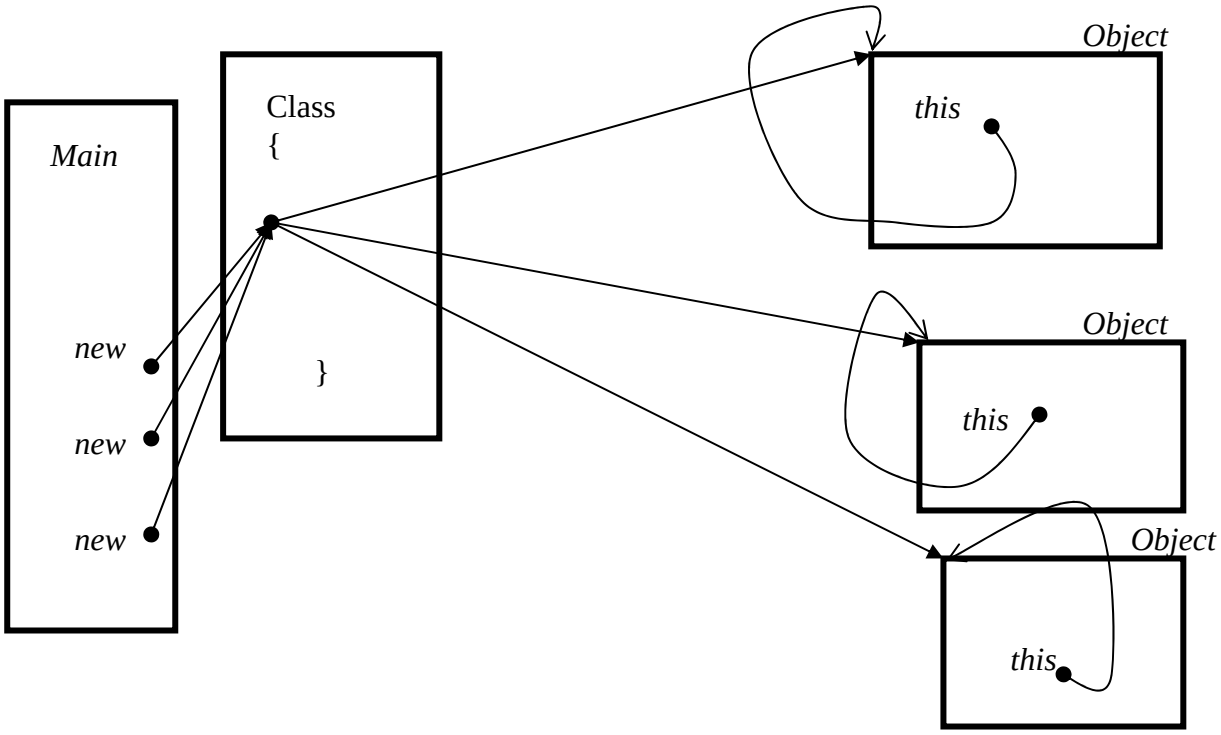
თ. ბახტაძე

ობიექტზე ორიენტირებული პროგრამირების ენა C#



"ტექნიკური უნივერსიტეტი"

საქართველოს ტექნიკური უნივერსიტეტი

თ. ბახტაძე

ობიექტზე ორიენტირებული პროგრამირების ენა C#

თბილისი
2006

ეს წიგნი ამჟამად ყველაზე თანამედროვე ოპერატორული ენის "სი შარპის" სახელმძღვანელოა ქართულ ენაზე. სახელმძღვანელოში დაწვრილებით არის აღწერილი ენის, როგორც არაობიექტური, ასევე ობიექტური ნაწილები. იგი განკუთვნილია, როგორც პროგრამირების შესწავლის დამწყებთათვის, ასევე მათთვის, ვისაც გამოცდილება აქვს სხვა, როგორც ობიექტზე ორიენტირებულ (Java, C++, Visual Basic), ისე ობიექტზე არაორიენტირებულ ენებში (Turbo Pascal, Basic, Fortran). სახელმძღვანელოში განხილულია ენის ყველა ასპექტი. მეორე მხრივ, ეს ენა შედის Visual Studio Net სისტემის შემადგენლობაში, რომლის შესაძლებლობები განხილულია მინიმალურად, ვინაიდან სცილდება სახელმძღვანელოს ფარგლებს.

უნდა აღინიშნოს, რომ ობიექტზე ორიენტირებული კონცეფცია ფაქტიურად გულისხმობს ღია ენის პრინციპს, ამდენად ყველა კლასის განხილვა შეუძლებელია სახელმძღვანელოში, ეს კონკრეტული დოკუმენტაციის საქმეა. განხილულია მხოლოდ ის კლასები, რომელიც ავტორის აზრით სავალდებულოა ენის ცნებების აღსაწერად. სახელმძღვანელოს ფარგლებს სცილდება ასევე ენის რეალიზაციის საკითხები. სახელმძღვანელო განკუთვნილია სტუდენტებისა და მაგისტრანტებისთვის, რომლეთაც სურთ შეისწავლონ ობიექტზე ორიენტირებული ენის საფუძვლები.

C# ანუ "სი შარპი" წარმოადგენს ობიექტზე ორიენტირებულ თანამედროვე პროგრამირების ენას. კერძოდ, ვისაც ნასწავლი აქვს Turbo Pascal, მას გაუადვილდება მასალის არაობიექტური ნაწილის ათვისება, ხოლო ვინც საფუძვლიანად იცის Java, იოლად აითვისებს ორივე ნაწილს. სახელმძღვანელოში უხვად არის მოყვანილი მაგალითები, რომლებიც თვალსაჩინოს ხდის მასალის აღქმას. პროგრამების გაშვება შესაძლებელია როგორც Visual Studio Net-ის, ასევე Net FrameWork სისტემების საშუალებით. სახელმძღვანელო დახმარებას გაუწევს სტუდენტს ნებისმიერი პროგრამული პროდუქტის ათვისებასა და რეალიზაციაში.

რეცენზენტები
პროფესორი ა. ფრანგიშვილი
პროფესორი გ. სურგულაძე

სარჩევი

შესავალი 4
თავი I: მარტივი პროგრამა C#-ზე 5
თავი II: ცვლადები, ტიპები, კონსტანტები და ოპერაციები 11
თავი III: მართვის ოპერატორები 30
თავი IV: ციკლის ოპერატორები 38
თავი V: მეთოდები 44
თავი VI: სახელსივრცეები 53
თავი VII: შესავალი კლასებში 59
თავი VIII: კლასის მემკვიდრეობითობა 62
თავი IX: პოლიმორფიზმი 66
თავი X: თვისებები 70
თავი XI: ინდექსერები 74
თავი XII: სტრუქტურები 79
თავი XIII: ინტერფეისები 81
თავი XIV: შესავალი დელეგატებსა და მოვლენებში 84
თავი XV: გამონაკლისი შემთხვევების დამუშავება 96
თავი XVI: ატრიბუტების გამოყენება 100
თავი XVII: ჩამოთვლები 107
თავი XVIII: ოპერაციათა გადატვირთვა 114
თავი XIX: დაუცველი კოდი 119
თავი XX: მოდიფიკატორები 124
ლიტერატურა 129

შესავალი

ობიექტზე ორიენტირებული პროგრამირების კონცეფციის შესწავლა ძალზე მნიშვნელოვანია, ვინაიდან მასზეა დაფუძნებული თანამედროვე პროგრამული პროდუქტები. ეს კონცეფცია უფრო ნათელს ხდის სამომხმარებლო პროგრამებს და საშუალებას იძლევა იოლად გადავერთოთ ობიექტზე ორიენტირებული ნებისმიერი ენის გამოყენებაზე. C# (სი შარპი) წარმოადგენს ობიექტზე ორიენტირებული პროგრამირების ერთ-ერთ ყველაზე თანამედროვე ენას [1-7]. იგი ყველაზე მეტად ჰგავს Java, C++ და visual basic პროგრამირების ენებს. C# დამუშავებულია კორპორაცია Microsoft-ში და აქტიურად ვითარდება, როგორც Visual Studio Net-ის ერთ-ერთი შემადგენელი ნაწილი. ამ ენის საფუძვლიანი ათვისება საშუალებას იძლევა უფრო ღრმად გავერკვეთ ნებისმიერ თანამედროვე პროგრამულ პროექტში. სახელმძღვანელოში პროგრამის ცვლადები აკრეფილია ე. წ. აკადნუსხური ფონეტიკური შრიფტით, რაც ავტორის აზრით ქართველი მკითხველისთვის უფრო აშკარას ხდის განსხვავებას ენის საკვანძო სიტყვებისგან.

ზოგიერთი ექსპერტის შეფასებით მასში 70% Java, 10% C++ , 5% Visual Basic და 15 % ახალია. თუმცა, როგორც ყველაფერი, ახალიც საკმაოდ პირობითია, ვინაიდან ერთ-ერთ სიახლედ *goto* ოპერატორი მოიაზრება. სახელმძღვანელოში დაწვრილებით არის აღწერილი ენის როგორც არაობიექტური, ასევე ობიექტური ნაწილები. ამის გამო იგი განკუთვნილია როგორც დამწყებთათვის, ასევე გარკვეული გამოცდილების მქონე სტუდენტებისთვის. თითოეულ თავში წარმოდგენილი მაგალითები შემოწმებულია კომპიუტერზე.

თავი I

მარტივი პროგრამა C#-ზე

ამ თავში, რამდენიმე მარტივი მაგალითის საშუალებით, გთავაზობთ შესავალს C#-ში. ქვემოთ ჩამოთვლილია ამ თავში განხილული საკითხები:

- C#-პროგრამის ძირითადი სტრუქტურა;
- სახელსივრცეების *(namespaces)* ცნება;
- კლასის ცნება;
- *Main*-მეთოდის დანიშნულება;
- ბრძანების სტრიქონიდან მონაცემთა მიღება;
- კონსოლიდან (კლავიატურა+დისპლეი) შეყვანა/გამოყვანა (I/O).

ვიწყებთ დასრულებული მარტივი C#-პროგრამით. ეს მაგალითი ხსნის რამდენიმე ძირითად კონცეფციას, რომელიც გვხვდება ყველა C#-პროგრამაში, რომელსაც თქვენ შეადგენთ. მაგალითად, უნდა იცოდეთ, თუ როგორ იწყება პროგრამა, როგორი სტრუქტურა აქვს კოდს და უფრო ახლოს გაეცნოთ C#-პროგრამის სინტაქსს (ჩაწერის ზოგადი წესი).

ამონაბეჭდი 1-1. მარტივი პროგრამა: MogesalmebaT.cs

```
// სახელსივრცის აღწერა
using System;
// პროგრამის საწყისი კლასი
class MogesalmebaT_CS
{   // Main იწყებს პროგრამის შესრულებას
   public static void Main()
   {double a=3,b=4,c; // მართკუთხა სამკუთხედის გვერდებია
      // კონსოლზე დაბეჭდვა
      Console.WriteLine("მოგესალმებათ C შარპი");
c=Math.Sqrt(Math.Pow(a,2)+Math.Pow(b,2)); // ჰიპოტენუზის გამოთვლა
Console.WriteLine("კათეტი a={0}, კათეტი b={1}, ჰიპოტენუზა c={2} ",a,b,c);
} }
```

პროგრამის გამოსავალი:

```
მოგესალმებათ C შარპი
კათეტი a=3, კათეტი b=4, ჰიპოტენუზა c=5
```

ამონაბეჭდში 1-1 მოცემული პროგრამა შეიცავს 4 ძირითად ელემენტს: სახელსივრცის აღწერას, კლასს, *Main*-მეთოდსა და პროგრამის ბრძანებებს. იგი შეიძლება შევასრულოთ ორი გზით:

1. NET FRAMEWORK-ის გამოყენებით, რომელიც უფასოა და ინტერნეტიდან შეიძლება გადმოვტვირთოთ და დავაინსტალიროთ. ამის შემდეგ საწყისი ტექსტი, რომელიც მომზადებულია რომელიმე რედაქტორში, მაგალითად, (NOTEPAD), შეიძლება კომპილირებულ იქნეს შემდეგი ბრძანების სტრიქონით:

csc.exe MogesalmebaT.cs,

სადაც csc C# პროგრამა-კომპილატორის სახელია, cs – C#-ზე დაწერილი საწყისი ფაილის სახელის გაფართოება, ხოლო *MogesalmebaT.cs* - საწყისი ფაილის სახელი.

ზემოთ მოყვანილი ბრძანების შესრულების შედეგად მივიღებთ ფაილს სახელად *MogesalmebaT.exe*, რომელიც შეიძლება შესრულდეს. სხვა პროგრამების კომპილირება ანალოგიურად ხდება, *MogesalmebaT.cs*.-ის მათი სახელით ჩანაცვლების გზით. "csc"-ბრძანების არჩევითი პარამეტრების გასაგებად ბრძანების სტრიქონში აკრიფეთ "csc -help". ფაილის და კლასის სახელები შეიძლება სხვადასხვა იყოს.

2. *VISUAL STUDIO NET* –ის საშუალებით. ცხადია, იგი დაინსტალირებული უნდა იყოს. ამის შემდეგ ვიძახებთ მას: *Start=>Programs=> Microsoft Visual Studio .NET* მენიუს ძირითად ფანჯარაში: *File=>New=>Project* და ვირჩევთ შაბლონს: *ConsoleApplication1=>ENTER*. მიღებული ფაილის Class1.cs ტექსტს ვცვლით ჩვენი პროგრამის ტექსტით. პროგრამის შესასრულებლად: Debug=>Start Without Debuging. აღვნიშნოთ, რომ ამ სისტემასთან მუშაობისას ვიყენებთ Windows-თან მუშაობის ყველა სტანდარტულ საშუალებას. კერძოდ, კონტექსტურ მენიუს (მაუსის მარჯვენა კლავიშის გამოყენება)და ა.შ.

შენიშვნა VS.NET-მომხმარებელთათვის: Visual Studio .NET-ით მიღებული პროგრამის გაშვებისას ეკრანი შეიძლება იხსნებოდეს და იხურებოდეს სწრაფად. ამის თავიდან ასაცილებლად *Main*-მეთოდის ბოლოს დავამატოთ შემდეგი კოდი:

```
// აკავებს კონსოლის ფანჯარას გაქრობისგან
// როდესაც ვუშვებთ VS.NET
Console.ReadLine();
```

ყურადღება უნდა მიექცეს იმას, რომ C# მგრძნობიარეა დიდი და პატარა ასოების მიმართ. სიტყვა *"Main"* არ არის იგივე, რაც *"main"*. ეს სხვადასხვა იდენტიფიკატორებია.

სახელსივრცის აღწერა using System გვიჩვენებს, რომ მივუთითებთ სახელსივრცეს *System*. სახელსივრცეები შეიცავენ კოდების ჯგუფებს (მაგალითად, კლასებს), რომელნიც შეიძლება გამოძახებულ იქნეს C#-პროგრამებიდან. *using System* აღწერის საშუალებით პროგრამას ვეუბნებით, რომ მას შეუძლია მიუთითოს კოდი სახელსივრცეში *System* სიტყვა *System*-ის გამეორების გარეშე ყოველი მითითებისთვის. ეს საკითხი უფრო დეტალურად განხილულია მე-6 თავში.

კლასის აღწერა *class MogesalmebaT_CS* შეიცავს იმ მონაცემებისა და მეთოდების განსაზღვრებებს, რომელთაც თქვენი პროგრამა იყენებს შესასრულებლად. კლასი არის თქვენი პროგრამის ერთ-ერთი განსხვავებული ელემენტი, რომელიც გამოიყენება

ობიექტების აღსაწერად. ჩვენს შემთხვევაში კონკრეტულ კლასს არ აქვს მონაცემები, მაგრამ აქვს ერთი მეთოდი, რომელიც განსაზღვრავს ამ კლასის ქცევას. უფრო ამომწურავად კლასები განხილულია მე-7 თავში. ერთი მეთოდი *MogesalmebaT_CS class*-ის შიგნით აღწერს, თუ რა უნდა შეასრულოს ამ კლასმა. მეთოდის სახელია *Main*. მისი სახელი დაჯავშნილია, როგორც პროგრამის საწყისი წერტილი. *Main*-ს ხშირად უწოდებენ "შესვლის წერტილს" და თუ კომპილატორიდან როდისმე მიიღებთ შეტყობინებას შეცდომის შესახებ, რომელიც გვეუბნება, რომ ვერ იპოვნა "შესვლის წერტილი", ეს ნიშნავს, რომ თქვენ სცადეთ პროგრამის კომპილირება და შესრულება *Main*-მეთოდის გარეშე.

სიტყვა *Main*-ის წინ დგას მოდიფიკატორი *static*. ეს მოდიფიკატორი აღნიშნავს, რომ მოცემული მეთოდი არსებობს მხოლოდ ამ სპეციფიკურ კლასში, განსხვავებით კლასის ეგზემპლარისა. ეს საჭირო იმისთვის, რომ როცა პროგრამა იწყება, არც ერთი ობიექტ-ეგზემპლარი არ არსებობს. უფრო დეტალურად კლასების, ობიექტებისა და ეგზემპლარების შესახებ მოთხრობილია მე-7 თავში.

ყოველ მეთოდს აქვს შედეგის უკან დაბრუნების ტიპი. ამ შემთხვევაში ეს არის *void*, რომელიც გულისხმობს, რომ *Main*-ი უკან არ აბრუნებს მნიშვნელობას (ანუ პროცედურაა და არა ფუნქცია). ყოველ მეთოდს აქვს აგრეთვე პარამეტრების (არგუმენტების) ჩამონათვალი, რომელიც ფრჩხილებშია მოთავსებული და მოსდევს მის სახელს ნული ან მეტი პარამეტრით. სიმარტივისთვის ჩვენ არ დავამატეთ პარამეტრები *Main*-ს. მოგვიანებით ამ თავში ნახავთ, რა სახის პარამეტრები შეიძლება ჰქონდეს *Main*-მეთოდს. უფრო მეტს ამ მეთოდის შესახებ შეისწავლით მე-5 თავში.

Main-მეთოდის ქცევას განსაზღვრავს ოპერატორი *Console.WriteLine(...)* . *Console* არის კლასი სახელსივრცეში *System.*, ხოლო *WriteLine(...)* არის მეთოდი *Console* კლასში. ჩვენ ვიყენებთ წერტილს "." პროგრამის ელემენტების სუბორდინაციისთვის. შევნიშნოთ, რომ ოპერატორი შეგვეძლო დაგვეწერა აგრეთვე, როგორც *System.Console.WriteLine(...)*. ეს შეესაბამება შაბლონს "*სახელსივრცე.კლასი.მეთოდი*", როგორც სრულად განსაზღვრული ოპერატორი. თუ გამოტოვეთ *using System*-ის აღწერა პროგრამის დასაწყისში, მაშინ აუცილებელი იქნება *System.Console.WriteLine(...)* სახელის სრული ფორმით გამოყენება. ამ ოპერატორის შესრულება იწვევს "მოგესალმებათ C შარპი!"-ის ბეჭდვას კონსოლის ეკრანზე.

შევნიშნოთ, რომ კომენტარები აღნიშნულია "//"-ით. ეს არის მხოლოდ ცალკეული სტრიქონის კომენტარისთვის და ძალაშია სტრიქონის ბოლომდე. თუ რამდენიმე სტრიქონის კომენტირება გსურთ, დაიწყეთ "/*"-ით და დაასრულეთ "*/"-ით. ყველაფერი, რაც ამ ორ სიმბოლოს შორისაა მოქცეული, არის კომენტარი. შეგიძლიათ ერთი სტრიქონის კომენტარი მოაქციოთ მრავალი სტრიქონის კომენტარშიც, თუმცა ვერ მოათავსებთ მრავალი სტრიქონის კომენტარს მრავალი სტრიქონის კომენტარში. კომპილატორი

იგნორირებას უკეთებს კომენტარებს. ისინი უბრალოდ აღწერენ ჩვეულებრივი სალაპარაკო ენით, თუ რას ემსახურება მოცემული პროგრამა.

ყოველი ოპერატორი მთავრდება წერტილ-მძიმით ";". წინადადებები და მეთოდები იწყება მარცხენა ფიგურული ფრჩხილით "{"-ით და მთავრდება მაჯვენა ფიგურული ფრჩხილით "}". ასეთ ფრჩხილებს შორის "{" და "}" მოთავსებულო ყოველი ოპერატორი განიხილება, როგორც ბლოკი. ბლოკი განსაზღვრავს პროგრამის ელემენტების ე. წ. ჭვრეტის არეს (არსებობის ხანგრძლივობა და ხილვადობა).

Main-მეთოდში თქვენ ასევე შენიშნეთ დამატებითი ოპერატორი *Console.WriteLine(...)*. არგუმენტების სია ამ ოპერატორში განსხვავებულია ადრინდელისგან. იგი შეიცავს ფორმატიზაციის სტრიქონს "*{0}*"-პარამეტრით. პირველი პარამეტრი ფორმატიზაციის სტრიქონში იწყება ციფრით 0, მეორე – ციფრით 1 და ასე შემდეგ. პარამეტრი "*{0}*" გულისხმობს ბოლო ბრჭყალის მომდევნო არგუმენტს, რომლის მნიშვნელობაც ამ პოზიციაში უნდა მოთავსდეს (ბეჭდვისას).

საკვანძო სიტყვა *double* გამოიყენება ორმაგი სიზუსტის მცურავწერტილიანი ცვლადების აღსაწერად, *Math* არის *System* სახელსივრცეში შემავალი კლასის სახელი, ხოლო *Sqrt* და *Pow* – ამ კლასში შემავალი სტატიკური მეთოდების სახელებია, შესაბამისად, კვადრატული ფესვისა და ხარისხის გამოსათვლელად.

