege, rögtön meghívódik ez a függvény, és elvégzi a dolgát:

Üres szöveg esetén nincs teendője. Minden más esetben először meg kell szereznünk a beviteli parancsmező szövegét a RichTextBox példánytól. Ehhez szükségünk van a TextRange osztályra, mely egy bizonyos szövegtartomány tárolására képes. A szöveget ebben a környezetben TextPointer objektumokkal tudjuk kijelölni, méghozzá itt szükség van a tartalom kezdetét és végét jelző pontra, tehát a TextRange konstruktora ezt a két paramétert várva építi fel a szövegtartományt tároló objektumot.

A kinyert szövegen egy „navigátor”, `TextPointer` típusú objektummal navigálunk végig. Ez úgy történik, hogy ciklikusan megvizsgáljuk, hogy a navigátorunk elérte-e már a szöveg végét, ha igen, akkor megromázzuk a kinyert szavakat. Ha nem, akkor minden szövegrészből létrehozunk egy kontextust a `TextPointerContext` típus segítségével, úgy, hogy a navigátor által kiolvassuk a tőle visszafelé eső szövegrészt. Leellenőrizzük, hogy így egy számunkra megfelelő kontextust kapunk-e. Tulajdonképpen, ha az adott kontextus kompatibilis a `Run` típussal – amely egy formázott, vagy formázatlan szövegrész távot reprezentál – akkor erről a `Run` típussá kényszerített objektumból minden ciklusban kigyűjtjük a kulcsszavakat. Ahogy a kulcsszavak kigyűjtése megtörtént egy kontextusból, a navigátor felveszi a következő kontextus pozícióját.

A kulcsszavak kigyűjtését külön metódus végzi el, mely egy `Run` típusú objektumot vár paraméterként, mivel ennek az objektumnak már van `text` tulajdonsága, azaz ebből már lekérdezhető a szöveges tartalma is. Természetesen az objektumból kinyert szövegen kell végigiterálnunk, és ebben kell megkeresnünk a kulcsszavakat, amelyeknek így már tudni fogjuk az indexét is. A kulcsszavak keresése során sok fontos dologra kell odafigyelnünk. Fontos, hogy egy kulcsszót csak akkor emelhetünk ki, ha elszeparálva találjuk a többi kódrésztől, azaz ha előtte és utána vagy `white space` karakter áll, vagy speciális karakter. Itt nyer értelmet a *CommonSyntaxProvider* osztály által tartalmazott *specials*, karakter alapú lista, mely ezeket a speciális karaktereket tartalmazza, ide tartoznak például a pont, zárójelek, stb. karakterek. Erre azért van szükség, mert különféle programozási nyelvekben a kifejezéseket gyakran zárjuk zárójelek közé, illetve a pont karakter általában egy példány, vagy osztály tulajdonságainak, metódusainak elérésére szolgáló operátor, ezek a karakterek pedig közvetlenül állhatnak kulcsszavak előtt, vagy után. Így ha ilyen karaktereket találunk, az nem számít egybeolvadásnak a kulcsszóval.

Tehát, ha egy kulcsszót a megfelelő környezetében találtunk meg, fel kell vennünk egy listába. Ez úgy történik, hogy először a megfelelő indexek alapján kiszedjük az elemzés alatt álló szövegből, ehhez nyilván szükség van a szó kezdő és vég indexére. Meg kell vizsgálnunk, hogy a talált szó valóban a nyelvnek megfelelő kulcsszó-e. Ha igen, beletehetjük a listába.

```
string word = text.Substring(sIndex, eIndex - sIndex + 1);
```

A lista típusát először definiálnunk kell. Mivel ez az a lista, amely a formázni kívánt kulcsszavakat tárolja majd, szükségünk van egy olyan struktúrára, mely képes tárolni a kulcsszóhoz szükséges információkat. Erre a `struct` szerkezetet használjuk. Tehát definiálunk egy olyan `struct`-ot, melynek mezői maga a kulcsszó szövegrésze, ez természetesen egy `string` mező, illetve szükség van két, `TextPointer` típusú mezőre, melyek a kulcsszó kezdő és végpontjait reprezentálják.

```
struct Tag
{
    public TextPointer StartPosition;
    public TextPointer EndPosition;
    public string Word;
```

```
}
```

