LINQ

LINQ(*eng. Language-Integrated Query*) nam dozvoljava da vršimo upite nad podacima sa sintaksom nalik na **SQL**. Pre početka pisanja **LINQ** koda, neophodno je dodati **using System.Linq** klauzulu, kako bi mogli da se koriste članovi tog imenskog prostora.

Za početak je dat prikaz klasa, koje će biti korišćene u većini primera koji slede.

```
using System.Collections.Generic;

public class Customer
{
    public int ID { get; set; }
    public string Name { get; set; }
}
public class Order
{
    public int CustomerID { get; set; }
    public string Description { get; set; }
}

public static class Company
{
    static Company()
    {
        Customers = new List<Customer>
        {
            new Customer { ID = 0, Name = "May" },
            new Customer { ID = 1, Name = "Gary" },
            new Customer { ID = 2, Name = "Jennifer" }
        };
        Orders = new List<Order>
        {
            new Order { CustomerID = 0, Description = "Shoes" },
            new Order { CustomerID = 0, Description = "Purse" },
            new Order { CustomerID = 2, Description = "Headphones" }
        };
    }

    public static List<Customer> Customers { get; set; }
    public static List<Order> Orders { get; set; }
}
```

Customers i **Orders** kolekcije predstavljaju kolekcije objekata u memoriji. Kako bi postavljanje upita nad kolekcijom bilo što jednostavnije, klasa **Company** je statička, sa statičkim konstruktorom, koji inicijalizuje statička svojstva. Ukoliko se ovaj koncept apstrahuje, podaci bi mogli da se učitaju iz fajla ili baze podataka. Bez obzira na izvor podataka, osnovna **LINQ** sintaksa ostaje ista.

Upiti nad kolekcijama

Kako bi se postavio upit, neophodno je upotrebiti ključne reči **from** i **select**. Sintaksa izgleda kao **SQL**, što se može videti i iz sledećeg primera:

```
using System;
using System.Linq;
using System.Collections.Generic;

public class Program
{
    public void Main()
    {
        IEnumerable<Customer> customers =
            from cust in Company.Customers
            select cust;

        foreach (Customer cust in customers)
            Console.WriteLine(cust.Name);
    }
}
```

Upiti **LINQ-u-Objekat**(eng. *LINQ-to-Object*) rezultuju kolekcijom tipa **IEnumerable<T>**. U ovom slučaju, to je kolekcija **Customer** objekata. Ključna reč **from** specificira **range** promenljivu **cust**, koja predstavlja svaki objekat iz kolekcije. Kolekcija se specificira nakon ključne reči **in**.

Ključna reč **select** definiše šta se traži, odnosno tip upita. U ovom primeru, vraća se ceo objekat, što dovodi do toga da je dobijena kolekcija **customers** identična kolekciji **Company.Customers**. Ovo nije naročito korisno ukoliko se radi o **LINQ-u-**Objekat upitima, ali je vrlo korisno ako se podaci čitaju iz spoljnog izvora, kao što je baza podataka, gde je potrebno učitati kolekciju objekata u memoriju, zarad daljih manipulacija. Ključna reč **select** nam dozvoljava da izmenimo oblik povratnih podataka. Sledi primer upita koji nam vraća imena mušterija.

```
        IEnumerable<string> customers2 =
            from cust2 in Company.Customers
            select cust2.Name;
```

Kako bi došao do vrednosti svojstva **Name**, **select** klauzula koristi **cust2** promenljivu. Rezultat ovog upita vraća se u vidu kolekcije stringova. Ova transformacija **Customer** objekta u **string** naziva se **projekcija**.

Ponekad nam je potreban potpuno drugačiji objekat, gde taj objekat može biti definisan na sledeći način:

```
public class CustomerViewModel
{
    public string Name { get; set; }
}
```

Nova projekcija može biti napisana kao:

```
        IEnumerable<CustomerViewModel> customerVMs =
            from custVM in Company.Customers
            select new CustomerViewModel
            {
                Name = custVM.Name
            };
```

Ovde **select** kreira nove instance tipa **CustomerViewModel**, a zatim popunjava vrednost svojstva **Name** za svaku od tih instanci, upotrebljavajući vrednost **custVM.Name** svojstva, odgovarajućeg objekta. Rezultat ovog upita je kolekcija tipa **CustomerViewModel**.