ბრძანების სტრიქონიდან მონაცემთა მიღება

მრავალი პროგრამა იწერება ბრძანების სტრიქონიდან მონაცემთა მიღების შესაძლებლობით. ბრძანებათა სტრიქონი არის ადგილი, სადაც შესაძლებელია ბრძანებისა და მისი პარამეტრების შეყვანა. ოპერაციული სისტემა ავტომატურად იძახებს ბრძანების თანამოსახელე (იმავე სახელის მქონე) პროგრამას. თვით ბრძანება წარმოადგენს პროგრამას, რომელიც პარამეტრებს ბრძანებათა სტრიქონიდან ღებულობს. ბრძანების სტრიქონი ადრეული ოპერაციული სისტემებიდან მომდინარეობს. *Windows* შემთხვევაში Start=Run=><ბრძანების სახელი> <პარამეტრები> ან *Start=Run=>cmd=>Enter=>* <ბრძანების სახელი> <პარამეტრები>. ბრძანების სტრიქონიდან მონაცემები გადაეცემა შესაბამისი სახელის მქონე პროგრამის *Main*-მეთოდს. ამონაბეჭდში 1-2 ნაჩვენებია პროგრამა, რომელიც ღებულობს სიტყვას, როგორც ბრძანების პარამეტრს ბრძანებათა სტრიქონიდან და ბეჭდავს მას კონსოლზე (ჩვენს შემთხვევაში- დისპლეიზე სპეციალური ტიპის ფანჯარაში).

ამონაბეჭდი 1-2. ბრძანების სტრიქონიდან მონაცემთა მიღება: SaxeladMogesalmebaT.cs

```
// სახელსივრცის აღწერა
using System;
// პროგრამის საწყისი კლასი
class SaxeladMogesalmebaT
{// Main იწყებს პროგრამის შესრულებას.
   public static void Main(string[] argumentebi)
```

```
{  // კონსოლზე დაბეჭდვა // Write to console
   Console.WriteLine("გამარჯობათ, {0} , {1}!", argumentebi[0], argumentebi[1] );
   Console.WriteLine("მოგესალმებათ C შარპი!");
}}
```

ამონაბეჭდში 1-2 შეამჩნევთ *Main*-მეთოდში პარამეტრ-სიას. პარამეტრის სახელია *argumentebi*-ი. იგი შემდგომში შეგვიძლია გამოვიყენოთ ჩვენს პროგრამაში პარამეტრის მისათითებლად. გამოსახულება *string[]* განსაზღვრავს *argumentebi*-პარამეტრის ტიპს. ეს ტიპი შეიცავს სიმბოლოებს. ეს სიმბოლოები შეიძლება იყოს როგორც ცალკეული სიტყვები, ისე სიტყვათა ჯგუფები. კვადრატული ფრჩხილი "[]" აღნიშნავს მასივს, რომელიც სიის მსგავსია. ამიტომ *argumentebi*-პარამეტრის ტიპი არის სიტყვათა სია (სიტყვათა მიმდევრობა) ბრძანებათა სტრიქონში.

განვიხილოთ ბრჭყალის მომდევნო არგუმენტი. ეს არის არგუმენტი *argumentebi[0]*, რომელიც მიუთითებს პირველ ელემენტს (სტრიქონს) *argumentebi*-მასივში. მასივის პირველი ელემენტის ნომერია 0, შემდეგი ელემენტისა – 1 და ასე შემდეგ. მაგალითად, თუ ბრძანებათა სტრიქონში წერთ "SaxeladMogesalmebaT თენგიზ ხათუნა", მაშინ *argumentebi[0]*-ის მნიშვნელობა იქნება "თენგიზ", ხოლო *argumentebi[1]*- ის მნიშვნელობა - "ხათუნა".

დავუბრუნდეთ პარამეტრს "*{0}*" ფორმატიზებულ სტრიქონში. ვინაიდან *argumentebi[0]* არის ფორმატიზაციის სტრიქონის შემდეგი პირველი არგუმენტი *Console.WriteLine()* ოპერატორში, მისი მნიშვნელობა მოთავსდება ფორმატიზებული სტრიქონის პირველ პარამეტრში. ამ ბრძანების შესრულების შემდეგ *argumentebi[0]*-ის მნიშვნელობა, რომელიც არის "თენგიზ", ჩაანაცვლებს "*{0}*"-ს ფორმატიზებულ სტრიქონში. " SaxeladMogesalmebaT თენგიზ ხათუნა "-ბრძანების შესრულების შემდეგ დაიბეჭდება (გამოსასვლელზე) შემდეგი:

```
გამარჯობათ, თენგიზ, ხათუნა!
მოგესალმებათ C შარპი!
```

ფორმატიზაციის თითოეულ ელემენტს ზოგადად აქვს სახე:

{რიგითი ნომერი[,სიგრძე][:სპეციფიკატორი]}

კვადრატულ ფრჩხილებში მოთავსებული მდგენელები ფაკულტატურია და ჩვეულებრივ გამოიტოვება. ცხრილში 1-1. მოყვანილია სტანდარტული სპეციფიკატორები.

ცხრილი 1-1.

სპეციფიკატორი	აღწერა
"C", "c"	ფულადი
"D", "d"	ათობითი
"E", "e"	მაჩვენებლიანი
"F", "f"	ფიქსირებული წერტილით
"G", "g"	საერთო
"N", "n"	ციფრული
"P", "p"	პროცენტი
"X", "x"	თექვსმეტობითი

არასტანდარტული ციფრული სპეციფიკატორებისთვის გამოიყენება სიმბოლოები: "0","#",".","%", "E0" ,"E+0","E-0","e0","e+0","e-0", რომელთაგან 0 ნიშნავს ნიშნად და არა ნიშნად ნულს ან ციფრს, ხოლო # - მხოლოდ ნიშნადს, დანარჩენი სპეციფიკატორები გამოიყენება პროცენტისა და მაჩვენებლიანი რიცხვების წარმოსადგენად. თარიღის წარმოსადგენად გამოიყენება d,y და M მოდიფიკატორები. სტრიქონების შესწავლის დროს წარმოდგენილი იქნება სპეციფიკატორების რამდენიმე მაგალითი.

შენიშვნა. VISUAL STUDIO NET-ში ბრძანებათა სტრიქონის იმიტაციისთვის გამოიყენება შემდეგი საშუალება. Solution Explorer-ის ფანჯარაში მაუსის (თაგვის) მარჯვენა კლავიშს ვაჭერთ ჩვენი პროექტის სახელის თავზე. *Properties=>Configuration Properties=>Debuging*, შემდეგ კი ველში *Command line Arguments* ვწერთ სასურველ არგუმენტებს და Ok.

ურთიერთობა კონსოლის საშუალებით

სხვა გზა, რომელიც უზრუნველყოფს პროგრამაში მონაცემთა შეყვანას, არის ე. წ. კონსოლი. როგორც წესი, იგი მუშაობს შემდეგნაირად: თქვენ მომხმარებელს შეახსენებთ მონაცემთა შეყვანას, იგი დაბეჭდავს რაიმეს და დააჭერს ღილაკს *Enter*. პროგრამა გააანალიზებს შეყვანილ ინფორმაციას და შესაბამისად იმოქმედებს. ამონაბეჭდი 1-3 გვიჩვენებს, თუ როგორ შეიძლება განხორციელდეს ურთიერთობა მომხმარებელთან.

ამონაბეჭდი 1-3. კონსოლიდან მონაცემების შეყვანა: SekiTxvapasuxiMogesalmebaT.cs

```
// სახელსივრცის აღწერა
using System;
// პროგრამის საწყისი კლასი
class SekiTxvapasuxiMogesalmebaT
{// Main იწყებს პროგრამის შესრულებას..
   public static void Main()
   {   // კონსოლზე ბეჭდვა და მონაცემის შეყვანა
      Console.Write("რა გქვიათ თქვენ?: ");
      Console.Write("გამარჯობათ, {0}! ", Console.ReadLine());
      Console.WriteLine("მოგესალმებათ C შარპი!");  }}
```

თავი II

ცვლადები, ტიპები, კონსტანტები და ოპერაციები

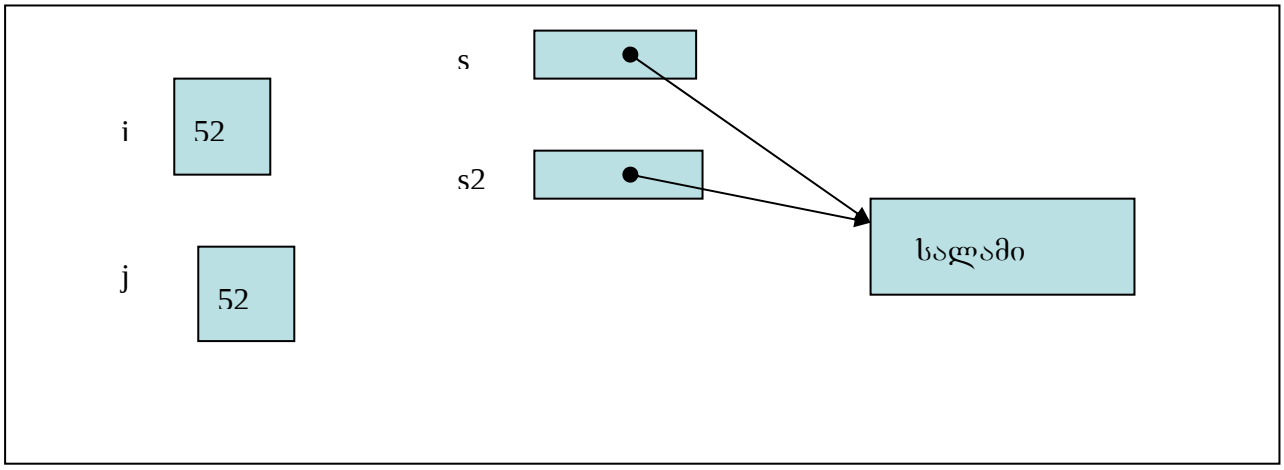
ამ თავის მიზანია შემდეგი საკითხების შესწავლა:

- ცვლადის არსი
- C#-ის თანმხლები ტიპები
- შესავალი C#-ის ოპერატორებში
- მასივების გამოყენება

ცვლადები და ტიპები

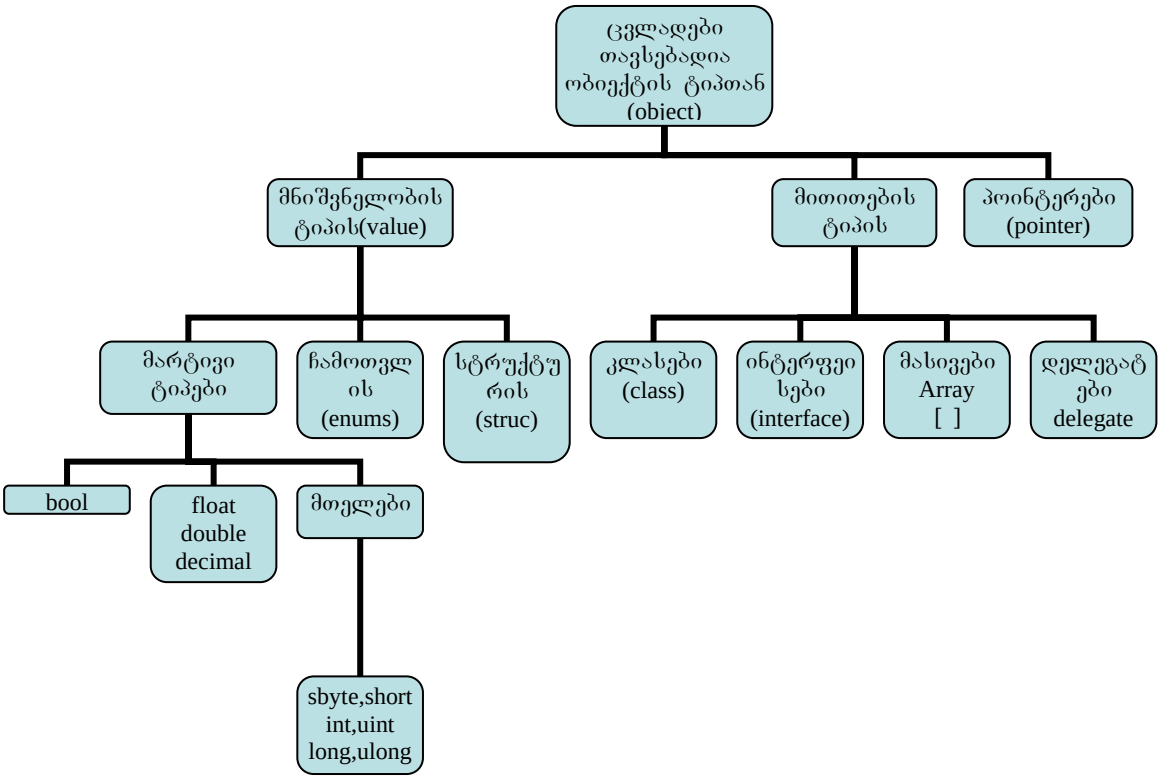
"ცვლადები" არის უბრალოდ მონაცემთა შესანახი ადგილი. თქვენ შეგიძლიათ მათში მოათავსოთ მონაცემები და გამოიყენოთ მათი მნიშვნელობები, როგორც C# -ის გამოსახულების ნაწილი. მონაცემთა ინტერპრეტაცია ცვლადებში იმართება ტიპების საშუალებით. ტრადიციულად ცვლადის სახელი ასოციირდება კომპიუტერის მეხსიერების მისამართთან. ამ მისამართზე მოთავსებულია მონაცემები. ეს მონაცემები წარმოადგენს ცვლადის რეალურ მნიშვნელობას. ცვლადის ტიპის მიხედვით ეს მნიშვნელობა შეიძლება ინტერპრეტირებულ იქნეს, როგორც რაიმე კონკრეტული რიცხვ(ებ)ი ან სიმბოლო(ები) ან როგორც რაიმე მისამართი (ანუ მითითება სხვა ცვლადზე) და ამ მისამართზე მოთავსებული მნიშვნელობა (ირიბი დამისამართება). მაგალითად, ოპერატორების მიმდევრობა: *int i=52;int j=i; string s="სალამი"; string s2=s;* შეიძლება თვალსაჩინოებისთვის შემდეგნაირად წარმოვადგინოთ ნახ.-ზე 2.1. მეხსირება თვალსაჩინოებისთვის წარმოდგენილია მართკუთხედებით, მითითებები – ისრით. i და j ცვლადების შემთხვევაში მათ მნიშვნელობად იღებენ მათ შესაბამის მისამართზე მოთავსებულ მნიშვნელობებს. s და s2 ცვლადის შემთხვევაში მინიჭებისას გამოიყენება ამ ცლადების მისამართები, ხოლო გამოსახულებებში – ამ მისამართზე მოთავსებული მნიშვნელობანი ანუ ცვლადის მნიშვნელობის ინტერპრეტაცია მისი გამოყენების კონტექსტზეა დამოკიდებული. s და s2 ცვლადის მნიშვნელობა ერთი და იგივეა ანუ ისინი მეხსიერების ერთსა და იმავე ნაწილს მიუთითებენ. C# არის მკაცრად "ტიპიზირებული" ენა.

ნახ 2.1.



აქ ყველა ოპერაცია ცვლადებზე სრულდება იმის მიხედვით, თუ რა ტიპისაა ცვლადი. არსებობს წესები, რომლებიც განსაზღვრავენ, რომელი ოპერაციაა დაშვებული იმისთვის, რომ არ დაირღვეს ცვლადში მოთავსებულ მონაცემთა ერთიანობა. ცვლადების კლასიფიკაცია ტიპების მიხედვით წარმოდგენილია ნახ. 2.2.

ნახ. 2.2.



C#-ის მარტივი ტიპები შედგება ლოგიკური (ბულის), სტრიქონული და სამი რიცხვითი ტიპისგან – მთელი, მცურავწერტილიანი და ათობითი. ტერმინი "*Integrals*" გულისხმობს შემდეგ ტიპებს: *sbyte, byte, short, ushort, int, uint, long, ulong, და char*. უფრო დაწვრილებით ამის შესახებ საუბარი გვექნება მოგვიანებით ამავე თავში. ტერმინი *"Floating Point"* მიუთითებს მცურავ და ორმაგ ტიპზე, რომელსაც დეტალურად განვიხილავთ ათობით ტიპთან ამავე თავში. სტრიქონის ტიპი წარმოადგენს სიმბოლოთა სტრიქონს და განიხილება ასევე მოგვიანებით ამავე თავში.

bool (ლოგიკური) ტიპი

ბულის (ლოგიკური) ტიპის აღსაწერად გამოიყენება საკვანძო სიტყვა *bool*. ამ ტიპის ცვლადებს აქვთ ორთაგან ერთი მნიშვნელობა: *true* ან *false*. ზოგიერთ სხვა ენაში იგივე დატვირთვა აქვს 0 და 1-ს. C# -ში მხოლოდ *true* და *false* აკმაყოფილებს ბულის ტიპის მოთხოვნებს. ეს არის ენის თანდაყოლილი საკვანძო სიტყვები. ამონაბეჭდი 2-1 წარმოგვიდგენს ბულის ტიპის ცვლადის გამოყენების მაგალითს პროგრამაში.

ამონაბეჭდი 2-1. ბულის მნიშვნელობების წარმოდგენა: Boolean.cs

```
using System;
class bulis
{   public static void Main()
    {       bool WeSmariti = true;
        bool mcdari = false;
        Console.WriteLine("ჭეშმარიტი = {0}  მცდარი  ={1}", WeSmariti, mcdari);
    }
}
```

ამონაბეჭდში 2-1 ბულის მნიშვნელობები ამონათდება კონსოლზე (ჩვენ შემთხვევაში ფანჯარა, სადაც შედეგები შეიყვანება კლავიატურიდან, ხოლო გამოიყვანება ეკრანზე იმავე ფანჯარაში), როგორც წინადადების ნაწილი. ბული ტიპისთვის დასაშვებია მხოლოდ *true* ან *false*, როგორც ნაჩვენებია მინიჭებისას *true* WeSmariti-თვის და *false* mcdari -თვის. ამ პროგრამის გაშვებისას მივიღებთ შემდეგ შედეგს:

```
ჭეშმარიტი = True   მცდარი   =False
```

int (მთელი) ტიპი

C#-ში მთელი ტიპი არის ტიპების კატეგორია. ესაა ნიშნიანი ან უნიშნო მთელი რიცხვები და სიმბოლური ტიპი. სიმბოლური (char) ტიპი არის უნიკოდ-სიმბოლო (ორი ბაიტი გამოიყენება კოდირებისთვის). ცხრილი 2-1 გვიჩვენებს მთელ ტიპებს, მათ ზომებსა და დიაპაზონს.

ცხრილი 2-1. მთელი ტიპის ზომები და დიაპაზონი C#

ტიპი	ზომა (ბიტებში)	დიაპაზონი
sbyte	8	-128 to 127
byte	8	0 to 255
short	16	-32768 to 32767
ushort	16	0 to 65535
int	32	-2147483648 to 2147483647
uint	32	0 to 4294967295
long	64	-9223372036854775808 to 9223372036854775807
ulong	64	0 to 18446744073709551615
char	16	0 to 65535

მთელი ტიპები კარგად მიესადაგება იმ ოპერაციებს, რომლებიც მოიცავს ოპერაციებს მთელ რიცხვებზე. char ტიპი წარმოადგენს ერთ უნიკოდ-სიმბოლოს. როგორც ცხრილიდან ჩანს, თქვენ გაქვთ ფართო არჩევანი მოთხოვნილებების შესაბამისად.

მცურავწერტილიანი და ათობითი ტიპი

C#-ში მცურავწერტილიანი (floating point) ტიპი არის ***float* ან *double*.** ის გამოიყენება ნამდვილი რიცხვების წარმოსადგენად. ათობითი ტიპი (4 ბიტი ერთი ათობითი ციფრის წარმოსადგენად) გამოიყენება, როდესაც წარმოვადგენთ ფინანსურ ან ფულად მონაცემებს. ცხრილი 2-2 გვიჩვენებს მცურავწერტილიან და ათობით ტიპს, მათ ზომას, სიზუსტესა და დიაპაზონს.

ცხრილი 2-3. მცურავწერტილიანი და ათობითი ტიპი, ზომა, სიზუსტე და დიაპაზონი

ტიპი	ზომა (ბიტებში)	სიზუსტე	დიაპაზონი
float	32	7 ციფრი	$\pm 1.5 \times 10^{-45}$ to 3.4×10^{38}
double	64	15-16 ციფრი	$\pm 5.0 \times 10^{-324}$ to 1.7×10^{308}
decimal	128	28-29 ათობითი თანრიგი	$\pm 1.0 \times 10^{-28}$ to 7.9×10^{28}

მცურავწერტილიანი ტიპი გამოიყენება, როდესაც ვაწარმოებთ ოპერაციებს ათწილადებზე, კერძოდ, კარგია სამეცნიერო გათვლებისთვის. რა თქმა უნდა, ათობითი (*decimal*) ტიპი საუკეთესო არჩევანია ფინანსური გათვლებისთვის, ვინაიდან თავიდან ვიცილებთ დამრგვალებით გამოწვეულ უზუსტობებს.

სიმბოლოს ტიპი

სტრიქონის აღწერისას ორმაგი ბრჭყალები გამოიყენება, ხოლო *Char* ტიპის სიმბოლოებისთვის - ერთმაგი. მაგალითად:

Char c='A'; ეს ტიპი გამოიყენება ნებისმიერი, მათ შორის *unicode* სიმბოლოების აღსაწერად. იგი იკავებს ერთ ან ორ ბაიტს.

ამონაბეჭდში 2-1.1 მოყვანილია ამ ტიპის ცვლადთან მუშაობის მაგალითები.