A listában való elhelyezéshez minden kulcsszóhoz létrehozunk egy struktúrát, melynek kezdő- és végpontját a `GetPositionAtOffset` függvénnyel állapítjuk meg, a már ismert kezdő- és végpont indexek segítségével, illetve a kulcsszó struktúra szövegrésze az ellenőrzés alatt álló szöveg lesz. Ezután már csak hozzá kell adnunk a listához az újonnan létrehozott struktúrát.

```
if (syntax.IsKnownTag(word))
{
    Tag t;
    t.StartPosition = run.ContentStart.GetPositionAtOffset(sIndex,
LogicalDirection.Forward);
    t.EndPosition = run.ContentStart.GetPositionAtOffset(eIndex + 1,
LogicalDirection.Backward);
    t.Word = word;
    CurrentKeyWords.Add(t);
}
```

Arra is kell figyelniünk, hogy az ellenőrzött szövegrész utolsó szava is formázás alá essen, ha az kulcsszó, ezért a jelenlegi megoldás mellett ehhez szükség van egy külön rutinra, amely az utolsó szót ellenőrzi.

Eljutottunk oda, hogy készen áll a feltöltött lista a formázásra. Ezzel is külön függvény foglalkozik:

```
void Format()
{
    textInput.TextChanged -= TextInput_TextChanged;

    TextRange r;
    for (int i = 0; i < CurrentKeyWords.Count; i++)
    {
        r = new TextRange(CurrentKeyWords[i].StartPosition,
CurrentKeyWords[i].EndPosition);
        r.ApplyPropertyValue(TextElement.ForegroundProperty,
new SolidColorBrush(keyWordColor));
        r.ApplyPropertyValue(TextElement.FontWeightProperty, FontWeights.Bold);
    }
    CurrentKeyWords.Clear();
    textInput.TextChanged += TextInput_TextChanged;
}
```

Fontos megemlíteni, hogy amikor formázási műveletet végzünk el egy bizonyos RichTextBox által tartalmazott szövegrészen, az is kiváltja a szöveg megváltozása eseményt, így olyan lenne, mintha formázás közben beleírnánk valamit. Mivel az egész folyamat a kiváltott eseménytől függ, újra meghívódna ez egész folyamat, ez pedig hibát eredményezne. Tehát a formázás idejére leiratkoztatjuk az eseményfigyelést, majd a végén vissza.

A formázás műveletében a már korábban kigyűjtött kulcsszavak listáját iterálunk végig, és egyenként elvégezzük rajtuk a formázási műveleteket.

Ahhoz, hogy kapcsolatot létesítsünk a RichTextBox szövege és a listánk között, egy TextRange objektumra van szükségünk. Mivel a listába elmentett kulcsszavak struktúrája tartalmazza a kulcsszavak eredeti szövegben elfoglalt kezdő- és végpozícióit, ezért ha a TextRange objektum konstruktorát ezzel a két mutatóval paraméterezzük fel, akkor ez az objektum az eredeti szövegbeli kulcsszóra fog mutatni. Majd ezen az objektumon elvégezzük a formázási műveleteket. Ezt a TextRange objektumon keresztül elérhető ApplyPropertyValue függvény segítségével végezhetjük el. A formázást elvégző függvény két paramétert vár, az első paraméter a formázás tulajdonsága, melyeket a TextElement osztályon keresztül érhetünk el, itt állíthatjuk be, hogy például a betűk színét akarjuk állítani. A másik paraméter a formázás értéke, jelen esetben a kiválasztott szín. Szinte mindenféle formázási lehetőség adott, esetünkben a színen kívül félkövérre állítjuk a kulcsszavakat.