Prethodni primer pretpostavlja da želimo da radimo sa kolekcijama određenog tipa. Međutim, ukoliko nam nije bitan tip kolekcije, i ne želimo da kreiramo novu klasu samo zbog jedne manipulacije, možemo upotrebiti anonimni tip, kao što je prikazano u sledećem primeru.

```
var customers3 =
    from cust3 in Company.Customers
    select new
    {
        Name = cust3.Name
    };
foreach (var cust3 in customers3)
    Console.WriteLine(cust3.Name);
```

Anonimni tipovi nemaju ime koje može da se koristi, iako **C#** može da kreira interno ime, za sopstvene potrebe. Kako bi se ovo zobišlo, za tip se koristi ključna reč **var**. Može se primetiti da projekcija, odnosno upit, koristi **new** bez tipa, što zapravo predstavlja anonimni tip. Unutar anonimnog tipa moguće je definisati bilo koja svojstva, samo ih je potrebno navesti. Za prolaz kroz kolekciju anonimnog tipa ponovo se koristi **var**.

Ukoliko je potrebno vratiti neku kolekciju kao povratni tip metode, potrebno je kreirati novi tip i projektovati rezultate upita na taj tip. Anonimni tipovi su osmišljeni za situacije gde je njihova upotreba ograničena njihovim postojanjem, odnosno njihova upotreba se svodi na isti deo koda u kom su i kreirani. Ključna reč **var** je dodata jeziku kako bi podržala ovaj scenario. Sledeći kod pokazuje uobičajen način za korišćenje **var** tipa.

```
var customer = new Customer();
```

Prethodna klauzula je kraća nego prilikom specificiranja tačnog tipa promenljive **customer**, što je ujedno i redudantno u ovom slučaju, jer je očigledno da je ona tipa **Customer**. Međutim, sledeći primer je manje očigledan:

```
var response = DoSomethingAndReturnResults();
```

Problem u prethodnoj klauzuli je taj što samo čitanje koda ne govori mnogo o tipu promenljive **response**. Ne zna se da li je to zaseban objekat ili kolekcija. U ovom slučaju, održavanje koda bi bilo jednostavnije ukoliko bi se specificirao tip promenljive.

Filtriranje podataka

Filtriranje podataka unutar kolekcije se može izvršiti upotrebom ključne reči **where**, kao u sledećem primeru.

```
var customers4 =
    from cust4 in Company.Customers
    where cust4.Name.Length > 3 && !cust4.Name.StartsWith("G")
    select cust4;

foreach (var cust4 in customers4)
    Console.WriteLine(cust4.Name);
```

U prethodnom kodu ime mušterije mora biti duže od 3 karatkera, što filtrira listu na **Gary** i **Jennifer**. Izraz sa desne strane operatora **&&** filtrira tu listu još više, uzimajući u obzir samo imena koja počinju slovom **'G'**.

U **LINQ-u-Objekat** upitima mogu se kreirati kompleksni uslovi unutar **where** klauzule, koristeći logičke operatore, zagrade za grupisanje i bilo kakvu logiku za filtriranje rezultata. Može se čak i pozvati druga metoda, koja bi procenila objekat koji se trenutno procenjuje. Rezultati **where** klauzule moraju biti **bool** tipa.

Uređivanje kolekcija

U **LINQ**-u, **orderby** klauzula nam dopušta da uredimo rezultujuću kolekciju po našoj želji. Naredni primer to demonstrira.

```
var customers5 =
    from cust5 in Company.Customers
    orderby cust5.Name descending
    select cust5;

foreach (var cust5 in customers5)
    Console.WriteLine(cust5.Name);
```

U ovom primeru, **orderby** klauzula sortira listu po imenima mušterija, na opadajući(*eng. descending*) način. Uobičajen redosled je rastući, što se i dobija ukoliko se izostavi ključna reč **descending** ili se specificira **ascending** umesto toga. Ispis rezultata je sledeći:

May

Jennifer

Gary

Povezivanje objekata

Ponekad postoje dve različite kolekcije objekata ili povezane tabele u bazi podataka, koje je potrebno povezati. Za ovo, koristi se ključna reč **join**.

```
var customerOrders =
    from cust in Company.Customers
    join ord in Company.Orders
        on cust.ID equals ord.CustomerID
    select new
    {
        ID = cust.ID,
        Customer = cust.Name,
        Item = ord.Description
    };

foreach (var custOrd in customerOrders)
    Console.WriteLine(
        $"Customer: {custOrd.Customer}, Item: {custOrd.Item}");
```

Nakon **from** klauzule, mogu se iskoristiti jedna ili više **join** klauzuli kako bi se pristupilo potrebnim tipovima. Ključna reč **on** nam dozvoljava da specificiramo ključ pomoću kojeg će kolekcije/tabele biti povezane. Ovde je upotrebljen normalan **join**, koji preskače bilo koji objekat tipa **Customer**, ukoliko za njega ne postoji odgovarajuća instanca klase **Order**. Sledeći primer nam dozvoljava da upotrebimo **join**, ali tako da nema preskakanja objekata:

```
var customerOrders2 =
    from cust in Company.Customers
    join ord in Company.Orders.DefaultIfEmpty()
        on cust.ID equals ord.CustomerID
    select new
    {

        ID = cust.ID,
        Customer = cust.Name,
        Item = ord.Description
    };

foreach (var custOrd2 in customerOrders)
    Console.WriteLine(
        $"Customer: {custOrd2.Customer}, Item: {custOrd2.Item}");
```

Ovde je razlika poziv **DefaultIfEmpty** metode, koja uključuje mušteriju imena **Gary**, iako ne postoji nijedna porudžbina koja odgovara njegovom **ID-u**.