ამონაბეჭდი 2-1.1.

```
using System;
namespace nschar
{       class Class1
        {               static void Main(string[] args)
                {               char chA = 'A';
```

```
                    char ch1 = '1';
                    string str = "test string";
char MyChar = 'X';  Console.WriteLine(MyChar);//გამოსავალი:X
MyChar = '\x0058';  Console.WriteLine(MyChar);// გამოსავალი:X
MyChar = (char)88;  Console.WriteLine(MyChar);// გამოსავალი:X
MyChar = '\u0058';  Console.WriteLine(MyChar);  // გამოსავალი:X
Console.WriteLine(chA.CompareTo('B'));      // გამოსავალი: "-1"
Console.WriteLine(chA.Equals('A'));         // გამოსავალი: "True"
Console.WriteLine(Char.GetNumericValue(ch1));   // გამოსავალი: "1"
Console.WriteLine(Char.IsControl('\t'));    // გამოსავალი: "True"
Console.WriteLine(Char.IsDigit(ch1));       // გამოსავალი: "True"
Console.WriteLine(Char.IsLetter(','));      // გამოსავალი: "False"
Console.WriteLine(Char.IsLower('u'));       // გამოსავალი: "True"
Console.WriteLine(Char.IsNumber(ch1));      // გამოსავალი: "True"
Console.WriteLine(Char.IsPunctuation('.'));     // გამოსავალი: "True"
Console.WriteLine(Char.IsSeparator(str, 4));   // გამოსავალი: "True"
Console.WriteLine(Char.IsSymbol('+'));      // გამოსავალი: "True"
Console.WriteLine(Char.IsWhiteSpace(str, 4));   // გამოსავალი: "True"
Console.WriteLine(Char.Parse("S"));         // გამოსავალი: "S"
Console.WriteLine(Char.ToLower('M'));       // გამოსავალი: "m"
Console.WriteLine('x'.ToString());          // გამოსავალი: "x"
                    }}}
```

შედეგებიდან და შესაბამისი მეთოდების ინგლისური სიტყვების მნიშვნელობებიდან გამომდინარე ადვილად გასაგებია ამ პროგრამის მსვლელობა. მაგალითად: *Char.IsDigit(ch1)* გამოსახულებაში *Char* კლასის სახელია, *IsDigit* სტატიკური მეთოდის სახელია, *ch1* მეთოდის არგუმენტია ცვლადის სახით და ნიშნავს: არის ***ch1*** ცვლადის მნიშვნელობა ციფრი?

სტრიქონის ტიპი

string ტიპი არის ტექსტური სიმბოლოების სტრიქონი. სტრიქონის შექმნის სტანდარტული საშუალებაა ბრჭყალებში მოქცეული ლიტერების (სიმბოლოების) გამოყენება. "ეს არის სტრიქონის მაგალითი". თქვენ ხედავდით სტრიქონებს, დაწყებული პირველი გაკვეთილიდან, სადაც ვიყენებდით *Console.WriteLine* მეთოდს. კონსოლზე (ჩვენ შემთხვევაში - დისპლეიზე გარკვეული სახის ფანჯარა მონაცემების გამოსაყვანად და შესაყვანად) მონაცემების გასაგზავნად. ზოგიერთი სიმბოლო არაბეჭდვადია, მაგრამ ზოგჯერ წარმოიშობა მათი გამოყენების საჭიროება სტრიქონებში. ამისათვის C# აქვს

სპეციალური სინტაქსი არაბეჭდვადი სიმბოლოების წარმოსადგენად. მაგალითად, ჩვეულებრივ გამოიყენება ახალი სტრიქონის დაწყების ნიშანი ტექსტში, რომელიც შეესაბამება სიმბოლოს '\n'. უკუმიმართულების დახრილი ხაზი '\' წარმოადგენს სპეც. ნიშანს, ე. წ. escape-ს (გამსხლტომი). როდესაც ტექსტში გვხვდება escape სიმბოლო, 'n' აღარ აღიქმება, როგორც ანბანის ასო, არამედ იგი აღნიშნავს ახალ სტრიქონს (newline). ტექსტში უკუმიმართულების დახრილი ხაზის '\' წარმოსადგენად საჭიროა მისი ჩაწერა ორჯერ ზედიზედ ('\\'). ცხრილში 2-4 ნაჩვენებია ფართოდ გავრცელებული escape-მიმდევრობები.

ცხრილი 2-4. C# სიმბოლოთა Escape –მიმდევრობა და მნიშვნელობა

მმართველი მიმდევრობა	მნიშვნელობა
\'	Single Quote (ერთმაგი ბრჭყალი)
\"	Double Quote (ორმაგი ბრჭყალი)
\\	Backslash (უკუდახრილი)
\0	Null (ნული, ანუ რვა ნულოვანი ბიტი"00000000", გამოიყენება საბეჭდი მანქანის მართვისას)
\a	Bell (ზარი)
\b	Backspace (უკუწაშლა)
\f	Form Feed (ფორმის, ფურცლის მიწოდება)
\n	Newline (ახალი სტრიქონი)
\r	Carriage Return (ეტლის დაბრუნაბა)
\t	Horizontal Tab (ჰორიზონტალური ტაბულაცია)
\v	Vertical Tab (ვერტიკალური ტაბულაცია)

C#-ის სტრიქონები შეიძლება წარმოვადგინოთ ე. წ. სიტყვა-სიტყვითი ლიტერით, რომელიც წარმოადგენს სტრიქონს @-სიმბოლოს პრეფიქსით, როგორიცაა *@"რაიმე სტრიქონი"*. წარმოდგენის ამ მეთოდს ის თავისებურება აქვს, რომ escape სპეც. სიმბოლოებს აღიქვამს, როგორც ჩვეულებრივს უკეთესი კითხვადობისთვის. მაგალითად, ფაილის გზის განმსაზღვრელი წინადადება, როგორიც არის *"c:\\topdir\\subdir\\subdir\\myapp.exe"*. როგორც ხედავთ, უკუმიმართულებით დახრილი ხაზი არის escape (გამსხლტომი) სიმბოლო, რომელიც ნაკლებად კითხვადს ხდის სტრიქონს. სტრიქონის კითხვადობა შეგვიძლია გავაუმჯობესოთ სიტყვა-სიტყვითი ლიტერის გამოყენებით მსგავსად ამისა: *@"c:\topdir\subdir\subdir\myapp.exe"*.

ბრჭყალების წარმოსადგენად ამ შემთხვევაში გამოიყენება ორმაგი ბრჭყალები, მაგალითად, სტრიქონი *"copy \"c:\\საწყისი–ფაილი.txt\" c:\\ახალი–ფაილი.txt"* სიტყვა-სიტყვითი

ლიტერის საშუალებით შეიძლება ჩაიწეროს შემდეგნაირად: *@"copy ""c:\საწყისი–ფაილი.txt"" c:\ახალი–ფაილი e.txt".*

C# ენაში სტრიქონი განიხილება, როგორც *Char* ტიპის მასივი. სტრიქონის ინდივიდუალურ სიმბოლოებს შესაძლებელია მივმართოთ ინდექსით. როგორც ნაჩვენებია ქვემოთ:

```
string striqoni = " უკუმიმდევრობით ბეჭდვა";
System.Console.Write("{0}{1}",striqoni[0], striqoni[1]); // გამოსავალი: "უკ"
```

ზოგიერთი მაგალითი შემდგომი მასალის ათვისებასთან ერთად გახდება უფრო ნათელი.

სტრიქონში ცალკეული სიმბოლო შეიძლება ამორჩეულ იქნეს ინდექსის საშუალებით:

```
string striqoni = " უკუმიმდევრობით ბეჭდვა ";
for (int i = 0; i < striqoni.Length; i++) { System.Console.Write(striqoni[striqoni.Length - i - 1]);} //
გამოსავალი: "ავდჭებ თიბელუთრამიმუკუ"
```

ამ მაგალითში სტრიქონი უკუმიმდევრობით იბეჭდება. `Length`-სტრიქონის კლასის თვისებაა და მისი კონკრეტული ეგზემპლარის სიგრძეს ასახავს. ციკლი მეორდება i-ს მნიშვნელობებისთვის 0-დან სტრიქონის სიგრძემდე ბიჯით 1. ენის სპეციფიკაციის თანახმად არ გვაქვს უფლება შევცვალოთ სტრიქონის ცალკეული სიმბოლოები *(immutable-თვისება).* ასეთ შემთხვევაში, თუ საჭიროა, უბრალოდ ახალი სტრიქონი უნდა დავაფორმიროთ.

სტრიქონის ტიპის ცვლადებზე უამრავი ოპერაციაა შესაძლებელი, რომლებსაც *string* კლასი მეთოდების სახით მოიცავს. აქ მხოლოდ ზოგიერთი მაგალითით შემოვიფარგლებით. სტრიქონის ცალკეულ ნაწილებთან მუშაობა შესაძლებელია შემდეგი მეთოდების გამოყენებით: *SubString()-ქვესტრიქონი), Replace()-ჩაანაცვლე, Split()-გახლიჩე* და *Trim()-ჩამოაჭერი.*

```
string s1 = "ფორთოხალი"; string s2 = "წითელი"; s1 += s2; System.Console.WriteLine(s1);
// გამოსავალი: " ფორთოხალი წითელი "
s1 = s1.Substring(2, 5); System.Console.WriteLine(s1); // გამოსავალი: "რთოხა"
```

ამ მაგალითში ხდება ორი სრიქონის შეეერთება *(კონკატენაცია, ანუ ერთი მიეწერება მეორეს),* იქმნება ახალი სტრიქონი, ხოლო შემდეგ ქვესტრიქონი *Substring(2, 5)* მეთოდით გამოიყოფა. შედეგად ვღებულობთ მეორე პოზიციიდან ხუთ სიმბოლოს *(პოზიციები 0-დან ინომრება).*

სტრიქონების ობიექტები შეუცვლელია, რაც იმას ნიშნავს, რომ შექმნის შემდეგ მათი შეცვლა არ შეიძლება. მეთოდები, რომლებიც მოქმედებენ სტრიქონებზე სინამდვილეში, ქმნიან ახალ ობიექტებს. წინა მაგალითში, როცა ხდებოდა s1 და s2 სტრიქონების ერთ სტრიქონად გაერთიანება, ორივეს მნიშვნელობები ”ფორთოხალი” და ”წითელი” შეუცვლელი დარჩა. ოპერატორი += ქმნის ახალ სტრიქონს, რომელიც შეიცავს

შეერთების შედეგად მიღებულ მნიშვნელობას, რის შედეგადაც s1 მიუთითებს სრულიად განსხვავებულ სტრიქონზე. სტრიქონი, რომელიც შეიცავს "ფორთოხალი" კვლავ არსებობს, მაგრამ მასზე მიმართვა აღარ ხდება s1-ის კონკატენაციის შემდეგ.

თუ გვსურს სტრიქონში ტექსტის ერთი ნაწილის შეცვლა:

string s3="ბეისიკი კარგი ენაა";

System.Console.WriteLine(s3.Replace("ბეისიკი კარგი", "C შარპი უკეთესი"));

//ეკრანზე გამოდის "C შარპი უკეთესი ენაა"

თუ გვსურს სტრიქონის დასაწყისში და ბოლოში ჩამოვაჭრათ შუალედები, მაშინ ვიყენებთ *Trim* მეთოდს:

s=" .a ... d, b . g,,,,, c . j, k. ";
Console.WriteLine(s.Trim());//ეკრანზე გამოდის: ".a ... d, b . g,,,,, c . j, k."

tostring()

ვინაიდან C# ყველა ობიექტი წარმოქმნილია *Object*-კლასიდან რიცხვების მიმართ შეიძლება გამოვიყენოთ *ToString*_მეთოდი, რომელიც რიცხვით მნიშვნელობას გარდაქმნის სტრიქონად. ამ მეთოდის გამოყენებით გარდავქმნათ რაიმე რიცხვითი მნიშვნელობა სტრიქონად:

int weli=1983; string Setyobineba="ია დაიბადა " + weli.ToString();

System.Console.WriteLine(Setyobineba); // შედეგად მივიღებთ "ია დაიბადა 1983"

შესაძლებელია ასევე სტრიქონის შემცველობის კოპირება სიმბოლოების მასივში და პირიქით:

string striqoni="გაუმარჯოს საქართველოს!";

char[] simboloebi=striqoni.ToCharArray(0, striqoni.Length);

// სტრიქონის გარდაქმნა Char ტიპის მასივად

foreach (char c in simboloebi)

System.Console.Write(c); //ეკრანზე გამოდის "გაუმარჯოს საქართველოს!"

// Char(სიმბოლოების, ლიტერების) ტიპის მასივის გარდაქმნა სტრიქონად

striqoni=new string(simboloebi);

System.Console.WriteLine("სტრიქონი ={0}",striqoni);

System.Console.WriteLine(striqoni); //ეკრანზე გამოდის "გაუმარჯოს საქართველოს!"

შესაძლებელია აგრეთვე სიმბოლოს დუბლირებით სტრიქონის ფორმირება:

Console.WriteLine(new string('X',5)+'='+new string('Y',5));// "XXXXX=YYYYY"

ზედა მაგალითში ადგილი აქვს ორი სტრიქონის და ერთი სიმბოლოს კონკატენაციას.

რეგისტრის შეცვლა

სტრიქონში ასოების რეგისტრის შესაცვლელად გამოიყენება მეთოდები, *ToUpper()* ან *ToLower(),* შემდეგნაირად:

string striqoni=”ნინო დაიბადა, 1982”;

System.Console.WriteLine(striqoni.ToUpper()); //ეკრანზე გამოდის “ნინო დაიბადა, 1982”

System.Console.WriteLine(striqoni.ToLower()); //ეკრანზე გამოდის “ნინო დაიბადა, 1982”

შედარებები

ორი სტრიქონის შედარების ყველაზე მარტივი გზაა "==" და "!=" ოპერაციების გამოყენება, რომლებიც ასრულებენ რეგისტრზე დამოკიდებულ შედარებებს.

string feri1=”წითელი”; string feri2=”მწვანე”; string feri3=”წითელი”;

if (feri1==feri3) { System.Console.WriteLine(“ტოლია”); } if (feri1 !=feri2)

{ System.Console.WriteLine(“არ არის ტოლი”); }

სტრიქონის ობიექტებს ასევე გააჩნიათ *CompareTo()* მეთოდი, რომელიც შედეგში აბრუნებს მთელ რიცხვს დაფუძნებულს იმაზე, ერთ-ერთი სტრიქონი პატარაა, ვიდრე (<) ან დიდია, ვიდრე (>) მეორე. როცა ხდება სტრიქონების შედარება, გამოიყენება ასოების ციფრული კოდი (Unicode) და ქვედა რეგისტრს გააჩნია ნაკლები მნიშვნელობა, ვიდრე მაღალ რეგისტრს.

string striqoni1=”ABC”; string striqoni2=”abc”;

if (striqoni1.CompareTo(striqoni2)>0) {

System.Console.WriteLine(“მეტია ვიდრე”); } else {

System.Console.WriteLine(“ნაკლებია ვიდრე”); }

სტრიქონში ქვესტრიქონის მოსაძებნად შეიძლება გამოვიყენოთ *IndexOf(). IndexOf()* აბრუნებს -1-ს, თუ საძებნი ქვესტრიქონი არ იყო აღმოჩენილი ან აბრუნებს სტრიქონის პირველი ასოს პოზიციას, რომლის ათვლაც ხდება ნულიდან.

string striqoni=”თეა დაიბადა, 1987”;

System.Console.WriteLine(striqoni.IndexOf(“დაიბადა”)); //ეკრანზე გამოდის 4

System.Console.WriteLine(striqoni.IndexOf(“1985”)); // ეკრანზე გამოდის -1

სტრიქონის დაყოფა ქვესტრიქონებად

სტრიქონის დაყოფა ქვესტრიქონებად, მაგალითად, წინადადების დაყოფა ცალკეულ სიტყვებად, პროგრამირებისთვის ჩვეულებრივი ამოცანაა. მეთოდი *Split()* იღებს განმაცალკავებელი სიმბოლოების *(char)* მასივს, მაგალითად, გამოტოვების სიმბოლო, და აბრუნებს ქვესტრიქონების მასივს. ჩვენ შეგვიძლია შევაღწიოთ ამ მასივში *foreach*-ით შემდეგნაირად:

char[] ganmacalkavebeli=new char[] { ' ' }; string striqoni=”C შარპი დახვეწილი ენაა ”;

foreach (string qvestriqoni in striqoni.Split(ganmacalkavebeli)) {

System.Console.WriteLine(qvestriqoni); }

ამ კოდის შესრულების შედეგად ყველა სიტყვა ცალ-ცალკე დაიბეჭდება თითო სტრიქონზე:

C

შარპი

დახვეწილი

ენაა

სტრიქონის ფორმატიზაციისთვის შესაძლებელია შემდეგი კოდის გამოყენება:

```
string Cemisaxeli = "თენგიზი";
String.Format("სახელი = {0}, საათი = {1:hh}", Cemisaxeli, DateTime.Now);
string FormatString1 = String.Format("{0:dddd MMMM}", DateTime.Now);
string FormatString2 = DateTime.Now.ToString("dddd MMMM");
int MTeli = 100;
Console.WriteLine("{0:C}", MTeli);
string Cemisaxeli = "ხათუნა";
String.Format("სახელი = {0}, საათი = {1:hh}, წუთი = {1:mm}",
      Cemisaxeli, DateTime.Now);
string FormatPrice = String.Format("ფასი = {0,10:C}", mTeli);
```

ზემომოყვანილ მაგალითებში *Format* და *ToString* შესაბამისი კლასების მეთოდებია, ხოლო ***Now*** არის ***DateTime*** კლასის თვისება, რომლის მნიშვნელობაა მიმდინარე დრო.

კონსტანტები

კონსტანტები კლასის წევრებია, რომლებიც წარმოადგენს მუდმივ მნიშვნელობებს, ერთხელ გამოითვლება კომპილაციის დროს და არ იცვლება პროგრამის მსვლელობისას. მათ აღსაწერად გამოიყენება საკვანძო სიტყვა *const*. ქვემოთ მოყვანილია კონსტანტების აღწერის მაგალითები:

```
class A
{     public const double neper= 2.71828182845905, pi = 3.14159265358979, Z = 3.0;
}
```

ეკვივალენტურია:

```
class A
{   public const double neper= 2.71828182845905;
  public const double pi = 3.14159265358979;
  public const double Z = 3.0;
}
```

დასაშვებია სხვა კლასების კონსტანტების გამოყენება ახალი კონსტანტების განსასაზღვრად:

```
class A
{   public const int X = B.Z + 1;
  public const int Y = 10;
}
class B
```

```
{   public const int Z = A.Y + 1;
}
```

C#- ოპერაციები

შედეგები გამოითვლება გამოსახულებათა აგების საშუალებით. ეს გამოსახულებები აიგება ოპერატორებში ცვლადებისა და ოპერაციის ნიშნების ერთობლივი გამოყენებით. შემდეგ ცხრილში აღწერილია დასაშვები ოპერაციის ნიშნები (operators), მათი შესრულების თანამიმდევრობა და ასოციაციურობა.

ცხრილი 2-5. ოპერაციები რიგითობის და ასოციაციურობის მიხედვით

ოპერაციები კატეგორიების და რიგითობის მიხედვით	ოპერაციები	ასოციაციურობა გამოთვლის მიმართულებით
Primary (ძირითადი)	(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked	მარცხენა
Unary (უნარული)	+ - ! ~ ++x --x (T)x	მარცხენა
Multiplicative(მრავლობითი)	* / %	მარცხენა
Additive (ადიტიური)	+ -	მარცხენა
Shift (გადასვლის)	<< >>	მარცხენა
Relational (რელაციული)	< > <= >= is	მარცხენა
Equality (ტოლობის)	== !=	მარცხენა
Logical AND (ლოგიკური) "და")	&	მარცხენა
Logical XOR (გამომრიცხავი "ან")	^	მარცხენა
Logical OR (ლოგიკური "ან")	|	მარცხენა
Conditional AND (პირობის "და")	&&	მარცხენა
Conditional OR (პირობის "ან")	||	მარცხენა
Conditional > (პირობითი)	?:	მარჯვენა
Assignment (მინიჭების)	= *= /= %= += -= <<= >>= &= ^= |=	მარჯვენა

მარცხენა ასოციაციურობა ნიშნავს, რომ ოპერაციები სრულდება მარცხნიდან მარჯვნივ, მარჯვენა ასოციაციურობა კი პირიქით – მარჯვნიდან მარცხნივ, როგორც მინიჭების ოპერატორში, სადაც ყველაფერი წინასწარ გამოითვლება მარჯვნივ, შედეგი კი თავსდება ცვლადში მარცხნივ.

ოპერაციათა უმრავლესობა უნარული ან ბინარულია. უნარულ ოპერაციაში მონაწილეობს მხოლოდ ერთი ცვლადი, ხოლო ბინარულში – ორი ცვლადი. ამონაბეჭდი 2-2. წარმოგვიდგენს, თუ როგორ გამოიყენება უნარული ოპერაციები.

ამონაბეჭდი 2-2. უნარული ოპერაციები: Unaruli.cs

```
using System;
class Unaruli
{   public static void Main()
  {  int unaruli = 0;
     int winaswarizrdiT;
     int winaswariklebiT;
     int SemdgomizrdiT;
     int SemdgomiklebiT;
     int dadebiTi;
     int uaryofiTi;
     sbyte bituriara;
     bool logikuriara;
     winaswarizrdiT = ++unaruli;
     Console.WriteLine("წინასწარი ზრდით: {0}", winaswarizrdiT);
     winaswariklebiT = --unaruli;
     Console.WriteLine("წინასწარი კლებით: {0}", winaswariklebiT);
     SemdgomiklebiT = unaruli--;
     Console.WriteLine("შემდგომი კლებით: {0}", SemdgomiklebiT);
     SemdgomizrdiT = unaruli++;
     Console.WriteLine("შემდგომი ზრდით: {0}", SemdgomizrdiT);
     Console.WriteLine("უნარულის საბოლოო მნიშვნელობა: {0}", unaruli);
     dadebiTi = -SemdgomizrdiT;
     Console.WriteLine("დადებითი: {0}", dadebiTi);
     uaryofiTi = +SemdgomizrdiT;
     Console.WriteLine("უარყოფითი: {0}", uaryofiTi);
     bituriara = 0;
     bituriara = (sbyte)(~bituriara);
     Console.WriteLine("ბიტური არა: {0}", bituriara);
     logikuriara = false;
     logikuriara = !logikuriara;
     Console.WriteLine("ლოგიკური არა: {0}", logikuriara);
```

```
    }
}
```

გამოსახულების გამოთვლისას შემდგომი ზრდით (post-increment (x++)) და შემდგომი კლებით (post-decrement (x--)) ოპერაციები უკან აბრუნებენ ცვლადების მიმდინარე მნიშვნელობებს და შემდეგ ხდება მათი შესაბამისი ცვლილება. რა თქმა უნდა, როდესაც ვიყენებთ წინასწარი ზრდისა (pre-increment (++x)) და წინასწარი კლების (pre-decrement (--x)) ოპერაციებს, ჯერ ხდება ოპერაციის გამოყენება ცვლადზე, ხოლო შემდეგ მისი საბოლოო მნიშვნელობის უკან დაბრუნება.