Ha az összes kulcsszót megformáztuk a listából, akkor kiürítjük a listát. A formázás végeztével a parancsbeviteli mezőt megvalósító RichTextBox objektum TextChanged eseményére újra feliratkoztatjuk a *TextInput_TextChanged* függvényünket, hogy ezután újra elinduljon a formázási folyamat, ha megváltozik a szöveg.

RichTextBox komponens WPF alapokon

Mindazon helyeken a szoftverben, ahová kódot írhatunk, a beviteli mezőt, illetve a parancssort RichTextBox komponens reprezentálja. Mivel azonban a programot WPF alapokon valósítottuk meg, azt a komponenst kellett használnunk, amit ez a technológia nyújt. Itt azonban rögtön nehézségekbe ütköztünk, mivel ez nagyban eltér a már jól ismert Windows Forms által nyújtott RichTextBox komponensről.

A legnagyobb eltérés az, hogy az általunk használt kontroll tartalma nem csak egyszerű szöveg lehet, hanem lehetőséget nyújt paragrafusok, képek, táblázatok, természetesen szövegek és egyéb formátumok tárolására, illetve lehetővé teszi a formázásukat. Éppen ezért, mivel felépítése általános tartalom kezelésére képes, sokkal nehezebb is kezelni. Tulajdonképpen minden olyan objektumot képes kezelni, mely kompatibilis a FlowDocument típussal.

Elemeit többféleképpen is felsorolhatjuk, megadhatjuk. Tulajdonképpen definiálhatjuk statikusan, XAML kódban leírva, illetve, ami sokkal hatásosabb, dinamikus módon C# kódban. Programunkban az utóbbi megoldásra volt szükség.

A következő kódrészletben láthatjuk, ahogyan statikus módon, XAML nyelven elemeket adunk a RichTextBox kontrollhoz:

```

<RichTextBox Margin="10,10,0,13" Name="myRichTextBox" HorizontalAlignment="
Left" VerticalAlignment="Top" Width="500" Height="300">
    <FlowDocument>
        <Paragraph>
            This is a paragraph, this can be edited.
            <Bold>Go ahead.</Bold>
        </Paragraph>

        <Paragraph Foreground="Blue">
            This paragraph's foreground color had been changed to blue.
        </Paragraph>
    </FlowDocument>
</RichTextBox>

```

Látható, hogy már itt is beállíthatjuk az objektumok formázását, jelen esetben az egyik paragrafusban megjelenő szöveg félkövér lesz, a másikban a szöveg színe kék.

A legtöbb esetben azonban szükséges, hogy e tulajdonságokat dinamikus módon változtathassuk, erre láthatunk példát az alábbi C# kódrészletben:

```

private void CreateAndLoadRichTextBox()
{
    FlowDocument mcFlowDoc = new FlowDocument();

    Paragraph para = new Paragraph();
    para.Inlines.Add(new Run("This is a paragraph, this can be edited."));
    para.Inlines.Add(new Bold(new Run("Go ahead.")));

    mcFlowDoc.Blocks.Add(para);

    RichTextBox mcRTB = new RichTextBox();
    mcRTB.Width = 560;
    mcRTB.Height = 280;

    mcRTB.Document = mcFlowDoc;

    ContainerPanel.Children.Add(mcRTB);
}

```

Először is létrehozuk a saját FlowDocument objektumunkat. Majd létrehozunk egy Paragraph objektumot, és szöveget adunk neki. Ezt egy Run objektumként kapja meg, ennek az osztálynak van text paramétere, mely a szöveget képes tárolni. A szöveget tároló objektumot becsomagolhatjuk egy formázó objektumba, mely esetünkben a félkövérséget állító osztály. A FlowDocument objektumnak van egy, a paragrafusokat tartalmazó Blocks, listát megvalósító tulajdonsága, melyhez hozzáadjuk a létrehozott és szöveggel ellátott paragrafust.