Dodatne range promenljive

Pored **range** promenljivih uvedenih putem **from** klauzule, dodatne **range** promenljive se mogu dodati upotrebom drugih klauzuli ili ključnih reči, poput:

- **from**(ukoliko ima više od jedne)
- **let**
- **into**

Pre bilo kakvih primera, dat je prikaz klase **Ingredient**, koja će biti upotrebljavana u nastavku.

```
class Ingredient
{
    public string Name { get; set; }
    public int Calories { get; set; }
}
```

U okviru **LINQ** izraza za upit **let** klauzula dozvoljava uvođenje dodatne **range** promenljive. Ova dodatna **range** promenljiva zatim može biti upotrebljena u klauzulama koje slede.

U narednom kodu, **let** klauzula se koristi za uvođenje nove **range** promenljive **isDairy**, koja će biti **bool** tipa.

```
Ingredient[] ingredients =
{
    new Ingredient{Name = "Sugar", Calories=500},
    new Ingredient{Name = "Egg", Calories=100},
    new Ingredient{Name = "Milk", Calories=150},
    new Ingredient{Name = "Flour", Calories=50},
    new Ingredient{Name = "Butter", Calories=200}
};

IEnumerable<Ingredient> highCalDairyQuery =
                        from i in ingredients
                        let isDairy = i.Name == "Milk" || i.Name == "Butter"
                        where i.Calories >= 150 && isDairy
                        select i;


foreach (var ingredient in highCalDairyQuery)
{
    Console.WriteLine(ingredient.Name);
}
```

Na izlazu ovog koda se ispisuje:

Milk
Butter

U prethodnom kodu **isDairy** promenljiva je uvedena, a zatim upotrebljena u okviru **where** klauzule. Treba primetiti da originalna **range** promenljiva **i** ostaje dostupna unutar **select** klauzule.

U ovom primeru, nova **range** promenljiva je obična skalarna vrednost, ali **let** klauzula može biti upotrebljena i za uvođenje podniza. U narednom kodu, **range** varijabla **ingredients** nije skalarna vrednost, već niz stringova.

```
string[] csvRecipes =
{
    "milk,sugar,eggs",
    "flour,BUTTER,eggs",
    "vanilla,ChEEsE,oats"
};


var dairyQuery =
    from csvRecipe in csvRecipes
    let ingredients = csvRecipe.Split(',')
    from ingredient in ingredients
    let uppercaseIngredient = ingredient.ToUpper()
    where uppercaseIngredient == "MILK"   ||
          uppercaseIngredient == "BUTTER" ||
          uppercaseIngredient == "CHEESE"
    select uppercaseIngredient;
```

```
foreach (var dairyIngredient in dairyQuery)
{
    Console.WriteLine("{0} is dairy", dairyIngredient);
}
```

U prethodnom primeru treba primetiti da je upotrebljeno više **let** klauzuli, kao i dodatna **from** klauzula. Upotreba dodatne **from** klauzule predstavlja još jedan način za uvođenje nove **range** varijable u izraz za upit.

Ključna reč **into** dozvoljava deklarisanje novog identifikatora, koji može da sadrži rezultat iz **select** klauzule.

Naredni primer demonstrira upotrebu ključne reči **into** za kreiranje novog anonimnog tipa i njegovu kasniju upotrebu u ostatku izraza za upit.

```
Ingredient[] ingredients =
{
    new Ingredient{Name = "Sugar", Calories=500},
    new Ingredient{Name = "Egg", Calories=100},
    new Ingredient{Name = "Milk", Calories=150},
    new Ingredient{Name = "Flour", Calories=50},
    new Ingredient{Name = "Butter", Calories=200}
};


IEnumerable<Ingredient> highCalDairyQuery =
    from i in ingredients
    select new // anonymous type
            {
                OriginalIngredient = i,
                IsDairy = i.Name == "Milk" || i.Name == "Butter",
                IsHighCalorie = i.Calories >= 150
            }
    into temp
    where temp.IsDairy && temp.IsHighCalorie
    // cannot write "select i;" as into hides the previous range variable i
    select temp.OriginalIngredient;



foreach (var ingredient in highCalDairyQuery)
{
    Console.WriteLine(ingredient.Name);
}
```

Ovaj kod ispisuje sledeće:

Milk
Butter

Ovde treba primetiti da **into** sakriva prethodnu range varijablu **i**, što znači da **i** ne može biti upotrebljeno u poslednjoj **select** klauzuli. S druge strane, **let** klauzula ne sakriva prethodne promenljive, što znači da one mogu biti upotrebljene i kasnije.