ამონაბეჭდში 2-2. უნარული ცვლადის საწყისი მნიშვნელობა არის 0 (zero). როცა გამოიყენება pre-increment (++x) ოპერატორი, ცვლადი *unaruli* იზრდება 1-ით და მნიშვნელობა 1 ენიჭება *winaswarizrdiT* ცვლადს. pre-decrement (--x) ოპერაცია აბრუნებს უნარის მნიშვნელობას უკან 0-ზე და შემდეგ ანიჭებს ამ მნიშვნელობას *winaswariklebiT* ცვლადს.

როცა გამოიყენება post-decrement (x--) ოპერაცია, *unaruli*-ს მნიშვნელობა 0 თავსდება *SemdgomiklebiT* ცვლადში და შემდეგ *unaruli* მცირდება -1-ით. შემდეგი post-increment (x++) ოპერაცია *unaruli*-ს მიმდინარე მნიშვნელობას -1-ს მიანიჭებს *SemdgomizrdiT* ცვლადს და ამის შემდეგ ზრდის *unaruli*-ს 0-მდე.

ცვლადის *bituriara* საწყისი მნიშვნელობა არის 0 (zero) და მის მიმართ გამოიყენება შებრუნების ოპერაცია (~). ბიტური უარყოფის (შებრუნების) ოპერაცია აბრუნებს ბიტებს ცვლადში. ამ შემთხვევაში 0-ის ორობითი წარმოდგენა "00000000" გარდაიქმნება -1-ად "11111111".

აღსანიშნავია, რომ გამოსახულება *(sbyte)(~bituriara)* სრულდება ნებისმიერ ტიპებზე, როგორიცაა *sbyte, byte, short* ან *ushort* და უკან აბრუნებს, როგორც *int* მნიშვნელობას. შედეგის *bituriara* ცვლადისთვის მისანიჭებლად გამოვიყენეთ ტიპის გარდაქმნის ცხადი ოპერატორი (Type), სადაც Type არის კონკრეტული ტიპი, რომელშიც გსურთ გარდაქმნა, ამ შემთხვევაში - *sbyte*). ტიპის ცხადი გარდაქმნის ოპერატორი ნაჩვენებია, როგორც უნარული ოპერატორი "(T)x" ცხრილში 2-4. ტიპის ცხადი გარდაქმნის ოპერატორი უნდა შესრულდეს აშკარად, როდესაც უფრო ფართო ტიპიდან გადავდივართ მომცრო ტიპზე, ვინაიდან არსებობს მონაცემთა დაკარგვის შესაძლებლობა. ზოგადად შეიძლება ვთქვათ, რომ მცირე ტიპის მინიჭებისას ფართო ტიპისთვის პრობლემებს არ ვხვდებით, ვინაიდან ფართო ტიპს აქვს საკმარისი მეხსიერება მცირე ტიპის მთლიანი მნიშვნელობის შესანახად. გარდა ამისა, ყურადღება გვმართებს ნიშნიანი და უნიშნო ტიპების გარდაქმნისას. დარწმუნებული უნდა იყოთ, რომ მონაცემთა მთლიანობა არ დარღვეულა.

ლოგიკური უარყოფის ოპერატორი (!) საშუალებას იძლევა, გადართოს ბულის ცვლადის მნიშვნელობა. მაგალითში *logikuriara* ცვლადი იცვლება *false*-დან *true*-ზე. შეიძლება მოელოდეთ ზემომოყვანილი პროგრამის შესრულების შემდეგ შედეგს:

```
წინასწარი ზრდით: 1
წინასწარი კლებით: 0
```

```
შემდგომი კლებით: 0
შემდგომი ზრდით: -1
უნარულის საბოლოო მნიშვნელობა: 0
დადებითი: 1
უარყოფითი: -1
ბიტური არა: -1
ლოგიკური არა: True
```

გარდა უნარული ოპერატორებისა C#-ს აქვს ბინარული ოპერაციები, რომლებიც გამოიყენება ორცვლადიანი გამოსახულებებისთვის. ამონაბეჭდი 2-3. წარმოგვიდგენს ბინარული ოპერაციების გამოყენებას.

ამონაბეჭდი 2-3. ბინარული ოპერაციები: Binary.cs

```
using System;
class Binaruli
{   public static void Main()
  {  int x, y, Sedegi;
     float  mcuraviSedegi;
     x = 7;
     y = 5;
     Sedegi = x+y;
     Console.WriteLine("x+y: {0}", Sedegi);
     Sedegi = x-y;
     Console.WriteLine("x-y: {0}", Sedegi);
     Sedegi = x*y;
     Console.WriteLine("x*y: {0}", Sedegi);
     Sedegi = x/y;
     Console.WriteLine("x/y: {0}", Sedegi);
     mcuraviSedegi = (float)x/(float)y;
     Console.WriteLine("x/y: {0}", mcuraviSedegi);
     Sedegi = x%y;
     Console.WriteLine("x%y: {0}", Sedegi);
     Sedegi += x;
     Console.WriteLine("Sedegi +=x: {0}", Sedegi);
  }
}
```

პროგრამის გამოსავალი:

```
x+y: 12
x-y: 2
```

```
x*y: 35
x/y: 1
x/y: 1.4
x%y: 2
Sedegi+=x: 9
```

ამონაბეჭდში 2-3 ნაჩვენებია ბინარული ოპერაციების მაგალითები. როგორც მოსალოდნელი იყო, ოპერაციები addition (+), subtraction (-), multiplication (*) და division (/) იძლევა სწორ არითმეტიკულ შედეგებს.

ცვლადი *mcuraviSedegi* მცურავწერტილიანი ტიპისაა. ჩვენ ცხადად გარდავქმნით მთელი ტიპის ცვლადებს *x* და *y* მცურავწერტილიანი ტიპის მნიშვნელობად (float).

არის აგრეთვე ნაშთის (%)ოპერაციის მაგალითი. ის ასრულებს ორი ცვლადის ერთმანეთზე გაყოფის ოპერაციას და იძლევა შედეგის ნაშთს.

ბოლო ოპერატორი გვიჩვენებს მინიჭების ოპერატორის სხვაგვარ ფორმას (+=) ოპერაციას. ამ ოპერაციის ეკვივალენტი შეიძლება ასე ჩაიწეროს: *Sedegi = Sedegi + x;* შედეგს აქვს იგივე მნიშვნელობა. როგორც ვნახეთ, (+=) ოპერაციის დროს, მხოლოდ ერთხელ ვიყენებთ ცვლადს, ბოლო ჩანაწერი კი ტრადიციულია.

მასივის ტიპი

მონაცემთა შემდეგი ტიპი არის მასივი (*Array*), რომელიც შეიძლება წარმოვიდგინოთ, როგორც კონტეინერი სპეციალური ტიპის სიის შესანახად (მასივთან მათემატიკურად ყველაზე ახლოს დგას ვექტორისა და მატრიცის ცნებები). როდესაც აღვწერთ მასივს, განვსაზღვრავთ ტიპს, სახელს, განზომილებებსა და ზომას.

ამონაბეჭდი 2-4. ოპერაციები მასივებზე: Array.cs

```
using System;
class Masivi
{   public static void Main()
    {   int[] mTeli = { 5, 10, 15 };
        bool[][] bulis = new bool[2][];
        bulis[0] = new bool[2];
        bulis[1] = new bool[1];
        double[,] ormagi = new double[2, 2];
        string[] striqoni = new string[3];
Console.WriteLine("მთელი[0]: {0}, მთელი[1]: {1}, მთელი[2]: {2}", mTeli[0], mTeli[1],
mTeli[2]);
        bulis[0][0] = true;
        bulis[0][1] = false;
```

```
        bulis[1][0] = true;
        Console.WriteLine("ბულის[0][0]: {0}, ბულის[1][0]: {1}", bulis[0][0], bulis[1][0]);
        ormagi[0, 0] = 3.147;
        ormagi[0, 1] = 7.157;
        ormagi[1, 1] = 2.117;
        ormagi[1, 0] = 56.00138917;
      Console.WriteLine("ორმაგი[0, 0]: {0}, ორმაგი[1, 0]: {1}", ormagi[0, 0], ormagi[1, 0]);
        striqoni[0] = "  ნინო ";
        striqoni[1] = "  ია ";
        striqoni[2] = "  თეა ";
        Console.WriteLine("სტრიქონი[0]: {0}, სტრიქონი[1]: {1}, სტრიქონი[2]: {2}",
    striqoni[0], striqoni[1], striqoni[2]);
      }}
```

პროგრამის გამოსავალი:

```
მთელი[0]: 5, მთელი[1]: 10, მთელი[2]: 15
ბულის[0][0]: True, ბულის[1][0]: True
ორმაგი[0,0]: 3.147, ორმაგი[1, 0]: 56.00138917
სტრიქონი[0]: ნინო, სტრიქონი[1]: ია, სტრიქონი[2]: თეა
```

ამონაბეჭდი 2-4 გვიჩვენებს მასივების გამოყენების სხვადასხვა სახეს. პირველი მაგალითი არის *mTeli* მასივი. მისთვის საწყისი მნიშვნელობების მინიჭება (ინიციალიზაცია) ხდება აღწერისას ცხადად მოცემული მნიშვნელობების საშუალებით.

შემდეგია ე. წ. უსწორმასწორო (jagged) მასივი სახელად bulis. ის ძირითადად არის მასივების მასივი (ანუ მასივის ელემენტი თავად წარმოადგენს მასივს). ჩვენ გვჭირდება *new* ოპერატორის გამოყენება გარკვეული ზომის ძირითადი მასივის ეგზემპლარის შექმნისთვის და შემდეგ *new* ოპერატორის კვლავ გამოყენება ყოველი ქვემასივისთვის. ამგვარ მასივებს უსწორმასწორო იმიტომ დაარქვეს, რომ ცხრილის სტრიქონებში ცვლადი რაოდენობის ელემენტებია დასაშვები. ეს უკანასკნელი უსწორმასწორობის შეგრძნებას იწვევს. ამგვარი მასივები მეხსიერების დაზოგვის საშუალებას იძლევა. განხილულ მაგალითში პირველი სტრიქონი შედგება ორი ელემენტისგან, ხოლო მეორე - ერთისგან.

მესამე მაგალითი არის ორგანზომილებიანი მასივი *ormagi*. მასივები შეიძლება იყოს მრავალგანზომილებიანი. თითოეული განზომილება გამოიყოფა მძიმით. მათი ეგზემპლარი უნდა შეიქმნას ასევე *new* ოპერატორის საშუალებით. დაბოლოს, გვაქვს *string*-ტიპის ერთგანზომილებიანი მასივი *striqoni*.

ყველა შემთხვევაში შეგიძლიათ დაინახოთ, რომ მასივის ელემენტები ასოციირებულია მთელი ტიპის ინდექსთან, რომლის საშუალებითაც ხდება მათზე მითითება. მასივების ზომები შეიძლება იყოს ნებისმიერი *int* ტიპის მნიშვნელობის და იწყება 1-დან. მათი ინდექსების მნიშვნელობები იწყება 0-დან.

ამონაბეჭდში 2.5 მოყვანილია მასივების განსაზღვრის და გამოყენების, ასევე მისი პარამეტრების დადგენის მაგალითი.

ამონაბეჭდი 2.5. მასივის პარამეტრების განსაზღვრა. masivisparametrebi.cs

```
      using System;
namespace T
{           class klasi1
            {     static int n=5;
                     static void Main()
{double[,]  marTkuTxa={{1,2,3},{4,5,6}};//{{პირველი სტრიქონი},{მეორე სტრიქონი}}
                Console.WriteLine("marTkuTxa.Length={0}",marTkuTxa.Length);
            Console.WriteLine("marTkuTxa.Rank={0}",marTkuTxa.Rank);
Console.WriteLine("marTkuTxa.GetUpperBound()={0}",marTkuTxa.GetUpperBound(0));
Console.WriteLine("marTkuTxa.GetLowerBound()={0}",marTkuTxa.GetLowerBound(0));
Console.WriteLine("marTkuTxa.GetUpperBound()={0}",marTkuTxa.GetUpperBound(1));
Console.WriteLine("marTkuTxa.GetLowerBound()={0}",marTkuTxa.GetLowerBound(1));
                        foreach(double x in marTkuTxa)
                        {     Console.Write("{0},",x);
                }
                Console.WriteLine();
int[][]usWormasWoro ={new int[] {1,2,3,4},new int[]{5,6,7,8,9,10}};
      Console.WriteLine("usWormasWoro.Rank={0}",usWormasWoro.Rank);
Console.WriteLine("usWormasWoro.GetUpperBound()={0}",usWormasWoro[0].GetUpperBou
nd(0));
Console.WriteLine("usWormasWoro.GetLowerBound()={0}",usWormasWoro[0].GetLowerBou
nd(0));
Console.WriteLine("usWormasWoro.GetUpperBound()={0}",usWormasWoro[1].GetUpperBou
nd(0));
Console.WriteLine("usWormasWoro.GetLowerBound()={0}",usWormasWoro[1].GetLowerBou
nd(0));
// უსწორმასწორო მასივის ბეჭდვა
                        foreach(int[] x in usWormasWoro)
                        {     foreach(int xx in x)
                                    Console.Write("{0},",xx);}
                                    Console.WriteLine();
// სამკუთხა მასივის ფორმირება
                  double[][]  samkuTxa=new double[n][];
                  for(int i=0;i<n;i++)
                  samkuTxa[i]=new double[i+1];
                  for(int i=0;i<n;i++)
            {     Console.WriteLine();
                  for(int k=0;k<i;k++)
```

```
            {     samkuTxa[i][k]=k;
                  Console.Write(samkuTxa[i][k]);
            }}
                  Console.WriteLine();
            }}
}
```

გამოსავალი:

marTkuTxa.Length=6

marTkuTxa.Rank=2

marTkuTxa.GetUpperBound()=1

marTkuTxa.GetLowerBound()=0

marTkuTxa.GetUpperBound()=2

marTkuTxa.GetLowerBound()=0

1,2,3,4,5,6,

usWormasWoro.Rank=1

usWormasWoro.GetUpperBound()=3

usWormasWoro.GetLowerBound()=0

usWormasWoro.GetUpperBound()=5

usWormasWoro.GetLowerBound()=0

1,2,3,4,5,6,7,8,9,10,

0

01

012

0123

ამონაბეჭდში 2.5 მოყვანილია მართკუთხა და უსწორმასწორო მასივების განსაზღვრის სხვადასხვა ხერხები. *Rank*-არის მასივის ტიპის თვისება, რომელიც განსაზღვრავს ინდექსების რაოდენობას, ხოლო *Length* - ელემენტების საერთო რაოდენობას. *GetLowerBound()* და *GetUpperBound()* მეთოდებია, რომლებიც განსაზღვრავენ ინდექსების ზედა და ქვედა მნიშვნელობებს შესაბამისად.

ყუთში ჩალაგება/ამოლაგება *(boxing/unboxing)*

ვინაიდან ნებისმიერი ტიპი თავსებადია ობიექტურ ტიპთან, შესაძლებელია მნიშვნელობის ტიპის გარდაქმნა მითითების ტიპად და პირიქით. ამ პროცესს უწოდებენ *boxing/unboxing*. ტერმინი უშუალოდ გამომდინარეობს IL შუალედური ენის თანამოსახელე ბრძანებებიდან. ამონაბეჭდში 2.6. მოყვანილია შესაბამისი მაგალითი.

ამონაბეჭდი 2.6. ჩალაგება/ამოლაგება **boxing.cs.**

```
using System;
class TestBoxing
{     public static void Main()
      {     int i = 19;
            object o = i;  // არაცხადი boxing
            i = 52;        //  i-მნიშვნელობის შეცვლა
            Console.WriteLine("i = {0}", i);
            Console.WriteLine("o = {0}", o);
            int j = (int) o;;// ცხადი unboxing-გი
        Console.WriteLine("j = {0}", i);
      }
}
```

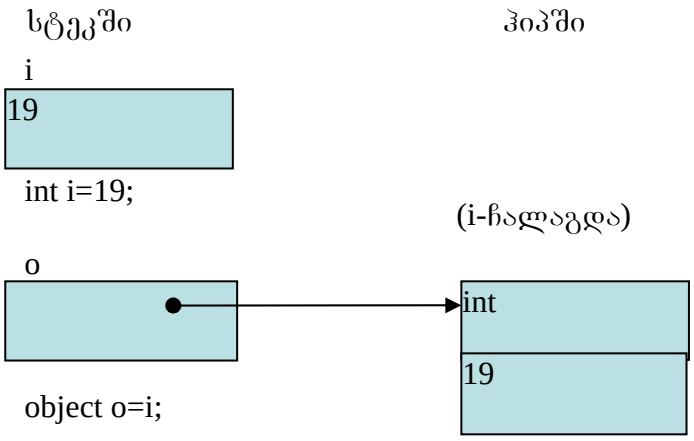
პროგრამის გამოსავალი:

i = 52

o = 19

j=19

ნახ. 2.1. ჩალაგების პროცესის ილუსტრაცია



როგორც ნახ.-დან 2.1 ჩანს, მნიშვნელობის ტიპის მინიჭებისას ობიექტური ტიპისადმი იქმნება ობიექტი, რომლის ტიპი იგივეა, რაც მინიჭებული ცვლადის. მეხსიერების გამოყოფა ხდება ობიექტებისთვის განკუთვნილ არეში ე. წ. "ჰიპში". მნიშვნელობის მქონე ცვლადებისთვის მეხსიერება გამოიყოფა ე. წ. "სტეკში".

თავი III

მართვის ოპერატორები – Selection (შერჩევა)

ამ გაკვეთილის მიზანია შემდეგი საკითხების განხილვა:

- *if* ოპერატორი.
- *switch* ოპერატორი.
- *break*-ის გამოყენება *switch* ოპერატორში.
- *goto* ოპერატორის სწორი გამოყენება.

if ოპერატორი

if ოპერატორი საშუალებას იძლევა, შევასრულოთ პროგრამის სხვადასხვა ნაწილი მოცემული პირობების მიხედვით. როცა პირობის გამოთვლა იძლევა *true* მნიშვნელობას, სრულდება ამ პირობის შესაბამისი პროგრამული ბლოკი (ოპერატორი ან ოპერატორთა მიმდევრობა). გვეძლევა ერთადერთი *if* ოპერატორის, მრავალი *else if* ოპერატორისა და არჩევითი (არასავალდებულო) *else* ოპერატორის გამოყენების არჩევანი. ამონაბეჭდი 3-1 გვიჩვენებს, თუ როგორ მუშაობს ამ სახის თითოეული ოპერატორი.

ამონაბეჭდი 3-1. ოპერატორი *if* : IfarCevani.cs

```
using System;
class IfarCevani
{   public static void Main()
  {  string striqoni;
     int mTeli;
     Console.Write("გთხოვთ შეიყვანოთ რიცხვი: ");
     striqoni = Console.ReadLine();
     mTeli = Int32.Parse(striqoni); //სტრიქონის გარდაქმნა მთელ ტიპად
   // mTeli = System.Convert.ToInt32(striqoni); იგივე შედეგს იძლევა
     // ერთადერთი გადაწყვეტილება და ქმედება ფიგურულ ფრჩხილებში
     if (mTeli > 0)
     {         Console.WriteLine("თქვენი რიცხვი {0} მეტია ნულზე.", mTeli);
     }
     // ერთადერთი გადაწყვეტილება და ქმედება ფიგურულის გარეშე
     if (mTeli < 0)
        Console.WriteLine("თქვენი რიცხვი {0} ნაკლებია ნულზე.", mTeli);
     if (mTeli!= 0)
     {   Console.WriteLine("თქვენი რიცხვი {0} არ არის ნულის ტოლი.", mTeli);
     }
```

```
else
{ Console.WriteLine("თქვენი რიცხვი {0} ტოლია ნულის.", mTeli);
}
// მრავალშემთხვევიანი გადაწყვეტილება
if (mTeli < 0 || mTeli == 0)
{ Console.WriteLine("თქვენი რიცხვი {0} ნაკლებია ან ტოლია ნულის .", mTeli);
}
else if (mTeli > 0 && mTeli <= 10)
{ Console.WriteLine("თქვენი რიცხვი {0} მოთავსებულია 1 დან 10.", mTeli);
}
else if (mTeli > 10 && mTeli <= 20)
{ Console.WriteLine("თქვენი რიცხვი {0} მოთავსებულია 11 დან 20.", mTeli);
}
else if (mTeli > 20 && mTeli <= 30)
{ Console.WriteLine("თქვენი რიცხვი {0} მოთავსებულია 21 დან 30.", mTeli);
}
else
{ Console.WriteLine("თქვენი რიცხვი {0} მეტია, ვიდრე 30.", mTeli);
}}}
```

ამონაბეჭდში 3-1 ოპერატორებში გამოყენებულია ერთი და იგივე შესაყვანი ცვლადი *mTeli,* როგორც მათი გამოთვლების (შესრულების) ნაწილი. არის მონაცემთა ინტერაქტიული შეყვანის სხვა გზაც. ქვემოთ მოყვანილია შესაბამისი პროგრამის კოდი:

```
Console.Write("გთხოვთ შეიყვანოთ რიცხვი: ");
striqoni = Console.ReadLine();
mTeli = Int32.Parse(striqoni);
```

პირველად ვბეჭდავთ კონსოლზე "გთხოვთ შეიყვანოთ რიცხვი: ". ოპერატორი *Console.ReadLine()* იწვევს პროგრამის ლოდინს მანამ, სანამ მომხმარებელი შეიყვანს მონაცემებს, რომელიც ბეჭდავს რიცხვს და აჭერს ღილაკს *Enter*. რიცხვი ბრუნდება სტრიქონის სახით ცვლადში *striqoni*, რომელიც სტრიქონის ტიპისაა. ვინაიდან უნდა გამოვთვალოთ მომხმარებლის შეტანილი ინფორმაცია *int* ტიპის სახით, *striqoni*-ი უნდა გადამუშავდეს. ეს ბრძანების *Int32.Parse(striqoni)* საშუალებით ხდება. (*Int32* და მსგავსი ტიპის გარდაქმნები განხილულია სხვა თავში, როგორც გაუმჯობესებული ტიპები) შედეგი თავსდება *mTeli* ცვლადში, რომელიც *int* ტიპის არის.