Szükségünk van egy RichTextBox objektumra, melynek a tartalma az előbb létrehozott objektum lesz. Természetesen a létrehozott RichTextBox objektum tulajdonságait itt is beállíthatjuk, például a szélességet és a hosszúságot. Ezen objektum Documents tulajdonsága hivatott tárolni a komponens tartalmát, így odaadjuk ennek a FlowDocument objektumot, ezzel beállítva a szöveget tartalomként. Végül felvesszük a létrehozott RichTextBox kontrollt a programunk megfelelő pontján.

Mivel esetünkben az a legfontosabb, hogy a kód szövegét formázzuk, a RichTextBox objektumunkat képessé kell tenni szöveg bevitelére, kiolvasására, formázására. Az előbbi betöltési forma túl általános, többféle objektum is kezelhető, ezért csak szöveg betöltésére létezik egyszerűbben

kezelhető megoldás is. Ebből egyik megoldásként ismertetném azt az esetet, amikor egyszerű szöveget helyezünk el tartalomként, illetve még egy hasznos esetet, amikor egy szövegfájlból töltjük be a szöveget.

Az előbbi esetet a következő kódrészletben láthatjuk. Itt egy System.IO névtérben található MemoryStream objektumot hozunk létre először, átadva neki a megfelelő kódolással ellátott szöveget, majd a RichTextBox komponens már képes fogadni ezt az objektumot:

```
public void SetString(string text, RichTextBox rtb)
{
    MemoryStream stream = new MemoryStream(ASCIIEncoding.UTF8.GetBytes(text));
    rtb.Selection.Load(stream, DataFormats.Text);
}
```

A másik megoldásban az elérési útvonallal megadott fájlból tölthetjük be a szöveget. Ehhez szükség van, az ugyancsak System.IO névtérben helyet kapó FileStream osztályra, mely a szöveg kiolvasását végzi el. Hibakezelési okok miatt leellenőrizzük, hogy az elérési útvonalon megadott fájl valóban létezik-e. Egy TextRange objektum fogja képviselni a szöveg helyét és tartalmát a RichTextBox komponensben. A FileStream objektum által beolvashatjuk a fájlt, a TextRange objektum megfelelő metódusa pedig képes fogadni ezt a stream-et, és képes betölteni az így kapott szöveget. Az adatfolyam kapcsolatát le kell zárunk a művelet végén.

```
private void LoadTextDocument(string fileName)
{
    TextRange range;
    FileStream fStream;
    if (File.Exists(fileName))
    {
        range = new TextRange(RichTextBox1.Document.ContentStart,
RichTextBox1.Document.ContentEnd);
        fStream = new FileStream(fileName, FileMode.OpenOrCreate);
        range.Load(fStream, System.Windows.DataFormats.Text );
        fStream.Close();
    }
}
```

Nem csak a RichTextBox kontroll tartalmát kell tudnunk módosítani, írni bele, képesnek kell lennünk annak kiolvasására is. Sajnos a komponens általános működése miatt nincs beépített metódus, mely vissza tudja adni a szöveget, ezért ezt nekünk kell megkonstruálnunk. Ehhez nem kell mást tennünk mint egy TextRange objektumba betölteni a tartalmat, annak kezdő és végpontja alapján, ezután ez az objektum az aktuális szövegre fog mutatni. A TextRange objektum pedig már rendelkezik string típusú Text tulajdonsággal, melyből így már egyszerűen kinyerhető a szöveg. Ezt a folyamatot írja le az alábbi kódrészlet:

```
string ConvertRichTextBoxContentsToString(RichTextBox rtb)
{
    TextRange textRange = new TextRange(rtb.Document.ContentStart,
rtb.Document.ContentEnd);
    return textRange.Text;
}
```

Kapcsolatteremtés – Bluetooth

Szoftverünket, és a parancsokat végrehajtó robotot össze kellett kötnünk. Evidens volt, hogy vezeték nélküli technológiát kell alkalmaznunk, a könnyebb irányíthatóság, a pontosság és az egyszerűbb kezelhetőség érdekében. Erre a legmegfelelőbbnek a Bluetooth technológiát tartottuk. Felmerült még az infra, illetve a WiFi lehetősége, azonban az előbbi megoldásnak határokat szab a távolság és a közvetlen rálátás szükségessége, az utóbbi kiépítése pedig anyagiasabb megoldás lenne.