Osim za predstavljanje rezultata iz **select** klauzule, ključna reč **into** može biti upotrebljena i za **group join** klauzulu. Ova klauzula može da stvori hijerarhijski grupisan rezultat, gde se stavke iz druge kolekcije povezuju sa stavkama iz prve kolekcije.

Za naredni primer definisane su klase **Recipe** i **Review**:

```
class Recipe
{
    public int Id { get; set; }
    public string Name { get; set; }
}

class Review
{
    public int RecipeId { get; set; }
    public string ReviewText { get; set; }
}
```

Ove dve klase modeluju činjenicu da recepti mogu da imaju 0, 1 ili više recenzija. **Review** klasa ima svojstvo **RecipeID**, koje sadrži identifikator recepta na koji se odnosi objekat **Review**.

Kako bi se shvatila uloga **group join** klauzule, prvo je prikazana primena redovne **join** klauzule.

```
Recipe[] recipes =
{
    new Recipe {Id = 1, Name = "Mashed Potato"},
    new Recipe {Id = 2, Name = "Crispy Duck"},
    new Recipe {Id = 3, Name = "Sachertorte"}
};

Review[] reviews =
{
    new Review {RecipeId = 1, ReviewText = "Tasty!"},
    new Review {RecipeId = 1, ReviewText = "Not nice :("},
    new Review {RecipeId = 1, ReviewText = "Pretty good"},
    new Review {RecipeId = 2, ReviewText = "Too hard"},
    new Review {RecipeId = 2, ReviewText = "Loved it"}
};


var query = from recipe in recipes
    join review in reviews on recipe.Id equals review.RecipeId
    select new // anonymous type
            {
                RecipeName = recipe.Name,
                RecipeReview = review.ReviewText
            };


foreach (var item in query)
{
    Console.WriteLine("{0} - '{1}'", item.RecipeName, item.RecipeReview);
}
```

Ovde se svaki objekat tipa **Recipe** povezuje sa njemu odgovarajućim objektima tipa **Review**, pri čemu se kreira novi objekat anonimnog tipa za svaki **Recipe-Review** par.

Ispis prethodnog koda je sledeći:

Mashed Potato - 'Tasty!'
Mashed Potato - 'Not nice :('
Mashed Potato - 'Pretty good'
Crispy Duck - 'Too hard'
Crispy Duck - 'Loved it'

Kao što se može primetiti, rezultat obične **join** klauzule ne povezuje sve recenzije sa odgovarajućim receptom, nego stvara zasebne parove. Takođe, može se primetiti i da recept **"Sachertorte"** ne postoji na izlazu, jer ne postoje recenzije koje se na njega odnose.

Za razliku od **join** klauzule, **group join** klauzula je u stanju da stvori grupe recenzija i poveže ih sa odgovarajućim receptima.

```
Recipe[] recipes =
{
    new Recipe {Id = 1, Name = "Mashed Potato"},
    new Recipe {Id = 2, Name = "Crispy Duck"},
    new Recipe {Id = 3, Name = "Sachertorte"}
};

Review[] reviews =
{
    new Review {RecipeId = 1, ReviewText = "Tasty!"},
    new Review {RecipeId = 1, ReviewText = "Not nice :("},
    new Review {RecipeId = 1, ReviewText = "Pretty good"},
    new Review {RecipeId = 2, ReviewText = "Too hard"},
    new Review {RecipeId = 2, ReviewText = "Loved it"}
};

var query =
    from recipe in recipes
    join review in reviews on recipe.Id equals review.RecipeId
    into reviewGroup
    select new // anonymous type
            {
                RecipeName = recipe.Name,
                Reviews = reviewGroup // collection of related reviews
            };

foreach (var item in query)
{
    Console.WriteLine("Reviews for {0}", item.RecipeName);
    foreach (var review in item.Reviews)
    {
        Console.WriteLine(" - {0}", review.ReviewText);
    }
}
```

Razliku između ovog i prethodnog primera predstavlja ključna reč **into**, koja stvara novu **range** promenljivu **reviewGroup**, a ona predstavlja niz recenzija koje odgovaraju **join** izrazu, što je u ovom slučaju **recipe.Id equals review.RecipeId**.

Kako bi se napravila izlazna kolekcija grupa, rezultat upita se projektuje u anoniman tip. Svaka instanca ovog anonimnog tipa u izlaznoj kolekciji predstavlja jednu grupu. Ovaj anonimni tip ima dva svojstva: **RecipeName**, koje dobija od elementa iz prve kolekcije, i **Reviews**, koja dolazi od rezultata **join** izraza.