აღსანიშნავია, რომ გვაქვს ჩვენთვის სასურველი ტიპის ცვლადები და გამოვითვლით მათ *if* ოპერატორით. პირველი ოპერატორი არის *if (ლოგიკური გამოსახულება) { ოპერატორები }* ფორმის, როგორც ნაჩვენებია ქვემოთ:

```
// ერთადერთი გადაწყვეტილება, ქმედება მოთავსებულია ფიგურულ ფრჩხილებში
if (mTeli > 0)
{ Console.WriteLine("თქვენი რიცხვი {0} მეტია ნულზე.", mTeli);
}
```

უნდა დაიწყოთ საკვანძო სიტყვით *if*, რასაც მოსდევს ფრჩხილებში მოთავსებული ლოგიკური გამოსახულება. ამ ლოგიკური გამოსახულების გამოთვლის შედეგად უნდა მივიღოთ ჭეშმარიტი ან მცდარი მნიშვნელობა (*true* or *false*). ამ შემთხვევაში პროგრამა ამოწმებს, მეტია თუ არა (>) იგი 0-ზე. თუ გამოსახულების მნიშვნელობა ჭეშმარიტია, სრულდება ფიგურულ ფრჩხილებში მოთავსებული ოპერატორები. (ფიგურულ ფრჩხილებში მოთავსებულ გამოსახულებას ვუწოდებთ ბლოკს "block".) ბლოკში შეიძლება იყოს ერთი ან რამდენიმე ოპერატორი. თუ ლოგიკური გამოსახულების გამოთვლის შედეგად მივიღებთ მცდარ მნიშვნელიბას, მაშინ ხდება ბლოკის შიგნით მოთავსებული ოპერატორების იგნორირება და პროგრამის შესრულება გრძელდება ამ ბლოკის მომდევნო ოპერატორით.

შემდეგი *if* ოპერატორი პირველის ანალოგიურია იმის გამოკლებით, რომ მას არ აქვს ბლოკი, როგორც ეს ნაჩვენებია ქვემოთ:

```
// ერთადერთი გადაწყვეტილება და ქმედება ფრჩხილების გარეშე
if (mTeli < 0)
   Console.WriteLine ("თქვენი რიცხვი {0} ნაკლებია ნულზე.", mTeli);
```

თუ ლოგიკური გამოსახულების მნიშვნელობა ჭეშმარიტია, სრულდება ლოგიკური გამოსახულების მომდევნო პირველივე ოპერატორი. როცა ლოგიკური გამოსახულების მნიშვნელობა მცდარია, ლოგიკური გამოსახულების მომდევნო პირველ ოპერატორს გამოვტოვებთ და ვასრულებთ პროგრამის შემდეგ ბრძანებას. *if* ოპერატორის ეს ფორმა მართებულია, როცა შესასრულებელია მხოლოდ ერთი ოპერატორი. თუ გსურთ შეასრულოთ ორი ან მეტი ოპერატორი, როცა ლოგიკური გამოსახულების მნიშვნელობა ჭეშმარიტია, ისინი უნდა მოათავსოთ ბლოკში. გირჩევთ, *if* ოპერატორის შესაბამისი ოპერატორები ყოველთვის მოათავსოთ ბლოკში, იმისდა მიუხედავად, ერთია ოპერატორი თუ მეტი. ეს დაგეხმარებათ შეცდომების თავიდან აცილებაში, თუ მოგვიანებით ოპერატორის დამატებას გადაწყვეტთ და დაგავიწყდებათ მათი მოთავსება ფიგურულ ფრჩხილებში. გარდა ამისა, ფიგურული ფრჩხილების დამატებითი გამოყენება უფრო კითხვადს ხდის პროგრამას.

ხშირ შემთხვევებში შეიძლება დამატებითი პირობების შემოწმება, როცა არ სრულდება წინა პირობა. ეს if/else ოპერატორით ხორციელდება. ეს მესამე ტიპის ოპერატორია, რომელიც ნაჩვენებია ამონაბეჭდში 3-1:

```
// ან ერთი ან მეორე გადაწყვეტილება
if (mTeli != 0)
{   Console.WriteLine("თქვენი რიცხვი {0} არ არის ნულის ტოლი.", mTeli);
}
```

```
else
{   Console.WriteLine("თქვენი რიცხვი {0} ტოლია ნულის..", mTeli);
}
```

თუ ლოგიკური გამოსახულების მნიშვნელობა ჭეშმარიტია, სრულდება ოპერატორები, რომლებიც უშუალოდ მოსდევს *if* ოპერატორს. როცა ლოგიკური გამოსახულების მნიშვნელობა მცდარია, სრულდება ბლოკში *else* საკვანძო სიტყვის მომდევნო ოპერატორები.

როცა თანამიმდევრულად მრავალი პირობაა შესამოწმებელი, შეგიძლიათ გამოიყენოთ *if* ოპერატორის if/else if/else ფორმა.

კოდის ეს ნაწილი *if* საკვანძო სიტყვით იწყება. ამ შემთხვევაშიც გამოითვლება მომდევნო ბლოკები, თუ ლოგიკური გამოსახულების მნიშვნელობა ჭეშმარიტია. ამ შემთხვევაში, რასაკვირველია, გამოითვლება ლოგიკური პირობების ქვემიმდევრობა *else if* საკვანძო სიტყვებთან. *else if* საკვანძო სიტყვას მოსდევს ლოგიკური გამოსახულება, მსგავსად if საკვანძო სიტყვისა. წესები მსგავსია, როცა ლოგიკური გამოსახულების მნიშვნელობა *else if* საკვანძო სიტყვის შემდეგ ჭეშმარიტია, სრულდება უშუალოდ ლოგიკური გამოსახულების მომდევნო ბლოკი. ეს გრძელდება მანამ, სანამ არ გამოითვლება ყველა შემთხვევა. მაგრამ მთლიანი *if/else if* მიმდევრობა უნდა მთავრდებოდეს დამასრულებელი ნაწილით *else*. როცა არც ერთი ლოგიკური გამოსახულება, რომელიც მოსდევს საკვანძო სიტყვებს *if* ან *else if* , არ იძლევა ჭეშმარიტ ლოგიკურ მნიშვნელობებს, მაშინ სრულდება *else* საკვანძო სიტყვის მომდევნო ბლოკი. სრულდება *if/else if/else* ოპერატორის მხოლოდ ერთი სექცია.

ბოლო ოპერატორის განსხვავება წინამორბედებისგან ლოგიკურ გამოსახულებებშია. ლოგიკური გამოსახულება *(mTeli < 0 || mTeli == 0)* შეიცავს პირობით ოპერაციას OR (||) . ორივე შემთხვევაში ჩვეულებრივი ოპერაცია OR (|) და პირობითი ოპერაცია OR (||) გამოითვლის ჭეშმარიტ მნიშვნელობას, თუ ოპერაციაში მონაწილე ერთი ქვეგამოსახულება მაინც ჭეშმარიტია. ძირითადი განსხვავება OR ოპერაციის ორ ფორმას შორის ისაა, რომ ჩვეულებრივი ოპერაცია OR (|) ყოველთვის გამოითვლის ორივე ქვეგამოსახულებას. მაშინ, როცა პირობითი OR გამოითვლის მეორე ქვეგამოსახულებას მხოლოდ იმ შემთხვევაში, თუ პირველი მცდარია.

ლოგიკური გამოსახულება შეიცავს *(mTeli > 0 && mTeli <= 10)* პირობით AND ოპერაციას. ჩვეულებრივი ოპერაცია AND (&) და პირობითი ოპერაცია AND (&&) ჭეშმარიტ შედეგს იძლევა, როცა ორივე ქვეგამოსახულება ორივე მხარეს ჭეშმარიტია. განსხვავება AND ოპერაციის ამ ორ ფორმას შორის ისაა, რომ ჩვეულებრივი ოპერაცია AND ყოველთვის გამოითვლის ორივე ქვეგამოსახულებას. მაშინ, როცა პირობითი AND ოპერაცია მეორე ქვეგამოსახულებას გამოითვლის მხოლოდ იმ შემთხვევაში, თუ პირველი ჭეშმარიტია.

პირობით ოპერაციებს (&& და ||) საზოგადოდ უწოდებენ შემოკლებულ ოპერაციებს, ვინაიდან ისინი ყოველთვის არ გამოითვლიან მთლიან გამოსახულებას. ამრიგად, მათ აგრეთვე იყენებენ უფრო ეფექტური კოდის შესაქმნელად არასაჭირო ლოგიკის იგნორირების გზით.

Switch-ოპერატორი

პროგრამის მართვის ერთ-ერთი შემდეგი ოპერატორი არის *switch*, რომელიც მოცემული საწყისი პარამეტრებისთვის გამოითვლის თანამიმდევრულად ლოგიკური პირობების სიმრავლეს და პირველი პირობის შესრულებისთანავე ასრულებს შესაბამის შტოს და გამოტოვებს დანარჩენს. *switch* ოპერატორი იყენებს bool, enums, integral და string ტიპის მნიშვნელობებს.

ამონაბეჭდი 3-2 გვიჩვენებს, როგორ გამოიყენება *switch* ოპერატორი მთელი და სტრიქონული ტიპებისთვის.

ამონაბეჭდი 3-2. გადართვის ოპერატორი: SwitchSelection.cs

```
using System;
class GadaerTearCevanze
{   public static void Main()
  {   string striqoni;
      int mTeli;
      dasawyisi:
  Console.Write("გთხოვთ შეიყვანოთ რიცხვი, რომელიც მოთავსებულია 1 და 3 შორის: ");
      striqoni = Console.ReadLine();
      mTeli = Int32.Parse(striqoni);
      // switch with integer type
      switch (mTeli)
      {   case 1:
            Console.WriteLine("თქვენი რიცხვია {0}.", mTeli);
            break;
          case 2:
            Console.WriteLine("თქვენი რიცხვია {0}.", mTeli);
            break;
          case 3:
            Console.WriteLine("თქვენი რიცხვი {0}.", mTeli);
            break;
          default:
            Console.WriteLine("თქვენი რიცხვი {0} არ არის 1 და 3 შორის.", mTeli);
```

```
            break;
        }
        gadawyvetileba:
Console.Write("დაბეჭდე \"გააგრძელე\" შესასრულებლად \"დაასრულე\" გასაჩერებლად: ");
        striqoni = Console.ReadLine();
        // switch with string type
        switch (striqoni)
        {  case "gaagrZele":
            goto dasawyisi;
           case "daasrule":
            Console.WriteLine("ნახვამდის.");
            break;
           default:
            Console.WriteLine("თქვენს მიერ შეყვანილი {0} არ არის სწორი.", striqoni);
            goto gadawyvetileba;
        }}}
```

ამონაბეჭდი 3-2 გვიჩვენებს *switch* ოპერატორების წყვილს. *switch* ოპერატორი საკვანძო სიტყვით *switch* იწყება, რომელსაც მოსდევს *switch* გამოსახულება. პირველ *switch* ოპერატორში ამონაბეჭდში 3-2 *switch* გამოსახულება იყენებს *int* ტიპს.

Switch ბლოკი მოსდევს *switch* გამოსახულებას, სადაც ერთი ან რამდენიმე შესაძლო ვარიანტი უნდა ეთანადებოდეს *switch* გამოსახულებას. თითოეული შემთხვევა დაჭდევებულია საკვანძო სიტყვით *case*, რომელსაც მოსდევს *switch* გამოსახულების ტიპის მქონე კონკრეტული მნიშვნელობა, რომელიც მთავრდება ორი წერტილით (:). ჩვენს შემთხვევაში გვაქვს *case 1:*, *case 2:* და *case 3:*. როცა გამოითვლება (განისაზღვრება, შესრულდება) *switch* გამოსახულება, მისი მნიშვნელობის შესაბამისობისას კონკრეტულ შემთხვევასთან სრულდება ამ შემთხვევის მომდევნო ოპერატორები განშტოების (მართვის გადაცემის) ოპერატორების ჩათვლით, რომელიც შეიძლება იყოს *break, continue, goto* , *return*, ან *throw*. ცხრილში 3-1 შეჯამებულია განშტოების ოპერატორები.

ცხრილი 3-1. C#-ის განშტოების ოპერატორები

განშტოების ოპერატორები	**აღწერა**
break	ტოვებს switch ბლოკს
continue	წყვეტს ციკლის ოპერატორების შესრულებას და გადადის ციკლის შემდეგ ბიჯზე
goto	ტოვებს ციკლის ბლოკს და გადადის პირდაპირ კონკრეტულ ჭდეზე
return	ტოვებს კონკრეტულ მეთოდს მისი შედეგის დაბრუნებით და გადადის ამ მეთოდის გამომძახებელი ოპერატორის მომდევნო ოპერატორზე

throw	იწვევს გამონაკლისის ინიციირებას (იხ. მე-15 თავი)

შეგიძლიათ ასევე გამოიყენოთ სათადარიგო (ნაგულისხმები) *default* შემთხვევა, რომელიც მოსდევს ყველა დანარჩენ შემთხვევას. თუ არც ერთი წინა შემთხვევა არ სრულდება (არ აკმაყოფილებს პირობას), მაშინ ამოირჩევა *default* შემთხვევა და სრულდება მისი შესაბამისი ოპერატორები. თუმცა *default* ჭდის გამოყენება არჩევითია (არასავალდებულო), უკიდურესად რეკომენდებულია ყველა შემთხვევაში მისი გამოყენება. ეს დაგეხმარებათ გაუთვალისწინებელი შემთხვევების დაძლევაში და თქვენს პროგრამას უფრო მდგრადს გახდის.

ყოველი *case* ჭდე უნდა მთავრდებოდეს გადართვის ოპერატორით, როგორც აღწერილია ცხრილში 3-1. ჩვეულებრივ ეს არის *break* ოპერატორი. *break* ოპერატორი იწვევს *switch* ოპერატორიდან გამოსვლას და პროგრამის შესრულების გაგრძელებას *switch* ბლოკის მომდევნო ოპერატორიდან. არსებობს ორი გამონაკლისი: თანამიმდევრული *case* ოპერატორები, რომელთა შორისაც არ არის სხვა ოპერატორები, და ოპერატორი *goto*. ქვემოთ მოყვანილ მაგალითში ნაჩვენებია case ოპერატორების კომბინაცია:

```
switch (mTeli)
{ case 1:
  case 2:
  case 3:
    Console.WriteLine("თქვენი რიცხვია {0}.", mTeli);
    break;
  default:
    Console.WriteLine("თქვენი რიცხვი {0} არ არის 1 და 3 შორის.", mTeli);
    break;
}
```

Case ოპერატორების, რომელთა შორის არ არის სხვა ოპერატორები, ერთად მოთავსებით, თქვენ ქმნით ერთ არჩევანს მრავალი მნიშვნელობისთვის. Case კოდის გარეშე ავტომატურად გადასცემს მართვას მომდევნო case-ს. ზემომოყვანილ მაგალითში *mTeli* მნიშვნელობებისთვის 1, 2 ან 3 სრულდება ოპერატორი (კოდი), რომელიც მოსდევს *case 3*-ს.

მართვის მიმდინარეობის შეცვლის სხვა გზა არის *switch* ოპერატორში *goto* ოპერატორის გამოყენება. თქვენ შეგიძლიათ გადახვიდეთ სხვა case ოპერატორზე ან საერთოდ გამოხვიდეთ switch ოპერატორიდან. შემდეგი *switch* ოპერატორი ამონაბეჭდში 3-2 გვიჩვენებს *goto* ოპერატორის გამოყენებას.

ოპერატორი *goto* იწვევს გადასვლას იმ ჭდეზე, რომელიც მოსდევს საკვანძო სიტყვას *goto*. პროგრამის შესრულებისას, თუ მომხმარებელი აკრეფს "continue"-ს, მაშინ შესრულდება ოპერატორი *"goto dasawyisi:"*. პროგრამა დატოვებს switch ოპერატორს და შესრულება

გაგრძელდება ოპერატორით, რომელიც დაჭდევებულია *dasawyisi*: სიტყვით. ამას აქვს ციკლის ეფექტი, რაც საშუალებას გვაძლევს, შევასრულოთ ერთი და იგივე კოდი მრავალჯერ. ციკლი დამთავრდება, თუ მომხმარებელი აკრეფს "quit". ამ შემთხვევაში მოხდება *case "quit"*-ის არჩევა, შესრულდება მისი მომდევნო ოპერატორი და დაიბეჭდება "Bye". რა თქმა უნდა, ციკლი უნდა შეიცავდეს კონსოლიდან შეყვანის ოპერატორს. თუ არ იქნა შეყვანილი არც "quit" და არც "continue", მაშინ შესრულდება default საკვანძო სიტყვით მონიშნული (დაჭდევებული) ოპერატორი და შემდეგ გადასვლა მოხდება "gadawyvetileba:" ჭდით მონიშნულ ოპერატორზე. აქ კვლავ შეიძლება გამოვიყენოთ შეყვანის ოპერატორი, რათა გადავწყვიტოთ, გავაგრძელოთ თუ დავასრულოთ ციკლი.

თავი IV
ციკლის ოპერატორები

მოცემულ თავში განხილულია შედეგი საკითხები:

- ოპერატორი *while* loop.
- ოპერატორი *do* loop.
- ოპერატორი *for* loop.
- ოპერატორი *foreach* loop.
- ოპერატორი *break*.
- ოპერატორი *continue* .

ციკლი *while*

ციკლი **while** ამოწმებს პირობას და თუ ის ჭეშმარიტია, იმეორებს ბლოკის შესრულებას მანამ, სანამ პირობა არ გახდება მცდარი. მისი სინტაქსი შემდეგია : ***while (<ლოგიკური გამოსახულება>) { <ოპერატორები> }.*** ოპერატორები შეიძლება იყოს **C#** ნებისმიერი დასაშვები ოპერატორები. ლოგიკური გამოსახულება გამოითვლება ბლოკის შესრულებამდე. თუ ლოგიკური გამოსახულება ჭეშმარიტია, მაშინ სრულდება ბლოკი და მართვა გადაეცემა ***while*** ოპერატორის დასაწყისს, რათა ლოგიკური გამოსახულება (პირობა) შემოწმდეს თავიდან. როდესაც პირობა გახდება მცდარი, მაშინ პროგრამის შესრულება გრძელდება ***while*** ოპერატორის მომდევნო ოპერატორიდან ანუ რაც მოსდევს ფიგურულ ფრჩხილებში მოთავსებულ ბლოკ-ოპერატორებს. ამგვარი ციკლის ორგანიზებისას ყურადღება უნდა მიექცეს იმას, რომ ლოგიკურ გამოსახულებაში მონაწილე ცვლადები ბლოკის შესრულებისას იმგვარად განახლდეს, რომ რაღაც მომენტში ლოგიკური პირობა დაირღვეს. წინააღმდეგ შემთხვევაში ბლოკის ოპერატორები მუდმივად შესრულდება, ე. ი. გვექნება ე. წ. პროგრამის ჩაციკვლა. ამონაბეჭდში 4-1 მოცემულია ციკლი ***while*** გამოყენების მაგალითი.

ამონაბეჭდი 4-1. While ციკლი: WhileLoop.cs

```
using System;
class WhileLoop
{   public static void Main()
  {      int mTeli = 0;
     while (mTeli < 10)
     {  Console.Write("{0} ", mTeli);
```

```
        mTeli++;
    }
    Console.WriteLine();
  }}
```

ამონაბეჭდში 4-1 ნაჩვენებია *while* ციკლის მარტივი მაგალითი. ის იწყება საკვანძო სიტვით *while,* რომელსაც მოსდევს ლოგიკური გამოსახულება. ამ შემთხვევაში ვამოწმებთ *mTeli* ცვლადს იმის დასადგენად, ნაკლებია (<) თუ არა იგი 10-ზე. ვინაიდან მისი საწყისი მნიშვლობა 0 იყო, ლოგიკური პირობის (გამოსახულების) მნიშვნელობა ჭეშმარიტი იქნება. პირველი შემოწმებისას, რადგან ლოგიკური გამოსახულების მნიშვნელობა ჭეშმარიტია, სრულდება ბლოკში მოთავსებული ოპერატორები. ხდება რიცხვის დაბეჭდვა ეკრანზე (კონსოლზე). ოპერატორი (++) *mTeli* ზრდის ამ ცვლადის მნიშვნელობას 1-ით. ბლოკში მოთავსებული ოპერატორების შესრულების პროცესი გრძელდება მანამ, სანამ mTeli ცვლადის მნიშვნელობა არ გახდება 10. ამ შემთხვევაში ლოგიკური გამოსახულების მნიშვნელობა მცდარი გახდება და მართვა გადაეცემა ციკლის ოპერატორის მომდევნო ოპერატორს. ამ შემთხვევაში კონსოლზე ამოიბეჭდება რიცხვები 0-დან 9-მდე და ბეჭდვის პოზიცია გადაინაცვლებს შემდეგ სტრიქონში (ანუ დაიბეჭდება ცარიელი სტრიქონი).

***do* ციკლი**

do ციკლი while ციკლის მსგავსია. განსხვავება მხოლოდ იმაშია, რომ იგი პირობას ამოწმებს მხოლოდ ბლოკის ბოლოს. ეს იძლევა იმის გარანტიას, რომ ბლოკი ერთხელ მაინც აუცილებლად შესრულდება. while ციკლის შემთხვევაში ამის გარანტია თვით პროგრამისტმა უნდა უზრუნველყოს, ანუ ისეთი საწყისი მნიშვნელობები მიანიჭოს ლოგიკურ გამოსახულებაში შემავალ ცლადებს, რომ მისი მნიშვნელობა პირველივე შესრულებისას აღმოჩნდეს ჭეშმარიტი (true).