Kiválasztáskor mérlegelnünk kellett a távolságot, mely legalább 2-3 méter, a megbízhatóságot, az energiaellátást, az interferenciát, az adatátviteli sebességet, illetve az összeköttetés kiépítésének költségét. Mindezen szempontokat figyelembe véve, egyértelmű volt, hogy a Bluetooth technológia lesz a legjobb megoldás a kapcsolat megteremtésére.

A Bluetooth egy rádiófrekvenciával működő vezeték nélküli, eszközök összekötését, közöttük zajló adatátvitelt és kommunikációt, kialakítható hálózatot (PAN¹) megvalósító technológia. Full-duplex tulajdonságú, azaz a kommunikáció mindkét irányban, és egy időben történhet, pont-pont, illetve pont-multipont típusú összeköttetést is támogatva. A technológia a 2.402GHz és 2.480GHz közötti rádiófrekvenciás tartományban 79 vivőfrekvenciát használ 1MHz-es csatornaosztással, és frekvenciaugraltatásos módszert alkalmaz, 1600-as másodpercenkénti frekvenciaváltással. Ez azt jelenti, hogy kommunikáció közben egy algoritmus pszeudóvéletlen módon frekvenciát váltva, a használható 79 csatorna között váltogatva kommunikál, és mivel egy-egy csatornát csak nagyon rövid ideig használ, így ha eközben olyan csatornára is vált, amelyen már zajlik kommunikáció, az interferencia elhanyagolható mértékű és hatású.

Szabványtól függően változik a rendszer által áthidalható kommunikációs távolság, a leggyakoribb szabvány a Class 2, kb. 10 méter hatótávig használható, más szabványban erősítéssel akár 100 méterig is lehetséges a jelátvitel. Ennek távolsága és erőssége függ a közegtől is. Maximális átviteli sebessége 24 Mbit/s, és igencsak energiagazdaságos.

A kommunikáció felépítése két Bluetooth technológiát alkalmazó eszköz között párosítással jön létre. Mielőtt megkezdjük a kommunikációt két eszköz között, párosítanunk kell őket.

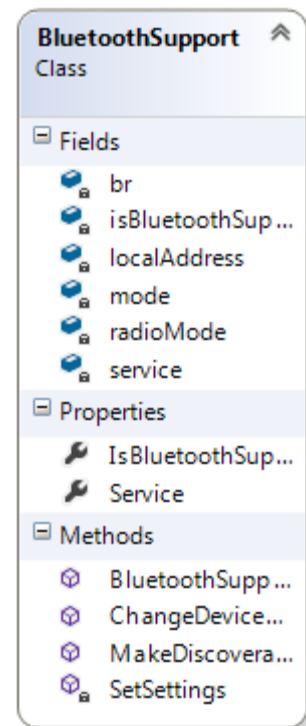
Ahhoz, hogy a szoftverünk Bluetooth-on, keresztül tudjon kommunikálni az Arduino egységgel, szükség van egy olyan API²-ra, mely képes kezelni ezt a technológiát, méghozzá .NET-es környezetben. Szerencsére 32feet.NET projekt támogatja a Bluetooth kapcsolat felépítését, a kommunikációt, és megfelelő interfészt biztosít a kezelésére. Ehhez nyilván szükség van egy-egy bluetooth adapterre is, az adó illetve a vevő félénél is. Szükségünk van két segédosztályra, melyek a kapcsolat kezelésére szolgálnak.