Izlaz ovog koda proizvodi sledeći ispis:

Reviews for Mashed Potato
- Tasty!
- Not nice :(
- Pretty good
Reviews for Crispy Duck
- Too hard
- Loved it
Reviews for Sachertorte

Kod ovog ispisa se može primetiti hijerarhijska struktura rezultata upita, kao i to da recept **"Sachertorte"** postoji na izlazu, jer je za njega kreirana prazna grupa, iako u njoj ne postoje nikakve recenzije.

Upotreba standardnih operatora

Do sada je obrađena osnovna **LINQ** sintaksa, ali ima još mnogo toga što se može postići, prvenstveno upotrebom standardnih operatora. Naredni kodovi predstavljaju nabacane primere, koji demonstriraju upotrebu korisnih standardnih operatora upita. Operatori upita mogu biti upotrebljeni samostalno, ili u kombinaciji sa drugim oepratorima, kako bi se stvorili kompleksniji upiti.

Iako smo do sada radili samo sa **IEnumerable<T>** kolekcijama, gde **T** predstavlja projektovani tip upita, postoji mnogo standardnih operatora upita, koji vraćaju drugačije tipove kolekcija. Ovi operatori uključuju **ToList**, **ToArray** i **ToDictionary** operatore, kao i mnoge druge. Sledi primer koji pretvara rezultat u listu.

```
var custList =
    (from cust in Company.Customers
     select cust)
    .ToList();
custList.ForEach(cust => Console.WriteLine(cust.Name));
```

Prethodni kod zatvara upit unutar zagrada i zatim poziva **ToList** operator. **ForEach** metoda nad listom dozvoljava prosleđivanje lambde.

LINQ upiti koriste odloženo izvršavanje. Ovo znači da se upiti ne izvrše dok se ne pozove **foreach** petlja ili neki od standardnih operatora upita, koji zahtevaju podatke.

Viđeno je kako ključne reči **select**, **where**, **orderby** i **join** pomažu u izgradnji upita. Svaka od ovih klauzula ima svoj ekvivalent u vidu standardnog operatora upita. Ovi standardni operatori koriste tečnu sintaksu i obezbeđuju različite načine za izvršavanje istih upita. Slede primeri **Where** i **Select** operatora, koji kopiraju **where** i **select** klauzule.

```
var customers6 =
    Company.Customers
           .Where(cust => cust.Name.StartsWith("J"));
foreach (var cust6 in customers6)
    Console.WriteLine(cust6.Name);

var customers7 =
    Company.Customers.Select(cust => cust.Name);
foreach (var cust7 in customers7)
    Console.WriteLine(cust7);
```

Where lambda mora imati **bool** povratnu vrednost, a **Select** lambda nam dozvoljava da specificiramo projekciju.

Mogu se upotrebiti i set operacije poput **Union**, **Except** i **Intersect**. Naredni kod nam daje primer **Union** operatora.

```
var additionalCustomers =
    new List<Customer>
    {
        new Customer { ID = 1, Name = "Gary" }
    };
var customerUnion =
    Company.Customers
           .Union(additionalCustomers)
           .ToArray();
foreach (var cust in customerUnion)
    Console.WriteLine(cust.Name);
```

Dovoljno je samo proslediti kompatibilnu kolekciju i **Union** će proizvesti kombinovanu kolekciju svih objekata. U ovom primeru je upotrebljen i **ToArray** operator, koji vraća niz objekata tipa **Customer**.

Takođe, postoji i koristan set operatora za selektovanje: **First**, **FirstOrDefault**, **Single**, **SingleOrDefault**, **Last** i **LastOrDefault**. Naredni primer prikazuje upotrebu operatora **First**.

```
Console.WriteLine(Company.Customers.First().Name);
```

Jedina stvar koju treba napomenuti prilikom upotrebe **First** operatora je mogućnost javljanja **InvalidOperationException**-a, sa porukom **„Sequence contains no elements“**. Ova sekvenca sadrži elemente, ali to nije garantovano. Uvek je sigurnije koristiti operator sa **OrDefault** sufiksom, kao u sledećem primeru:

```
var empty =
    Company.Customers
           .Where(cust => cust.ID == 999)
           .SingleOrDefault();

if (empty == null)
    Console.WriteLine("No values returned.");
```

Prethodni primer ispisuje **„No values returned“**. S obzirom da nema mušterije sa identifikatorom **999**, **SingleOrDefault** operator vraća **null** vrednost, što je uobičajena vrednost za referentni tip objekta.

LINQ u XML

Osnovne funkcionalnosti **"LINQ u XML"** interfejsa omogućavaju kreiranje, modifikaciju, upite nad podacima i čuvanje **XML** podataka.