ქვემოთ, ამონაბეჭდში 4-2, მოყვანილია do ციკლის ოპერატორის გამოყენების მაგალითი.

ამონაბეჭდი 4-2. Do ციკლი: Docikli.cs

```
using System;
class Docikli
{   public static void Main()
  {      string arCevani;
     do
     { //  მენიუს ბეჭდვა
        Console.WriteLine("ჩემი მისამართების წიგნია\n");
        Console.WriteLine("A – დაამატეთ ახალი მისამართი");
        Console.WriteLine("D – წაშალეთ მისამართი");
```

```
    Console.WriteLine("M – შეცვალეთ მისამართი");
    Console.WriteLine("V – დაათვალიერეთ მისამართები");
    Console.WriteLine("Q - დაასრულეთ\n");
    Console.WriteLine("აირჩიეთ (A,D,M,V,or Q): ");
    // შეყავს მომხმარებლის არჩევანი
    arCevani = Console.ReadLine();
    // მიიღე გადაწყვეტილება მომხმარებლის არჩევანის მიხედვით
    switch(arCevani)
    {  case "A":
       case "a":
          Console.WriteLine("თქვენ გსურთ დაამატოთ მისამართი.");
          break;
       case "D":
       case "d":
          Console.WriteLine("თქვენ გსურთ წაშალოთ an მისამართი.");
          break;
       case "M":
       case "m":
          Console.WriteLine("თქვენ გსურთ შეცვალოთ მისამართი.");
          break;
       case "V":
       case "v":
       Console.WriteLine("თქვენ გსურთ დაათვალიეროთ მისამართების სია.");
          break;
       case "Q":
       case "q":
          Console.WriteLine("ნახვამდის.");
          break;
       default:
          Console.WriteLine("{0} არ არის კორექტული არჩევანი", arCevani);
          break;
    }
    // პაუზა, რომელიც მომხმარებელს არჩევანის გაკეთების საშუალებას აძლევს
    Console.Write("დააჭირეთ Enter ღილაკს გასაგრძელებლად...");
    Console.ReadLine();          Console.WriteLine();
  } while (arCevani != "Q" && arCevani != "q"); // გააგრძელე, მანამ მომხმარებელი
```

```
// არ ისურვებს დამთავრებას
}}
```

ამონაბეჭდი 4-2 გვიჩვენებს do ციკლის ოპერატორს მოქმედებაში. do ციკლის ოპერატორის სინტაქსია *do { < ოპერატორები > } while (<ლოგიკური გამოსახულება >);*. აქ < და > მეტასიმბოლოებია, ანუ ისინი გამოიყენება სინტაქსური წესის აღსაწერად და არ წარმოადგენს ენის ნაწილს.

Main() –მეთოდში *arCevani* ცვლადი აღწერილია (გამოცხადებულია) *string* ტიპად. შემდეგ კონსოლზე იბეჭდება მენიუს წინადადებების მიმდევრობა. პროგრამამ უნდა მიიღოს (შეიყვანოს მონაცემები) კონსოლიდან, რისთვისაც გამოყენებულია *Console.ReadLine()* მეთოდი, რომელიც მომხმარებლის მიერ დაბეჭდილ მონაცემებს ანიჭებს *myChoic* ცვლადს. ამ ცვლადის მნიშვნელობის მიხედვით უნდა გავაკეთოთ არჩევანი, რის საუკეთესო საშუალებაცაა *switch* ოპერატორი. პროგრამაში დიდი და პატარა ასოს გამოყენების შემთხვევები გაერთიანებულია ერთ შემთხვევად.

for ციკლი

for ციკლი მუშაობს while ციკლის მსგავსად, მხოლოდ იმ განსხვავებით, რომ *for* ციკლს გააჩნია ინიციალიზაციისა და პირობის მოდიფიკაციები. *for* ციკლი იმ შემთხვევაშია გამოსადეგი, როცა ცნობილია, რამდენჯერ უნდა შესრულდეს ოპერატორები ციკლის შიგნით. *for* საკვანძო სიტყვის შემდეგ ფრჩხილებში მოთავსებული გამოსახულება შედგება წერტილ-მძიმით გამოყოფილი სამი სექციისგან. *for* ციკლის ოპერატორის სინტაქსი შემდეგნაირად გამოიყურება: *(<ინიციალიზაციის სია>; <ლოგიკური გამოსახულება>; <იტერაციის სია>) { <ბრძანებები> }*.

ინიციალიზაციის სია არის მძიმეებით გამოყოფილი გამოსახულებების მიმდევრობა. ეს გამოსახულებები *for* ციკლის არსებობის მანძილზე მხოლოდ ერთხელ გამოითვლება. ეს არის ერთჯერადი ოპერაცია ციკლის პირველ შესრულებამდე. ეს სექცია ჩვეულებრივ გამოიყენება მთელი ტიპის ცვლადის ინიციალიზაციისთვის (მნიშვნელობათა მინიჭებისთვის), რომელიც გამოიყენება, როგორც მთვლელი. ამის შემდეგ ხდება ლოგიკური გამოსახულების გამოთვლა. აქ არის მხოლოდ ერთი ლოგიკური გამოსახულება, მაგრამ იგი შეიძლება იყოს ნებისმიერი სირთულის. მთავარია, იძლეოდეს შედეგს *true* ან *false*. ლოგიკური გამოსახულება ჩვეულებრივ გამოიყენება მთვლელი ცვლადის მნიშვნელობის შესამოწმებლად. როდესაც ლოგიკური გამოსახულების მნიშვნელობა ჭეშმარიტია, სრულდება ფიგურულ ფრჩხილებში მოთავსებული ოპერატორები. ამის შემდეგ მართვა კვლავ გადაეცემა ციკლის დასაწყისს და გამოითვლება გამეორების სია, რომელიც ჩვეულებრივ გამოიყენება მთვლელის გასაზრდელად ან შესამცირებლად. გამეორების სია შეიძლება შეიცავდეს მძიმით გამოყოფილ ოპერატორებს, მაგრამ, როგორც წესი, ეს მხოლოდ ერთი ოპერატორია. ამონაბეჭდი 4-3 გვიჩვენებს, თუ როგორ გამოიყენება *for* ციკლი.

ამონაბეჭდი 4-3. For ციკლი: ForLoop.cs

```
using System;
class Forcikli
{   public static void Main()
  {      for (int i=0; i < 20; i++)
    {          if (i == 10)
               break;
          if (i % 2 == 0)
               continue;
          Console.Write("{0} ", i);       }
      Console.WriteLine();
  }}
```

ჩვეულებრივ *for* ციკლის ოპერატორები სრულდება გამხსნელი ფიგურული ფრჩხილიდან დამხურავ ფიგურულ ფრჩხილამდე წყვეტის გარეშე, თუმცა ამონაბეჭდში 4-3 მოცემულია ორი გამონაკლისი. ეს არის *if* ოპერატორები, რომლებიც ცვლიან პროგრამის მიმდინარეობას *for* ციკლის ბლოკში.

პირველი *if* ოპერატორი ამოწმებს, არის თუ არა *i* 10-ის ტოლი. აქ ხედავთ *break* ოპერატორის სხვა გამოყენებას. მისი მოქმედება მსგავსია selection-ოპერატორებში მისი მოქმედებისა, რაც განვიხილეთ მე-3 თავში. იგი უბრალოდ წყვეტს ციკლს და მართვას გადასცემს *for* ბლოკის მომდევნო ოპერატორს.

შემდეგი *if* ოპერატორი გამოიყენება ნაშთის გამოსათვლელად. იგი ამოწმებს *i* ცვლადს ლუწობაზე (არის თუ არა იგი ორის ჯერადი). იგი იძლევა ჭეშმარიტ მნიშვნელობას, თუ *i* უნაშთოდ იყოფა ორზე (ნაშთი უდრის 0). როდესაც ზემომოყვანილი პირობა ჭეშმარიტია, სრულდება *continue* ოპერატორი, რაც იწვევს მართვის გადაცემას ციკლის გამეორების სიაში.

როცა პროგრამის მართვა აღწევს *continue* ოპერატორს ან ბლოკის დასასრულს, იტერაციის სიაზე გადასვლა ხდება აგრეთვე ციკლის ბლოკის დასრულებისას. ეს არის მოქმედებათა მძიმეებით გამოყოფილი თანამიმდევრობა. ამონაბეჭდში 4-3 ნაჩვენებია სტანდარტული ქმედება, რაც გულისხმობს მთვლელის გაზრდას. როგორც კი ის დასრულდება, მართვა გადაეცემა *for* ციკლის ლოგიკური გამოსახულების გამოთვლას.

მსგავსად *while* ციკლისა *for* ციკლიც გრძელდება მანამ, სანამ ლოგიკური გამოსახულება ჭეშმარიტია. როგორც კი ლოგიკური გამოსახულება მცდარი ხდება, მართვა გადაეცემა *for* ბლოკის მომდევნო ოპერატორს.

რა თქმა უნდა, *break* და *continue* ოპერატორები შეიძლება გამოყენებულ იქნეს ციკლის ნებისმიერ ოპერატორში.

***foreach* ციკლი**

foreach ციკლი გამოიყენება სიაში შემავალი ელემენტების წასაკითხად. იგი ოპერირებს მასივებზე ან კოლექციებზე, როგორიცაა ArrayList, რომელიც შეგიძლიათ იხილოთ სახელსივრცეში System.Collections. foreach ციკლის სინტაქსი შემდეგნაირად გამოიყურება: *foreach (<ტიპი> <ელემენტის სახელი> in <სია>) { <ოპერატორები> }*. ტიპი არის სიაში შემავალი ელემენტის ტიპი. მაგალითად, თუ სიის ტიპი არის *int[]* , მაშინ ტიპი იქნება *int*.-ი.

ელემენტის სახელი თქვენს მიერ არჩეული იდენტიფიკატორია, რომელიც შეიძლება იყოს ნებისმიერი, მაგრამ სასურველია, რომ ჰქონდეს შინაარსობრივი დატვირთვა. მაგალითად, თუ სია შეიცავს ადამიანთა წლოვანების მასივს, მაშინ შინაარსობრივი სახელი სასურველია იყოს "ასაკი". საკვანძო სიტყვა *in* აუცილებელია.

ზოგადად, კოლექცია არის ნებისმიერი ობიექტების კრებული. System.Collections სახელსივრცეში შეგვიძლია ყველა ხელმისაწვდომი ტიპი (კლასი) ვიხილოთ.

ციკლში *foreach* სიიდან ელემენტების წაკითხვისას სია არის მხოლოდ კითხვადი (read-only). ეს იმას ნიშნავს, რომ ვერ შეცვლით მასივის ელემენტებს foreach ციკლში.

ყოველი იტერაციისას *foreach* ციკლში სიიდან ახალი მნიშვნელობა მოითხოვება. ეს გრძელდება მანამ, სანამ ციკლს შეუძლია, მოგვცეს მნიშვნელობა. ეს მნიშვნელობა თავსდება ელემენტის ცვლადში, რომელიც მხოლოდ კითხვადია, რასაც მოსდევს ოპერატორების *foreach* ციკლის ბლოკის ოპერატორების შესრულება. მთელი კოლექციის გავლის შემდეგ მართვა გადაეცემა *foreach* ბლოკის მომდევნო პირველ შესრულებად ოპერატორს. ამონაბეჭდი 4-4 წარმოგვიდგენს *foreach* ციკლს.

ამონაბეჭდი 4-4. ForEach ციკლი: ForEachLoop.cs

```
using System;
class ForEachcikli
{   public static void Main()
  {      string[] saxelebi = {"ხათუნა", "ნინო", "ია", "თეა"};
     foreach (string person in saxelebi)
     {          Console.WriteLine("{0} ", person);
 }
 }}
```

ამონაბეჭდში 4-4 პირველი, რაც გავაკეთეთ *Main()* მეთოდის შიგნით, არის ის, რომ აღვწერეთ და მივანიჭეთ საწყისი მნიშვნელობები *saxelebi* მასივს 4 სტრიქონის სახით. ეს არის სია, რომელიც *foreach* ციკლში გამოიყენება. ელემენტად გამოყენებულია ცვლადი სახელად "person", რომელშიც თანამიმდევრულად თავსდება *saxelebi* მასივის ყოველი შემადგენელი ელემენტი ციკლის განმავლობაში. მანამ, სანამ მასივი იძლევა სახელებს, ვიყენებთ *Console.WriteLine()* მეთოდს *person* ცვლადის თითოეული მნიშვნელობის ეკრანზე ამოსაბეჭდად.

თავი V
მეთოდები

ამ სახელმძღვანელოს წინა თავებში პროგრამის მთელი ფუნქციონირება განთავსებული იყო *Main()* მეთოდში. ეს მოხერხებული იყო მცირე პროგრამებისთვის, როცა ვიხილავდით პროგრამირების ადრეულ კონცეფციებს. მეთოდები გვეხმარება კოდის გამოყოფაში მოდულების სახით, რომლებიც მოცემულ დავალებას ასრულებენ. მეთოდის სინონიმებად პირველ მიახლოებაში შეიძლება აღვიქვათ ფუნქციები და პროცედურები ტრადიციული პროგრამირების ენებიდან. მოცემულ თავში განხილულია შემდეგი საკითხები:

- მეთოდის სტრუქტურა.
- განსხვავება სტატიკურ და ეგზემპლარის მეთოდებს შორის.
- ობიექტის ეგზემპლარის შექმნა.
- მეთოდის გამოძახება შექმნილი ობიექტისთვის.
- პარამეტრების 4 ტიპი.
- *this* საკვანძო სიტყვის გამოყენება.

მეთოდის სტრუქტურა

მეთოდები საშუალებას იძლევა, დავყოთ პროგრამა სხვადასხვა ნაწილებად. შეგვიძლია მივაწოდოთ ინფორმაცია მეთოდებს. ისინი შეასრულებენ ერთ ან მეტ ოპერატორს და დაგვიბრუნებენ შედეგს. პარამეტრების გადაცემის და მათი მნიშვნელობების დაბრუნება არჩევითია და დამოკიდებულია იმაზე, თუ რა გვინდა, რომ გააკეთოს მეთოდმა. ქვემოთ მოყვანილია მეთოდის აღწერის სინტაქსი:

ატრიბუტები მოდიფიკატორები დაბრუნების-ტიპი მეთოდის-სასახელი(პარამეტერები) { ოპერატორები }

ატრიბუტებსა და მოდიფიკატორებს ქვემოთ განვიხილავთ, შემდეგ თავებში. დაბრუნების ტიპი (return-type) შეიძლება C#-ის ნებისმიერი ტიპი იყოს. იგი შეიძლება მივანიჭოთ ცვლადს შემდგომი გამოყენებისთვის. მეთოდის სახელი უნიკალური იდენტიფიკატორია, რომლის საშუალებითაც ვიძახებთ მეთოდს. პროგრამის უკეთესი კითხვადობისთვის სასურველია, რომ მეთოდის სახელს შინაარსობრივი დატვირთვა ჰქონდეს და შეესაბამებოდეს იმ დავალებას, რომელსაც იგი ასრულებს. პარამეტრები საშუალებას იძლევა, გავაგზავნოთ და დავაბრუნოთ ინფორმაცია მეთოდ(იდან)ში. პარამეტრები გამოყოფილია ფრჩხილებით. ფიგურულ ფრჩხილებში მოთავსებული ოპერატორები ასახავს მეთოდის ფუნქციონირებას.

ამონაბეჭდი 5-1. ერთი მარტივი მეთოდი: ErTimeTodi.cs

```
using System;
class ErTimeTodi // ამ კლასში აღწერილია მეთოდი
```

```
{   string miiRearCevani()  //  უარგუმენტებო მეთოდი
   {      string arCevani;
      // მენიუს ბეჭდვა
   Console.WriteLine("ჩემი მისამართების წიგნია\n");
   Console.WriteLine("A – დაამატეთ ახალი მისამართი");
        Console.WriteLine("D – წაშალეთ მისამართი");
        Console.WriteLine("M – შეცვალეთ მისამართი");
        Console.WriteLine("V – დაათვალიერეთ მისამართები");
        Console.WriteLine("Q - დაასრულეთ\n");
        Console.WriteLine("აირჩიეთ (A,D,M,V,or Q): ");
        // შეყავს მომხმარებლის არჩევანი
        arCevani = Console.ReadLine();
      Console.WriteLine();
      return arCevani; // მეთოდი აბრუნებს შედეგს: მომხმარებლის არჩევანს
   }}
class mTavari  // ეს კლასი საჭიროა პროგამის გასაშვებად
{  public static void Main() //მთავარი მეთოდი
   {       string arCevani;
      ErTimeTodi meTodisnimuSi = new ErTimeTodi();
      do      {
         arCevani = meTodisnimuSi.miiRearCevani();
         //მიიღე გადაწყვეტილება მომხმარებლის არჩევანის მიხედვით
         switch(arCevani)
         {    case "A":
            case "a":
               Console.WriteLine("გსურთ დაამატოთ მისამართი.");
               break;
            case "D":
            case "d":
               Console.WriteLine("გსურთ წაშალოთ მისამართი.");
               break;
            case "M":
            case "m":
               Console.WriteLine("გსურთ შეცვალოთ მისამართი.");
               break;
            case "V":
            case "v":
```

```
            Console.WriteLine("გსურთ დაათვალიეროთ მისამართი.");
            break;
        case "Q":
        case "q":
            Console.WriteLine("ნახვამდის.");
            break;
        default:
            Console.WriteLine("{0} არ არის სწორი არჩევანი", arCevani);
            break;
    }
    // პაუზა, საშუალებას იძლევა ვიხილოთ შედეგები
    Console.WriteLine();
    Console.Write("დააჭირეთ Enter ღილაკს გასაგრძელებლად...");
    Console.ReadLine();
    Console.WriteLine();
  } while (arCevani != "Q" && arCevani != "q"); // გააგრძელე, მანამ მომხმარებელი არ
ისურვებს დამთავრებას
}}
```

პროგრამა ამონაბეჭდში 5-1 მსგავსია მე-4 თავში მოცემული პროგრამისა *Docikli*, გარდა ერთი განსხვავებისა - მენიუს ბეჭდვა და ინფორმაციის შეყვანა *Main()* მეთოდიდან გატანილია ახალ მეთოდში, რომელსაც ეწოდება *miiRearCevani()*. დაბრუნების ტიპი არის *string*-ი. ეს სტრიქონი გამოიყენება *Main()* მეთოდის *switch* ოპერატორში. მეთოდის სახელი "*miiRearCevani*" შერჩეულია იმგვარად, რომ იგი აღწერს იმ ქმედებას, რომელსაც ეს მეთოდი ახდენს გააქტიურებისას. ვინაიდან ფრჩხილები ცარიელია, პარამეტრების საშუალებით ინფორმაციის გაცვლას მეთოდსა და ძირითად პროგრამას შორის ადგილი არა აქვს.

მეთოდის ბლოკში პირველად აღვწერეთ ცვლადი *arCevani*. თუმცა მას იგივე სახელი და ტიპი აქვს, რაც *arCevani* ცვლადს *Main()*მეთოდში. ეს ორივე უნიკალური ცვლადებია. ეს ლოკალური ცვლადებია და მხოლოდ იმ ბლოკებშია ხედვადი (არსებობენ), რომლებშიც არიან აღწერილი. სხვა სიტყვებით რომ ვთქვათ, *arCevani* –მ არაფერი იცის *miiRearCevani()* –ში *arCevani* –ის *Main()* მეთოდში არსებობის შესახებ და პირიქით, ე. ი. თუ ჩვენთვის ცნობილი არ არის მეთოდის ტექსტი, ფაქტიურად არ ვიცით, რა ლოკალური ცვლადებია მასში გამოყენებული.

miiRearCevani() მეთოდი ბეჭდავს მენიუს კონსოლზე და ღებულობს მომხმარებლის მიერ შეყვანილ ინფორმაციას. ოპერატორი *return* უკან აბრუნებს მონაცემს *arCevani* ცვლადის საშუალებით მეთოდის გამომძახებელთან, რომელიც მოთავსებულია *Main()* მეთოდში და არის *miiRearCevani()*-ი. გაითვალისწინეთ, რომ *return* ოპერატორის

საშუალებით დაბრუნებული მნიშვნელობა იმავე ტიპის უნდა იყოს, რაც ფუნქციის აღწერაში დაბრუნების ტიპი, ჩვენს შემთხვევაში - *string*.

Main() მეთოდში უნდა შევქმნათ ახალი *ErTimeTodi* ობიექტი, ვიდრე გამოვიყენებდეთ *miiRearCevani()*-ს. ეს არის *miiRearCevani()* –ს აღწერის გამო. ვინაიდან *miiRearCevani()* –ს აღწერისას არ გამოვიყენეთ მოდიფიკატორი *static*, ამიტომ *miiRearCevani()* გახდა ეგზემპლარის მეთოდი. განსხვავება ეგზემპლარისა (კლასის) და *static* მეთოდებს შორის იმაში მდგომარეობს, რომ კლასის მრავალი ეგზემპლარი შეიძლება შეიქმნას და ყოველ ეგზემპლარს ექნება საკუთრივ მისთვის გამოყოფილი *miiRearCevani()* მეთოდი. თუმცა, როცა მეთოდი არის *static*, არ არსებობს მეთოდის არავითარი ეგზემპლარი და შეგიძლიათ გააააქტიუროთ *static* მეთოდის მხოლოდ ერთი აღწერა.