¹ Personal Area Network

² Application Programming Interface

BluetoothSupport osztály

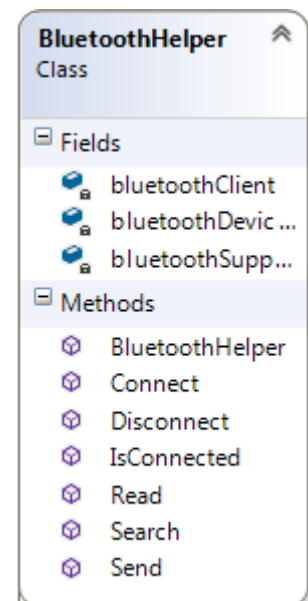
Ez az osztály szolgál arra, hogy általa a saját eszközünk Bluetooth beállításait számon tarthassuk, illetve módosítsuk azt. Osztályunk tartalmaz egy BluetoothRadio példányt, mely a Bluetooth adaptert azonosítja, amelyen keresztül kommunikálunk. A RadioMode példány segítségével állíthatjuk be, hogy az eszközünk éppen milyen állípotban van. Megkülönböztetünk felfedezhető és kikapcsolt módot, illetve alapállapotban a kapcsolódni képes mód, mely a már párosított eszközökhöz való kapcsolódást teszi lehetővé. Ha egy eszköz még nincs párosítva, az csak akkor tud csatlakozni hozzánk, ha az állapot felfedezhető módban van. Párosítás után már nem szükséges a felfedezhető mód a kommunikációhoz. Ezen kívül string-ként tároljuk a BluetoothRadio példány címét is, a *localAddress* mezőben, a könnyebb elérhetőség érdekében. Szükség van arra, hogy tudjuk, az eszközünk támogatja-e a Bluetooth technológiát, a logikai típusú *isBluetoothSupported* mezővel ezt jelezzük. A különféle állapotok közötti váltásra a *ChangeDeviceMode* metódus szolgál.



BluetoothHelper osztály

A BluetoothHelper osztály szolgál az egész kapcsolat létrehozásának, fenntartásának, és a kommunikáció lebonyolítására. Alapvető jelentőségű a három mező, egy BluetoothSupport példány, mellyel saját eszközünk paramétereit állíthatjuk, például a láthatóságot, egy BluetoothClient példány, melyen keresztül zajlik az egész kommunikáció, és egy BluetoothDeviceInfo típusú tömb, mely a felderített eszközöket, és a hozzájuk tartozó fontos információkat fogja tárolni.

A kapcsolat első lépése a felderítés. Erre való a *Search* függvény. Ez lényegében felfedezhetőre állítja az eszközünket, és a kliensen keresztül felderíti a közelben lévő Bluetooth-t támogató eszközöket, és felveszi a tömbbe.



Ahhoz, hogy kommunikálni tudjunk egy eszközzel, kapcsolódnunk kell hozzá. Ehhez kapcsolódni képes módban kell lennie az eszközünknek. A kapcsolódáshoz tudnunk kell annak az eszköznek a címét, amihez kapcsolódni akarunk, ezt egy BluetoothAddress objektum kezeli, és a kapcsolódni kívánt eszközt jellemző BluetoothDeviceInfo objektum DeviceAddress paraméterével olvasható ki, melyet a megfelelő konverzióval alakíthatunk címmé. A kapcsolódás a kliensen keresztül, a Connect metódus segítségével valósul meg. A kapcsolódást végző rutin az alábbi kódrészleten látható:

```
public void Connect(BluetoothDeviceInfo btDevice)
{
    bluetoothSupport.ChangeDeviceMode("Connectable");
}
```

```
        BluetoothAddress btAddress =
BluetoothAddress.Parse(btDevice.DeviceAddress.ToString());
        if (BluetoothSecurity.PairRequest(btAddress, "1234"))
        {
            try
            {
                bluetoothClient.Connect(new BluetoothEndPoint(btAddress,
bluetoothSupport.Service));
            }
            catch { }
        }
        else
        {
            throw new
Exception(App.Current.TryFindResource("notconnect").ToString());
        }
    }
}
```