Na nekom višem nivou, **"LINQ u XML"** se sastoji od dva ključna elementa, koji su dizajnirani tako da dobro rade zajedno:

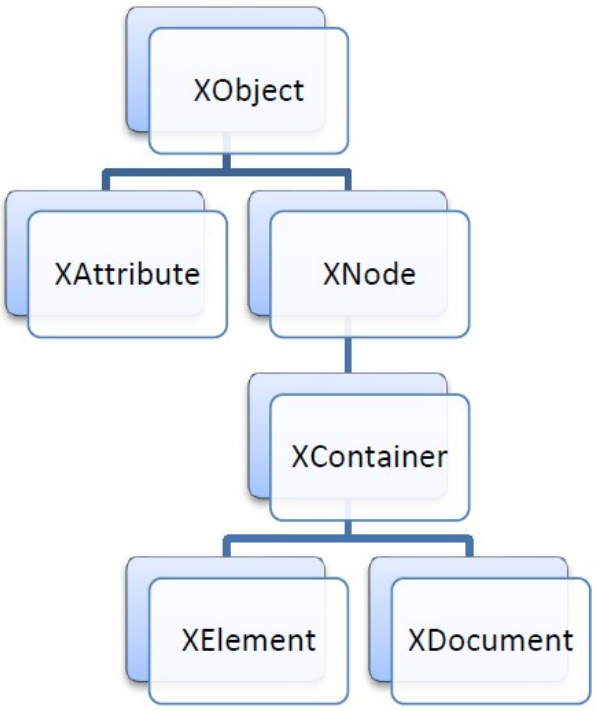
- Objektni model **XML** dokumenta(*eng. XML document object model, X-DOM*)
- Dodatne "LINQ u XML" operacije upita

X-DOM

X-DOM predstavlja set **.NET** tipova za dinamički pristup **XML** dokumentima. Iako se **X-DOM** tipovi mogu upotrebljavati nezavisno od **LINQ** upita, oni su osmišljeni tako da omogućavaju jednostavnu interakciju prilikom upotrebe **LINQ**-a.

Tipovi od kojih je sačinjen **X-DOM** se koriste za predstavljanje strukture **XML** dokumenta ili jednog njegovog dela. Ovi tipovi predstavljaju dinamički(*in-memory*) model strukture i sadržaja **XML** podataka.

Na narednoj slici su predstavljeni ključni tipovi.



XContainer predstavlja abstraktnu klasu iz koje se izvode **XEelement** i **XDocument**. Zbog nasleđivanja, obe izvedene klase mogu da sadrže nula, jedan ili više **XNode** objekata.

XElement predstavlja element u **XML** dokumentu, poput **<ingredient>** elementa. **XElement** može da sadrži vrednost, kao što je **"Butter"**, kao i nekoliko **XAttributes** objekata.

XDocument proširuje korenski **XElement** tako da sadrži i dodatne informacije, poput **XML** verzije i zaglavlja. Ukoliko ove informacije nisu potrebne, dovoljno je koristiti **XElement**.

XAttribute predstavlja atribute **XML** elemenata. Na primer, **calories** atribut bi mogao biti predstavljen kao **XAttribute** objekat: **<ingredient calories="500">**

XNode objekat predstavlja koncept pripadanja roditeljskom čvoru(**XElement**), do kojeg može da navigira. Za razliku od **XContainer** klase, nema dečje čvorove.

XName ne predstavlja deo hijerarhije, ali se koristi u mnogim metodama kako bi se određeni čvor pronašao unutar **XML** dokumenta. Koristi se za predstavljanje imena **XML** elementa ili atributa.

Kreiranje X-DOM objekata

Kako bi se **XElement** kreirao iz stringa, može se iskoristiti statička **Parse** metoda, kao u narednom kodu.

```
string xmlString = @"<ingredients>
                        <ingredient>Sugar</ingredient>
                        <ingredient>Milk</ingredient>
                        <ingredient>Butter</ingredient>
                    </ingredients>";

XElement xdom = XElement.Parse(xmlString);

Console.WriteLine(xdom);
```

Ovo proizvodi sledeći ispis:

```
<ingredients>
      <ingredient>Sugar</ingredient>
      <ingredient>Milk</ingredient>
      <ingredient>Butter</ingredient>
</ingredients>
```

Za kreiranje **XElement** objekta na osnovnu fizičkog **XML** fajla, može se upotrebiti **XElement.Load** metoda, uz specificiranje putanje do fajla koji se učitava.

Tipovi koji sačinjavaju **X-DOM** mogu biti instancirani, kao i redovni **.NET** tipovi. Kako bi se kreirao **X-DOM** koji predstavlja prethodni **<ingredients> XML**, može se napsiati sledeći kod.

```
XElement ingredients = new XElement("ingredients");

XElement sugar = new XElement("ingredient", "Sugar");
XElement milk = new XElement("ingredient", "Milk");
XElement butter = new XElement("ingredient", "Butter");

ingredients.Add(sugar);
ingredients.Add(milk);
ingredients.Add(butter);

Console.WriteLine(ingredients);
```

Ovaj kod proizvodi ispis koji je potpuno isti prethodnom. Iako je struktura kreiranog **XML** koda jednostavna, sam način njegovog kreiranja je dosta nejasan, zbog čega se kao alternativa koristi funkcionalna konstrukcija.