ამგვარად, როგორც დავადგინეთ, *miiRearCevani()* არ არის *static* და ამიტომ იძულებული ვართ, შევქმნათ ახალი ობიექტი იმისთვის, რომ გამოვიყენოთ ეს მეთოდი. ეს მოცემულია აღწერაში *ErTimeTodi meTodisnimuSi = new ErTimeTodi()*. აღწერის მარცხენა მხარე წარმოადგენს მითითებას ობიექტზე *meTodisnimuSi*, რომლის ტიპიც არის *ErTimeTodi* (ანუ *meTodisnimuSi* მითითების – reference - ტიპის ცვლადი). *meTodisnimuSi* ცვლადი არ არის თავად ობიექტი, იგი არის ცვლადი, რომელიც მიუთითებს *ErTimeTodi*-ტიპის ობიექტზე. აღწერის მარჯვენა მხარე წარმოადგენს მითითებას ახალ *ErTimeTodi*-ტიპის ობიექტზე. საკვანძო სიტყვა *new* არის C#-ში ოპერაცია, რომელიც ქმნის კლასის ეგზემპლარს ობიექტის სახით სპეციალური ტიპის მეხსიერებაში, ე. წ. heap-ში. ახლა, როდესაც შევქმენით *ErTimeTodi*-ტიპის ობიექტის ეგზემპლარი და მასზე მივუთითებთ *meTodisnimuSi* ცვლადით, შეგვიძლია ობიექტის შემადგენელ ნაწილებთან მანიპულირება ამ ცვლადის საშუალებით.

მეთოდები, ველები და კლასის სხვა წევრები შეიძლება მითითებულ იქნას "." (dot) ოპერატორით. თუ გვსურს *miiRearCevani()*-ს გამოძახება, ვწერთ: *meTodisnimuSi.miiRearCevani()*. ამის შემდეგ პროგრამა ასრულებს *miiRearCevani()* ბლოკს და უკან აბრუნებს შედეგებს. შედეგს ვანიჭებთ *arCevani* ცვლადს *Main()* მეთოდში მინიჭების ოპერატორის საშუალებით "=" (assignment). აქედან პროგრამის დარჩენილი ნაწილი სრულდება წინა თავებში აღწერილის ანალოგიურად.

ამონაბეჭდში 5-2 ნაჩვენებია მეთოდის პარამეტრების გადაცემის ხერხები. გამოყენებულია კონსტრუქციები პარამეტრების გაგზავნის დემონსტრირებისთვის. C#-ის მეთოდი აღიქვამს 4 სახის მოდიფიკატორს: *out*, *ref*, *params*, და *value*. ამ მოდიფიკატორების გამოყენების საილუსტრაციოდ შევქმენით კლასი *meTodebi*, რომელიც შეიცავს ერთ ველსა და ხუთ მეთოდს.

Main()-ში ვიძახებთ `procedura, Gamarjoba, Jami, Secvla` მეთოდებს. ვინაიდან `procedura` -ს `x` პარამეტრს არ აქვს მოდიფიკატორი, იგი განიხილება *value* მნიშვნელობის ფორმალურ პარამეტრად. მეთოდის აღწერაში გამოიყენება ფორმალური პარამეტრები,

ხოლო მეთოდის გამოძახებაში – ფაქტიური პარამეტრები. ფაქტიური პარამეტრებია: a, b, c,w. ფორმალური პარამეტრებია: x, y, z, w. ფაქტიური მნიშვნელობანი, პოზიციური შესაბამისობით, ენიჭება ფორმალურ არგუმენტებს (ეს რეალურად ხდება არგუმენტის ფაქტიური მნიშვნელობის გადაწერით *stack*-ში. *stack*-ი არის იმგვარად ორგანიზებული მეხსიერება, რომ მასში ჩაწერილი ბოლო ინფორმაცია იკითხება პირველი. ანალოგია ცეცხლსასროლი იარაღის მჭიდთან - ბოლოს ჩადებული ტყვია გაისვრის პირველი. მეთოდი იყენებს *stack*-ში ჩაწერილ მონაცემებს, ხოლო დასრულებისას ათავისუფლებს *stack*-ის მეხსიერებას). *value* მოდიფიკატორით მოცემული ფორმალური ცვლადები (პარამეტრები) ლოკალურია და მათი ნებისმიერი ცვლილება არ მოქმედებს ფაქტიურ ცვლადებზე, რომელთაც გამომძახებელი ფაქტობრივ არგუმენტებად იყენებს.

ამონაბეჭდი 5-2. მეთოდის პარამეტრები: meTodisparametrebi.cs

```
class meTodebi
{     int w=7;
// x პარამეტრის გაგზავნა მნიშვლელობით value (გაგზავნა, იგულისხმება), y  ref
// (გაგზავნა /მიღება), და z  out (მიღება)
        public  void procedura(int x, ref int y, out int z,int w)
        {// x=1,y=1,w=3,this.w=5
Console.WriteLine("x={0},y={1},w={2},this.w={3}",x,y,w,this.w);
                x++;  y++;z = 5;// x=2,y=2
        }
        // ღებულობს პარამეტრების ცვლად რაოდენობას
        public static int Jami(params int[] ricxvebi)
        {                int jami = 0;
                foreach (int i in ricxvebi)
                        jami += i;
                return jami;
        }
        /* ქვემოთ მოცემულია მეთოდების გადატვირთვის მაგალითი, ორი თანამოსახელე,
მაგრამ სხვადასხვა პარამეტრებიანი მეთოდის სახით. */
        public  void Gamarjoba(string saxeli, string TavsarTi)
        {                Console.WriteLine("mogesalmebiT, " + TavsarTi + " " + saxeli);
        }
        public  void Gamarjoba(string saxeli)
        { this.Gamarjoba(saxeli, "");
        }
        public static void Secvla(int[] arr)
```

```
        {arr[0]=888;  // ახდენს ორიგინალური ელემენტის ცვლილებას.
arr = new int[5] {-3, -1, -2, -3, -4};  // ცვლის მხოლოდ ლოკალურ მნიშვნელობას.
Console.WriteLine("meTodis SigniT pirveli elementis mniSvnelobaa: {0}",arr[0]);
        }
public static void SecvlamiT(ref int[] arr)
        {       arr[0]=888;  // ახდენს ორიგინალური ელემენტის ცვლილებას.
                arr = new int[5] {-3, -1, -2, -3, -4};  /* ცვლის ორიგინალური ელემენტების
მნიშვნელობებს და მეხსიერებაში მითითების მნიშვნელობას ანუ საწყის მისამართს */
Console.WriteLine("მეთოდის შიგნით პირველი ელემენტის მნიშვნელობაა: {0}",arr[0]);
        }}
class mTavari
{       public static void Main()
        {int a = 1, b = 1, w=3, c;  // c არ სჭირდება საწყისი მნიშვნელობების მინიჭება
                meTodebi meTodisnimuSi = new meTodebi ();
meTodisnimuSi.procedura (a, ref b, out c,w);
// a - გადაეცემა მნიშვნელობით, b – მითითებით, c - დაბრუნებით
                Console.WriteLine("{0} {1} {2}", a, b, c);  // 1 2 5
                int total = meTodebi.Jami(4, 3, 2, 1);   // abrunebs 10
                meTodisnimuSi.Gamarjoba("Tengiz", "batono");
//  მეთოდების არჩევა არგუმენტების მიხედვით
                meTodisnimuSi.Gamarjoba("xaTuna");
                int[] masivi = {1,4,5};
                Console.WriteLine("Main-ის შიგნით, მეთოდის გამოძახებამდე, პირველი
ელემენტია  : {0}", masivi [0]);
                meTodebi.Secvla(masivi);
                Console.WriteLine("Main-ის შიგნით, მეთოდის გამოძახების შემდეგ, პირველი
ელემენტია: {0}", masivi [0]);
        }
int[] masivimit = {1,4,5};
                Console.WriteLine("Main-is შიგნით, მეთოდის გამოძახებამდე, პირველი
ელემენტია : {0}", masivimit [0]);
                meTodebi.SecvlamiT(ref masivimit);
                Console.WriteLine("Main-ის შიგნით, მეთოდის გამოძახების შემდეგ, პირველი
ელემენტია: {0}", masivimit [0]);
}
```

პროგრამის გამოსავალია:

```
x=1,y=1,w=3,this.w=7
```

1 2 5

მოგესალმებით, ბატონო თენგიზ

მოგესალმებით, ხათუნა

Main-ის შიგნით, მეთოდის გამოძახებამდე, პირველი ელემენტია : 1

მეთოდის შიგნით პირველი ელემენტის მნიშვნელობაა: -3

Main-ის შიგნით, მეთოდის გამოძახების შემდეგ, პირველი ელემენტია: 888

Main-ის შიგნით, მეთოდის გამოძახებამდე, პირველი ელემენტია : 1

მეთოდის შიგნით პირველი ელემენტის მნიშვნელობაა: -3

Main-ის შიგნით, მეთოდის გამოძახების შემდეგ, პირველი ელემენტია: -3

procedura და *jami* მეთოდების გამოძახება განსხვავებულია *Main()*-ში გამოყენებული გამოძახებებისგან. *meTodisnimuSi* მითითების (*reference)* ნაცვლად ვიყენებთ სიტყვას meTodebi, ვინაიდან სტატიკური მეთოდებია და არ თხოულობენ ეგზემპლარის შექმნას. კლასის შიგნით მეთოდის გამოსაძახებლად ვიყენებთ საკვანძო სიტყვას this, მაგალითად: *(this.Gamarjoba(saxeli, ""));*. ეს საკვანძო სიტყვა ასევე შეიძლება გამოვიყენოთ კლასში განსაზღვრული და თანამოსახელე ფორმალური არგუმენტის გასარჩევად, რაც ნაჩვენებია procedura-ში w ცვლადის მაგალითზე. მეთოდი სახელად procedura იყენებს *ref* პარამეტრს. ეს გულისხმობს იმას, რომ მითითება პარამეტრზე (მისამართი) გადაეცემა მეთოდს. ეს მითითება იმავე ობიექტზე მიუთითებს ე. წ. *heap*-ში, რაზეც ფაქტობრივი არგუმენტი მეთოდის გამოძახებაში. *(heap*-ს უწოდებენ ობიექტების განსათავსებელ მეხსიერებას. აქვე შეგახსენებთ, რომ ობიექტი წარმოადგენს კლასის ეგზემპლარს (ნიმუშს). ერთ კლასს შეიძლება უამრავი ობიექტი შეესაბამებოდეს. *ref* იმას ნიშნავს, რომ ფორმალური არგუმენტის მნიშვნელობის ნებისმიერი ცვლილება იწვევს ფაქტიური არგუმენტის მნიშვნელობის შესაბამის ცვლილებას. მნიშვნელობის ცვლადების შემთხვევაში კოდი არ ცვლის მითითებას, მაგრამ იგი იწვევს მითითებული ობიექტის ცვლილებებს. მასივების გადაცემის შემთხვევაში მათი ელემენტების მნიშვნელობა ყოველთვის იცვლება, ვინაიდან მასივი თავად გახლავთ მითითების ტიპის. ეს ნაჩვენებია *Secvla* მეთოდის მაგალითზე. იმ შემთხვევაში, თუ გვსურს შევცვალოთ მითითების მნიშვნელობა (ანუ ფიზიკური მისამართი მეხსიერებაში), უნდა გამოვიყენოთ *ref* მოდიფიკატორი, რაც ჩანს *SecvlamiT* მეთოდის მაგალითზე.

საკვანძო სიტყვა *out* გამოიყენება ისეთი ფორმალური პარამეტრებისთვის, რომლებიც მეთოდის გამოძახებისას არაფერს ღებულობენ, ხოლო მეთოდის შესრულებისას მხოლოდ უკან აბრუნებენ მნიშვნელობებს. ეს უფრო ეფექტურია, ვინაიდან პროგრამა მეთოდში არ ახორციელებს დამატებით კოპირებას. ეს ლოკალურ კომპიუტერზე დიდ ეფექტს არ იძლევა. რა თქმა უნდა, ქსელში განაწილებული პროცესის შემთხვევაში, როცა ინფორმაციის გადაცემა ხდება საკომუნიკაციო გზებით *out* პარამეტრის გამოყენებას მეტი უპირატესობა აქვს.

მეთოდს *procedura* აქვს out პარამეტრი. *out* პარამეტრი მხოლოდ უკან აბრუნებს მნიშვნელობას გამომძახებელ ფუნქციაში. მეთოდის შესრულების დაწყებისას მისი მნიშვნელობა განუსაზღვრელია, შესაბამისად, მისი გამოყენება არ შეიძლება მანამ, სანამ არ მივანიჭებთ რაიმე მნიშვნელობას. მეთოდის შესრულების შედეგად *out* პარამეტრის მნიშვნელობა მიენიჭება შესაბამისი ფაქტობრივი არგუმენტის ცვლადს. *out* პარამეტრს აუცილებლად უნდა მივანიჭოთ მნიშვნელობა მეთოდიდან დაბრუნებამდე. მეთოდიდან დაბრუნება ხდება მისი ბოლო ოპერატორის შესრულებისას ან *return* ოპერატორის შესრულებისას. ეს უკანასკნელი გამოიყენება ისეთ მეთოდებში, რომლებიც აბრუნებენ მნიშვნელობებს.

მოცემულია მეთოდების გადატვირთვის მაგალითი, ორი თანამოსახელე მაგრამ სხვადასხვა პარამეტრებიანი მეთოდის სახით. თანამოსახელე მეთოდებიდან მეთოდის არჩევა ხდება არგუმენტების მიხედვით. ე. წ მეთოდის სიგნატურა: რაოდენობა, ტიპები და პოზიციური შესაბამისობა.

params პარამეტრი საშუალებას იძლევა, მეთოდი აღვწეროთ იმგვარად, რომ მან მიიღოს ცვლადი რაოდენობის არგუმენტები. ეს პარამეტრი უნდა იყოს ერთგანზომილებიანი ან უსწორმასწორო *(jagged)* მასივი. მეთოდი *Jami* ()-ის გამოძახებისას არგუმენტებად ვაგზავნით მძიმით გამოყოფილ პარამეტრებს. არგუმენტთა რაოდენობა სხვადასხვა მიმართვისას შეიძლება იყოს ცვლადი. თვით მეთოდის კოდში ვიყენებთ *foreach* ციკლს არგუმენტების ჯამის გამოსათვლელად. *params* პარამეტრი განიხილება მხოლოდ, როგორც შემავალი და მისი ნებისმიერი ცვლილება ლოკალურია.

ამონაბეჭდში 5-2 მოცემულია რამდენიმე მეთოდის მაგალითი, რომლებიც უკან აბრუნებენ მნიშვნელობას. რეკურსიული ფუნქციის სახელები მთავრდება ასო r-ით.

ამონაბეჭდი 5-2. არარეკურსიული და რეკურსიული მეთოდები. funqciebi.cs

```
using System;
namespace TB_moc1
{       class meTodebi
        {       // ფუნქციის განსაზღვრა
                public static double FactorialR(int n) // რეკურსიული
                {       if(n==0)
                        {return 1;}
                         else
                        {return n*FactorialR(n-1);}
                }
                public static double Factorial(int n)
                {       double F=1;
                        for(int i=2;i<=n;i++)
```

```
                { F=F*i;   }
                return F;
        }
        public static double FsumR(int n,double[] a) //  რეკურსიული
        {       if(n==0)
                {return Math.Sqrt(a[n]);}
                                        else
                {return Math.Sqrt(a[n-1]+FsumR(n-1,a));}
        }
        public static double Fsum(int n, double[] a)
        {       double S=0;
                for(int i=0;i<n;i++)
                {S=Math.Sqrt(S+a[i]);}
                                        return S;
        }}
        class mTavari
{       static void Main()
        {       int x=10; double Z;
                double [] a={0,1,2,3,4,5,6,7,8,9 };
                Z=meTodebi.FactorialR(x);
                Console.WriteLine("x={0},FactorialR={1}",x,Z);
                Z=meTodebi.Factorial(x);
Console.WriteLine("x={0},Factorial={1}",x,Z);
int xx=10;Console.WriteLine("x={0},FsumR(xx,a)={1}",xx,meTodebi.FsumR(xx,a));
            Console.WriteLine("x={0},Fsum(xx,a) ={1}",xx,meTodebi.Fsum(xx,a));
        }}}
```

პროგრამის გამოსავალი:

x=10,FactorialR=3628800

x=10,Factorial=3628800

x=10,FsumR(xx,a)=3.51282632199425

x=10,Fsum(xx,a) =3.51282632199425

ზემომოყვანილ მეთოდებში გამოითვლება რიცხვის ფაქტორიალი n! და ჯამი $\sqrt{a_n+\sqrt{a_{n-1}\sqrt{a_{n-2}+\ldots+\sqrt{a_1}}}}$ ჩვეულებრივი და რეკურსიული ხერხით. რეკურსიული არის ისეთი მეთოდი, რომელიც საკუთარ კოდში შეიცავს თავის თავზე მიმართვას.

თავი VI
სახელსივრცეები

ამ თავში განხილულია შემდეგი საკითხები:

- სახელსივრცის ცნება.
- *using* დირექტივის გამოყენება.
- *alias* დირექტივის გამოყენება.
- სახელსივრცის წევრები.

1-ელ თავში განვიხილეთ *using System* დირექტივა *MogesalmebaT* პროგრამაში. ეს დირექტივა *System* სახელსივრცის წევრების გამოყენების საშუალებას იძლევა.

სახელსივრცე არის C#-პროგრამის ელემენტი, რომელიც შექმნილია თქვენი პროგრამის ორგანიზებისთვის. იგი აგრეთვე უზრუნველყოფს სახელების კონფლიქტის თავიდან აცილებას. სახელსივრცის შემოტანა მოგვიანებით თავიდან აგვაცილებს სახელების კოლიზიას, თუ მოინდომებთ ამ კოდის ხელახალ გამოყენებას.

სახელსივრცეები არ შეესაბამება ფაილის ან დირექტორიის სახელებს., თუმცა მათ გასაგებად შეიძლება გამოიყენოთ ეს ანალოგია.

ამონაბეჭდი 6-1. C# სახელსივრცეები: saxelsivrceC_Sarpi.cs

```
// სახელსივრცის აღწერა
using System;
// სახელსივრცე zeda
namespace zeda
{   // პროგრამის საწყისი კლასი
   class ziriTadi
   {      // Main იწყებს პროგრამის შესრულებას.
      public static void Main()
      {         // კონსოლზე ბეჭდვა
         Console.WriteLine("ეს არის ახალი zeda სახელსივრცე.");
      }}}
```

ამონაბეჭდში 6-1 ნაჩვენებია, თუ როგორ შევქმნათ სახელსივრცე. ჩვენ აღვწერთ ახალ სახელსივრცეს სიტყვის *namespace* ჩასმით *zeda*-ს წინ. ფიგურულ ფრჩხილებში მოთავსებულია *zeda* სახელსივრცის წევრები.

ამონაბეჭდი 6-2. ერთმანეთში ჩაწყობილი სახელსივრცეები 1: Cawyobili dSaxelsivrce1.cs

```
using System;
namespace zeda
{   namespace qveda
```

```
{      class ziriTadi
   {         public static void Main()
      {            Console.WriteLine("ეს არის ახალი სასწავლო სახელსივრცე.");
      }
   }}}
```

სახელსივრცეები წარმოადგენს თქვენი კოდის (პროგრამის) სისტემატიზაციის საშუალებას. სახელსივრცეების ორგანიზება იერარქიული პრინციპით ხდება. უფრო ზოგად სახელებს ათავსებთ ზემოთ, ხოლო ნაკლებად ზოგადს – ქვემოთ (ხისებრი სრუქტურა). ეს იერარქიული სტრუქტურა შეიძლება წარმოვადგინოთ სახელსივრცეების ერთმანეთში მოთავსებით. ამონაბეჭდი 6-2. გვიჩვენებს, თუ როგორ წარმოვადგინოთ იერარქიული სახელსივრცეები.

ამონაბეჭდი 6-3. ერთმანეთში ჩაწყობილი სახელსივრცეები 2: CawyobiliSaxelsivrce2.cs

```
using System;
namespace zeda.qveda
{   class ziriTadi
   {      public static void Main()
      {         Console.WriteLine("ეს არის ახალი სასწავლო სახელსივრცე.");
   }}}
```

ამონაბეჭდი 6-3 გვიჩვენებს იერარქიული სახელსივრცეების წარმოდგენის სხვა გზას. იგი ახდენს იერარქიული სახელსივრცის აღწერას წერტილის გამოყენებით *zeda*-სა და *qveda*-ს შორის. შედეგი ზუსტად იგივეა, რაც ამონაბეჭდში 6-2. რა თქმა უნდა, ამონაბეჭდი 6-3 ჩასაწერად უფრო ადვილია.