Sledeći kod prikazuje kreiranje identične **<ingredients> XML** strukture, ali ovaj put uz upotrebu funkcionalne konstrukcije.

```
XElement ingredients =
    new XElement("ingredients",
        new XElement("ingredient", "Sugar"),
        new XElement("ingredient", "Milk"),
        new XElement("ingredient", "Butter")
    );

Console.WriteLine(ingredients);
```

U prethodnom primeru se može videti da konstruktor **XElement** tipa dozvoljava specificiranje proizvoljnog broja sadržanih elemenata, nakon navođenja imena elementa. Potpis ove verzije konstruktora je: **public XElement(XName name, params object[] content)**

Jedna od beneficija funkcionalne konstrukcije jeste ta što se rezultati **LINQ** upita mogu projektovati u **X-DOM**.

Naredni kod prepravlja prethodni primer, kako bi se **X-DOM** popunio pomoću rezultata **LINQ** upita. Rezultat je ista **XML** struktura.

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Milk", Calories = 150},
    new Ingredient {Name = "Butter", Calories = 200}
};

XElement ingredientsXML =
    new XElement("ingredients",
        from i in ingredients
        select new XElement("ingredient", i.Name,
                            new XAttribute("calories", i.Calories))
    );

Console.WriteLine(ingredientsXML);
```

U ovom primeru se može primetiti dodatak **XAttribute** objekta svakom **<ingredient>** elementu. Ovo je moguće jer **XElement** konstruktor prihvata neograničen broj sadržanih elemenata, čak i ako su u pitanju različiti tipovi. Ovaj princip nam omogućava istovremeno postavljanje tekstualnog sadržaja **XElement** objekta, kao i dodelu već pomenutog atributa.

Upiti nad X-DOM objektima

Upiti i navigacija kroz **X-DOM** objekte su implementirani na dva načina. Prvi način predstavljaju metode koje pripadaju samim **X-DOM** tipovima. Drugi način predstavlja set dodatnih operatora upita, koji su definisani u okviru **Linq.Extensions** klase. Metode **X-DOM** tipova vraćaju **IEnumerable** sekvence, nad kojima se dalje radi uz pomoć operatora upita.

Jednostavan primer demonstrira obe implementacije.

```
XElement xmlData = XElement.Load("recipes.xml");

// Here Descendants() is an instance method of XContainer
var allrecipes = xmlData.Descendants("recipe");

// Here Descendants() is a query operator extension method
var allIngredients = allrecipes.Descendants("ingredient");
```

U prethodnom kodu prva **Descendants** metoda pripada **XElement** klasi, odnosno baznoj klasi **XContainer**. Ova metoda vraća **IEnumerale<XElement>** sekvencu i smešta je u promenljivu **allRecipes**.

Naredni poziv **Descendants** metode se odnosi na operator upita koji ima sledeći potpis: **public static IEnumerable<XElement> Descendants<T>(this IEnumerable<T> source, XName name) where T : XContainer**

Pronalaženje dečjih čvorova

Postoji nekoliko svojstava i metoda koje omogućavaju pronalazak dečjih čvorova.

Metoda(parametar)/Svojstvo	Povratna vrednost	Radi nad
FirstNode	XNode	XContainer
LastNode	XNode	XContainer
Element(XName)	XElement	XContainer
Nodes()	IEnumerable<XNode>	XContainer, IEnumerable<XContainer>
DescendantNodes()	IEnumerable<XNode>	XContainer, IEnumerable<XContainer>
Elements()	IEnumerable<XElement>	XContainer, IEnumerable<XContainer>
Elements(XName)	IEnumerable<XElement>	XContainer, IEnumerable<XContainer>
Descendants()	IEnumerable<XElement>	XContainer, IEnumerable<XContainer>
Descendants(XName)	IEnumerable<XElement>	XContainer, IEnumerable<XContainer>

HasElements	Boolean	XElement
DescendantNodesAndSelf()	IEnumerable<XNode>	XElement, IEnumerable<XElement>
DescendantsAndSelf()	IEnumerable<XElement>	XElement, IEnumerable<XElement>
DescendantsAndSelf(X Name)	IEnumerable<XElement>	XElement, IEnumerable<XElement>

Naredni kod demonstrira izvlačenje prvog **<ingredient>** elementa koji ima dečje elemente. **Descendants** metoda se koristi za izvlačenje svih **<ingredient>** elemenata, dok se standardni operator **First** koristi za izvlačenje prvog elementa koji ima bar jedan dečji element. Provera da li neki element ima dečje elemente se vrši pomoću **HasElements** svojstva.

```
XElement xmlData = XElement.Load("recipes2.xml");

XElement firstIngredientWithSubElements = xmlData.Descendants("ingredient")
                                                  .First(x => x.HasElements);
```

Pronalaženje roditeljskih čvorova

X-DOM tipovi koji su izvedeni iz **XNode** tipa sadrže **Parent** svojstvo koje vraća roditeljski **XElement**.

Metoda(parametar)/Svojstvo	**Povratna vrednost**	**Radi nad**
Parent	XElement	XNode
Ancestors()	IEnumerable<XElement>	XNode, IEnumerable<XNode>
Ancestors(XName)	IEnumerable<XElement>	XNode, IEnumerable<XNode>
AncestorsAndSelf()	IEnumerable<XElement>	XElement, IEnumerable<XElement>
AncestorsAndSelf(XName)	IEnumerable<XElement>	XElement, IEnumerable<XElement>

Naredni primer pokazuje kako se **Parent** svojstvo koristi za penjanje uz **XML** hijerarhiju. Ovaj primer pronalazi prvi **<ingredient>** element koji ima dečje elemente, a zatim se penje za jedan nivo kako bi se dobio **<ingredients>** element. Nakon ovoga penje se za još jedan nivo i dolazi do **<recipe>** elementa.

```
XElement xmlData = XElement.Load("recipes2.xml");

XElement recipe = xmlData.Descendants("ingredient")
                         .First(x => x.HasElements)
                         .Parent // <ingredients>
                         .Parent; // <recipe name="Cherry Pie">
```

Pronalaženje bratskih čvorova

Naredna tabela prikazuje metode unutar **XNode** klase, koje se koriste za rad sa čvorovima istog nivoa.

Metoda(parametar)/Svojstvo	Povratna vrednost
IsBefore(XNode)	Boolean
IsAfter(XNode)	Boolean
PreviousNode	XNode
NextNode	XNode
NodesBeforeSelf()	IEnumerable<XNode>
NodesAfterSelf()	IEnumerable<XNode>
ElementsBeforeSelf()	IEnumerable<XElement>
ElementsBeforeSelf(XName)	IEnumerable<XElement>
ElementsAfterSelf()	IEnumerable<XElement>
ElementsAfterSelf(XName)	IEnumerable<XElement>

Naredni kod prikazuje upotrebu **NextNode** i **IsBefore** metoda. Takođe, demonstrira se i upotreba standardnih operatora u **"LINQ u XML"** upitima.

```
XElement xmlData = XElement.Load("recipes2.xml");

var applePieIngredients =
   xmlData.Descendants("recipe")
          .First(x => x.Attribute("name").Value == "Apple Pie")
          .Descendants("ingredient");
```

```
var firstIngredient = applePieIngredients.First(); // Sugar
var secondIngredient = firstIngredient.NextNode; // Apples
var lastIngredient = applePieIngredients.Skip(2).First(); // Pastry

var isApplesBeforeSugar =
        secondIngredient.IsBefore(firstIngredient); // false
```

Pronalaženje atributa

XElement objekat može imati nekoliko atributa. Naredna tabela prikazuje metode za rad sa atributima definisane unutar **XElement** klase.

Metoda(parametar)/Svojstvo	Povratna vrednost
HasAttributes	Boolean
Attribute(XName)	XAttribute
FirstAttribute	XAttribute
LastAttribute	XAttribute
Attributes()	IEnumerable<XAttribute>
Attributes(XName)	IEnumerable<XAttribute>

Naredni kod prikazuje upotrebu **Attribute(XName)** metode za pronalaženje određenog **XAttribute** objekta i izvlačenje njegove vrednosti pomoću **Value** svojstva. Takođe, prikazana je i upotreba **FirstAttribute** metode za izvlačenje prvo-definisanog atributa, kao i upotreba standardnih operatora upita nad **IEnumerable<XAttribute>** sekvencom.

```
var xml = @"
<ingredients>
    <ingredient name='milk' quantity='200' price='2.99'  />
    <ingredient name='sugar' quantity='100' price='4.99'  />
    <ingredient name='safron' quantity='1' price='46.77'  />
</ingredients>";

XElement xmlData = XElement.Parse(xml);

XElement milk =
    xmlData.Descendants("ingredient")
           .First(x => x.Attribute("name").Value == "milk");


XAttribute nameAttribute = milk.FirstAttribute; // name attribute

XAttribute priceAttribute = milk.Attribute("price");
```

```
string priceOfMilk = priceAttribute.Value; // 2.99

XAttribute quantity = milk.Attributes()
                          .Skip(1)
                          .First(); // quantity attribute
```