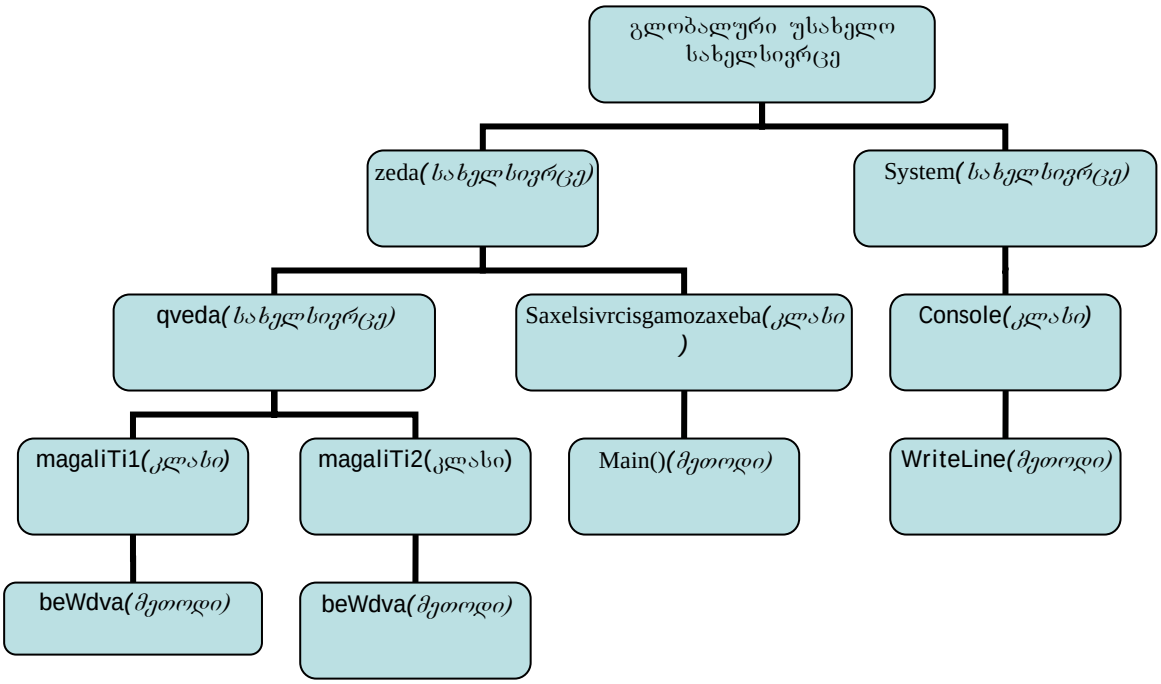
ამონაბეჭდი 6-4. სახელსივრცე წევრების გამოძახება: Saxelsivrcegamozaxeba.cs

```
// სახელსივრცის  დეკლარაცია
using System;
namespace zeda
{   // ჩაწყობილი სახელსივრცე
  namespace qveda
   {      class magaliTi1
      {         public static void beWdva()
         { Console.WriteLine("სხვა სახელსივრცის წევრის გამოძახების პირველი
მაგალითი..");
         }}}
   // პროგრამის საწყისი კლასი
   class Saxelsivrcisgamozaxeba
```

```
  {      public static void Main()
    {         // კონსოლზე ბეჭდვა
      qveda.magaliTi1.beWdva();
      qveda.magaliTi2.beWdva();
    }}}
// ზედა სახელსივრცის მსგავსი  ჩაწყობილი სახელსივრცე
namespace zeda.qveda
{   class magaliTi2
  {      public static void beWdva()
   { Console.WriteLine("სხვა სახელსივრცის წევრის გამოძახების მეორე მაგალითი.");
     }
  }
}
```

ამონაბეჭდი 6-4 გვიჩვენებს მაგალითს, თუ როგორ მივმართოთ სახელსივრცეს სრული სახელით. სრულად კვალიფიცირებული სახელი მოიცავს ყოველ ელემენტს, დაწყებული სახელსივრცის მთავარი სახელიდან და დამთავრებული მეთოდის სახელით. ზემომოყვანილ მაგალითში სახელსივრცე *zeda* მოიცავს სახელსივრცეს *qveda-ს* კლასით *magaliTi1* და მეთოდით *beWdva*. *Main()*-ი გამოიძახებს მეთოდს სახელით *qveda.magaliTi1. beWdva()* ვინაიდან *Main()*მეთოდი მოთავსებულია სახელსივრცეში *zeda*, კვალიფიცირებულ სახელში *zeda-ს* მითითება არ არის საჭირო. ე. ი. კვალიფიცირებულ სახელში მითითებული სახელსივრცის ზედა დონის ის სახელსივრცეც იგულისხმება, რომელსაც ეკუთვნის გამომძახებელი მეთოდი. პროგრამის ელემენტების იერარქია სქემატურად ხისებრი სტრუქტურით გამოხატულია ნახ. 6.1.

ნახ. 6.1.



ამონაბეჭდის 6-4 ბოლოს დამატებით გამოყენებულია სახელსივრცე *zeda.qveda.* კლასები *magaliTi1* და *magaliTi2* ორივე ეკუთვნის ერთსა და იმავე სახელსივრცეს. ისინი შესაძლებელია მოთავსებულ იქნეს სხვადასხვა ფაილებში და ეკუთვნოდეს ერთსა და იმავე სახელსივრცეს. *Main()*-ში *beWdva()*- მეთოდი გამოიძახება სახელით *qveda.magaliTi2.beWdva().* მიუხედავად იმისა, რომ *magaliTi2.beWdva()* განსაზღვრულია იმ სახელსივრცის ფიგურულ ფრჩხილებს გარეთ, რომელშიც განსაზღვრულია *Main()*-მეთოდი, არ არის საჭირო, რომ სახელსივრცე *zeda* იყოს სრულად კვალიფიცირებული სახელის ნაწილი. ეს იმიტომ ხდება, რომ *Saxelsivrcisgamozaxeba* კლასი, *Main()* მეთოდი და *magaliTi2* კლასი ეკუთვნის სახელსივრცეს *zeda.* შევნიშნოთ, რომ სავალდებულოა მხოლოდ სრულად კვალიფიცირებული სახელის უნიკალურობა. ვინაიდან ერთი და იგივე სახელი *beWdva()* სხვადასხვა კლასებს ეკუთვნის, ამ შემთხვევაში, ცალსახობა დაცულია და პირველი და მეორე გამოძახებისას სხვადასხვა მეთოდები სრულდება ანუ zeda.qveda.magaliTi1.beWdva(); და zeda.qveda.magaliTi2.beWdva(); მიუხედავად აშკარა მსგავსებისა, მაინც განსხვავებული სახელებია.

ამონაბეჭდი 6-5. დირექტივების გამოყენება: direqtivebisgamoyeneba.cs

```
// სახელსივრცის  დეკლარაცია
using System;
using zeda.qveda;
  // პროგრამის საწყისი კლასი
class Direqtivis gamoyeneba
{     public static void Main()
  {      // სახელსივრცის წევრის გამოძახება
     magaliTi.bewdva();
  }
}
// სახელსივრცე
namespace zeda.qveda
{   class magaliTi
  {      public static void bewdva()
     {          Console.WriteLine("using დირექტივის გამოყენების მაგალითი.");
     }
  }
}
```

თუ გსურთ გამოიძახოთ მეთოდი სრულად კვალიფიცირებული სახელის გარეშე, შეგიძლიათ გამოიყენოთ *using* დირექტივა. ამონაბეჭდში 6-5 ნაჩვენებია ორი *using*

დირექტივა. პირველია *using System*, მსგავსი დირექტივა ნანახი გაქვთ ამ სახელმძღვანელოს ყოველ ამონაბეჭდში. ის საშუალებას იძლევა, დავბეჭდოთ *System* სახელსივრცის მეთოდების სახელები სიტყვა *System*-ის გარეშე. *Console* არის კლასი, რომელიც *System* სახელსივრცის წევრია, მეთოდით *WriteLine()*. სრულად კვალიფიცირებული სახელია *System.Console.WriteLine(...)*.

ანალოგიურად, დირექტივა *using zeda.qveda* გამოვიყენოთ *zeda.qveda* სახელსივრცის წევრები სრულად კვალიფიცირებული სახელების გარეშე. ამის გამო ყოველთვის, როცა გვინდა გამოვიყენოთ *bewdva()* მეთოდი, ვბეჭდავთ *magaliTi.bewdva()*-ს *zeda.qveda.magaliTi.bewdva()*-ის მაგივრად.

სახელის იდენტიფიცირება ხდება ჯერ იმ სახელსივრცეში, სადაც ის მოთავსებულია, ხოლო წარუმატებლობის შემთხვევაში *using* დირექტივებით განსაზღვრულ სახელსივრცეებში.

ამონაბეჭდი 6-6. Alias დირექტივა: Aliasdirectiva.cs

```
using System;
using csTut = zeda.qveda.magaliTi; // ალიას –შემოკლებული სახელი
class AliasDirective
{   public static void Main()
    {           // სახელსივრცის წევრის გამოძახება
        csTut.bewdva();
        bewdva();
    }
    //პოტენციურად არაცალსახა მეთოდი
    static void bewdva()
    {       Console.WriteLine("არ არის წევრი zeda.qveda.magaliTi.");
    }
}

// სახელსივრცე
namespace zeda.qveda
{   class magaliTi
    {       public static void bewdva()
        {           Console.WriteLine("ეს არის წევრი zeda.qveda.magaliTi.");
        }
    }
}
```

ზოგჯერ შეგხვდებათ გრძელი სახელსივრცე და მოისურვებთ, შემოკლებულად მიმართოთ მას. ეს გააუმჯობესებს პროგრამის კითხვადობას და თავიდან აგაცილებთ სახელების კონფლიქტს. ამონაბეჭდი 6-6 გვიჩვენებს, თუ როგორ შევქმნათ ფსევდონიმი შემდეგი დირექტივით *using csTut = zeda.qveda.magaliTi.* ახლა გამოსახულება *csTut* შესაძლებელია გამოვიყენოთ ყველგან *zeda.qveda.magaliTi* ადგილას. ჩვენ ამას ვიყენებთ *Main()*-ში. *Main()*-ი იძახებს *bewdva()* მეთოდს *AliasDirectiv* კლასიდან. ეს იმავე სახელის მეთოდია, რაც *bewdva()* მეთოდი *magaliTi* კლასში. მიზეზი იმისა, რომ ორივე მეთოდი შეიძლება გამოძახებულ იქნეს ერთი და იმავე სახელით, არის ის, რომ *bewdva()* მეთოდი *magaliTi* კლასში კვალიფიცირებულია *csTut* ფსევდონიმით. როგორც მაგალითიდან ჩანს, ფსევდონიმის განსაზღვრისას კლასის სახელიც შეიძლება იქნეს გამოყენებული. ეს საშუალებას აძლევს კომპილატორს, ზუსტად განსაზღვროს, რომელი მეთოდი შეასრულოს.

შეცდომით რომ გამოგვეტოვებინა *csTut* მეთოდის გამოძახებაში, მაშინ კომპილატორი ორჯერ გამოიძახებდა *bewdva()* მეთოდს *AliasDirective* კლასიდან.

აქამდე ჩვენს სახელსივრცეებში მხოლოდ კლასები იყო მოთავსებული. რა თქმა უნდა, სახელსივრცეები სხვა ტიპებსაც მოიცავენ, როგორც ქვემოთ ცხრილშია ნაჩვენები:

Classes	Enumerations
Structures	Delegates
Interfaces	

თავი VII

შესავალი კლასებში

განხილულია შემდეგი საკითხები:

- კონსტრუქტორების გამოყენება.
- განსხვავება სტატიკურ და ეგზემპლარის წევრებს შორის.
- დესტრუქტორები.
- კლასის წევრები.

ამ სახელმძღვანელოს დასაწყისიდან ვიყენებთ ტერმინს "კლასები". ახლა უფრო დაწვრილებით განვიხილავთ, თუ რას წარმოადგენს კლასი და როგორ შევქმნათ ის. კლასის აღწერა იწყება საკვანძო სიტყვით *class*, რომელსაც მოსდევს კლასის სახელი და ფიგურულ ფრჩხილებში მოთავსებული კლასის წევრების სიმრავლე. ყოველ კლასს გააჩნია (ცხადი ან არაცხადი) კონსტრუქტორი (კლასის სახელის მქონე მეთოდ(ებ)ი), რომელიც ყოველთვის გამოიძახება, როცა იქმნება კლასის ეგზემპლარი. კონსტრუქტორების დანიშნულებაა კლასის წევრების ინიციალიზაცია (საწყისი მნიშვნელობების მინიჭება) კლასის ეგზემპლარის შექნისას.

კონსტრუქტორებს არ გააჩნიათ დასაბრუნებელი მნიშვნელობები და ყოველთვის აქვთ იგივე სახელი, რაც კლასს.

ამონაბეჭდში 7-1 მოცემულია კლასის მაგალითი.

ამონაბეჭდი 7-1. კლასების მაგალითი : Klasebi.cs

```
// სახელსივრცის დეკლარაცია
using System;
// დამხმარე კლასი
class Gamomyvani
{    string striqoni;
    // კონსტრუქტორი
    public Gamomyvani(string Semavali)
    {       striqoni = Semavali;    }
 // ეგზემპლარის მეთოდი
   public void striqonisbeWdva()
    {       Console.WriteLine("{0}", striqoni);    }

   //   დესტრუქტორი
    ~Gamomyvani()
    { // ზოგიერთი რესურსის განთავისუფლება
    }}
// პროგრამის საწყისი კლასი
class MagaliTi
{    // Main იწყებს პროგრამის შესრულებას
 public static void Main()
    {       // Gamomyvani-ის ეგზემპლარის შექმნა
```

```
Gamomyvani outCl = new Gamomyvani("ეს იბეჭდება gamomyvani კლასის მიერ .");
    // Gamomyvani კლასის მეთოდის გამოძახება
    outCl.striqonisbeWdva();
}}
```

ამონაბეჭდი 7-1 გვიჩვენებს ორ კლასს. ზედა კლასს აქვს კონსტრუქტორი, ეგზემპლარის მეთოდი (არასტატიკური) და დესტრუქტორი. მას ასევე გააჩნია ველი სახელად *striqoni*. ჩვენს შემთხვევაში *Gamomyvani* კონსტრუქტორი ღებულობს *string* ტიპის არგუმენტს *Semavali*. ეს სტრიქონი მიენიჭება კლასის ველს *striqoni*. არ არის სავალდებულო, კონსტრუქტორი ცხადად აღვწეროთ. იმ შემთხვევაში, თუ კლასში კონსრუქტორი არ არის, გამოიძახება წინასწარ განსაზღვრული სტანდარტული კონსტრუქტორი. იგი უნდა წარმოვიდგინოთ, როგორც უბრალო კონსტრუქტორი არგუმენტების გარეშე. იგი მიანიჭებს სტანდარტულ საწყის მნიშვნელობებს ველებს (მაგალითად, 0-ს). კონსტრუქტორი შეიძლება გამოვიყენოთ საკუთარ ინიციალიზერთანაც (საწყისი მნიშვნელობების მიმნიჭებელი კონსტრუქტორი). ქვემოთ მოყვანილია მაგალითი:

```
public Gamomyvani() : this("თანდაყოლილი სტრიქონი კოსტრუქტორისთვის") { }
```

წარმოვიდგინოთ, რომ მოცემული კონსტრუქტორი მოთავსებულია კლასში *Gamomyvani* ამონაბეჭდში 7-1. ამ კონსტრუქტორს მოსდევს ინიციალიზერი. ნიშანი ":" აღნიშნავს ინიციალიზერის დასაწყისს, რომელსაც მოსდევს *this* საკვანძო სიტყვა. საკვანძო სიტყვა *this* მიუთითებს კლასის ეგზემპლარის კონკრეტულ ობიექტზე. ამ შემთხვევაში გამოიძახება კონსტრუქტორი, რომელიც განსაზღვრულია იმავე ობიექტში. საკვანძო სიტვას *this* მოსდევს ფრჩხილებში მოთავსებული სტრიქონი. ფაქტიურად ამის შესრულების შედეგად ხდება *Gamomyvani* კლასის კონსტრუქტორის გამოძახება, ვინაიდან მისი ფორმალური პარამეტრი იმავე ტიპისაა, რაც *this* გამოძახებაში. ინიციალიზაცია იმის გარანტიაა, რომ კლასის ეგზემპლარის შექმნისას კლასის ველებს ექნებათ კლასის ავტორის მიერ განსაზღვრული მნიშვნელობები. კლასის ეგზემპლარის შექმნა ამ შემთხვევაში უნდა ხდებოდეს შემდეგნაირად: *Gamomyvani outCl = new Gamomyvani();*

ზედა მაგალითი გვიჩვენებს თუ, როგორ შეიძლება გააჩნდეს კლასს მრავალი კონსტრუქტორი. განსაზღვრული კლასის კონსტრუქტორის გამოძახება დამოკიდებულია პარამეტრების რაოდენობაზე და ყოველი პარამეტრის ტიპზე.

არსებობს ორი ტიპის კლასის წევრები: ეგზემპლარის და სტატიკური (*static*). ეგზემპლარის წევრები ეკუთვნის კონკრეტულ ობიექტს, რომელიც იქმნება კლასის საფუძველზე და რომელსაც ვუწოდებთ კლასის ეგზემპლარს. *MagaliTi*-ში *Main()* მეთოდი ქმნის *Gamomyvani*-ის ეგზემპლარს სახელად *outCl*. თქვენ შეგიძლიათ შექმნათ *Gamomyvani* კლასის უამრავი ეგზემლარი სხვადასხვა სახელებით. თითოეული ეს ეგზემპლარი ცალკე, დამოუკიდებლად არსებულია. მაგალითად, თუ შექმნით *Gamomyvani* კლასის ორ ეგზემპლარს, როგორც ნაჩვენებია ქვემოთ:

```
Gamomyvani oc1 = new Gamomyvani("Gamomyvani1");
Gamomyvani oc2 = new Gamomyvani("Gamomyvani2");
```

ჩვენ შევქმენით *Gamomyvani*-ს ორი სხვადასხვა ეგზემპლარი საკუთარი *striqoni* ველებითა და საკუთარი *striqonisbeWdva()* მეთოდებით. სტატიკური წევრები ეგზემპლარის შექმნის დროს არ მონაწილეობენ. თუ კლასი არის *static*, შეგიძლიათ მიმართოთ მას შემდეგი სინტაქსის გამოყენებით *<კლასის სახელი>.<სტატიკური წევრის სახელი>*. ეგზემპლარების სახელებია *oc1* და *oc2*.

დავუშვათ *Gamomyvani* აქვს შემდეგი *static* მეთოდი:

```
public static void mbeWdavi()
{       Console.WriteLine("ეს არის ერთ-ერთი ჩემთაგანი.");    }
```

Main()-დან იგი უნდა გამოიძახოთ შემდეგნაირად:

```
Gamomyvani.mbeWdavi();
```

კლასის სტატიკური წევრები უნდა იქნეს გამოძახებული კლასის სახელით და არა მათი ეგზემპლარის სახელით. ეს იმას ნიშნავს, რომ არ არის საჭირო კლასის ეგზემპლარის შექმნა მისი სტატიკური წევრების გამოსაყენებლად. ყოველთვის არსებობს სტატიკური წევრების მხოლოდ ერთი ასლი. სტატიკური წევრების საუკეთესო გამოყენებაა, როცა არ მოითხოვება შუალედური მდგომარეობა, მაგალითად, როგორიცაა მათემატიკური ფუნქციები. NET Frameworks კლასების ძირითადი ბიბლიოთეკა *mscorlib* შეიცავს *Math* კლასს და ინტენსიურად იყენებს სტატიკურ წევრებს.

კონსტრუქტორების შემდეგი ტიპია *static* კონსტრუქტორი. იგი გამოიყენება კლასში სტატიკური ველების ინიციალიზაციისთვის. *static* კონსტრუქტორის აღწერა ხდება საკვანძო სიტყვის *static* საშუალებით კონსტრუქტორის სახელის წინ. *static* კონსტრუქტორის გამოძახება ხდება მხოლოდ ერთხელ და ყველა სხვა კონსტრუქტორზე ადრე.

კლასს *Gamomyvani* აგრეთვე აქვს დესტრუქტორი. დესტრუქტორი გამოიყურება კონსტრუქტორის მსგავსად. განსხვავება მხოლოდ იმაშია, რომ წინ აქვს ტილდის ნიშანი "~" . იგი არ ღებულობს პარამეტრებს და არც უკან აბრუნებს მნიშვნელობას. დესტრუქტორი გამოიყენება იმ რესურსების გამოსათავისუფლებლად, რომელსაც კლასი იყენებს მისი არსებობის განმავლობაში. ჩვეულებრივ იგი გამოიძახება, როდესაც C#-ის გამოყენებული და უკვე არასაჭირო მეხსიერების კოლექტორი (garbage collector) გადაწყვეტს კლასის მიერ დაკავებული მეხსიერების რესურსის გამოთავისუფლებას.

აქამდე განვიხილეთ მხოლოდ კლასის წევრები, როგორიცაა ველები, მეთოდები, კოსტრუქტორები და დესტრუქტორები. ქვემოთ მოყვანილია კლასის წევრთა შესაძლო ტიპების სრული ნუსხა: Constructors, Fields, Destructors, Methods, Properties, Indexers, Delegates, Events , Nested Classes. ისინი განხილულია მომდევნო თავებში.

თავი VIII

კლასის მემკვიდრეობითობა

ამ თავში განხილულია შემდეგი საკითხები:

- ძირითადი კლასი;
- წარმოებული კლასი;
- წარმოებული კლასიდან ძირითადი კლასის ინიციალიზაცია;
- ძირითადი კლასის წევრების გამოძახება;
- ძირითადი კლასის წევრების დაფარვა.

მემკვიდრეობითობა არის ობიექტურად ორიენტირებული პროგრამირების კონცეფციის ერთ-ერთი მთავარი ნაწილი. იგი საშუალებას იძლევა, განმეორებით გამოვიყენოთ არსებული კოდი. კოდის განმეორებით გამოყენება ზოგავს პროგრამირების დროს და ზრდის მის ხარისხსა და საიმედოობას.

ამონაბეჭდი 8-1. მემკვიდრეობითობა: sabazoklasi.cs

```
using System;
public class MSobeliklasi
{   public MSobeliklasi()
    {       Console.WriteLine("მშობელი კონსტრუქტორი.");
    }
    public void bewdva()
    {       Console.WriteLine("მე ვარ მშობელი კლასი.");
    }
}
public class Svilobili : MSobeliklasi
{   public Svilobili()//კონსტრუქტორი
    {       Console.WriteLine("შვილობილი კონსტრუქტორი.");
    }
    public static void Main()
    {       Svilobili Svili = new Svilobili();
        Svili.bewdva();
    }
}
```

გამოსავალი:

მშობელი კონსტრუქტორი.

შვილობილი კონსტრუქტორი.

მე ვარ მშობელი კლასი.

ამონაბეჭდი 8-1 წარმოგვიდგენს ორ კლასს. ზედა კლასს ეწოდება *MSobeliklasi*, ხოლო წარმოებულ კლასს - *Svilobili*. ამ უკანასკნელის შექმნისას გვსურს, რომ მან შეძლოს *MSobeliklasi*-ში არსებული კოდის გამოყენება.

პირველ რიგში უნდა აღვწეროთ *MSobeliklasi*, როგორც *Svilobili*-ის ძირითადი (საბაზო კლასი). ეს ხორციელდება *Svilobili*-ის აღწერაში შემდეგნაირად: class *Svilobili : MSobeliklasi*. საბაზო კლასის აღწერა ხდება ორი წერტილის დამატებით წარმოებული კლასის სახელის შემდეგ, რასაც მოსდევს ძირითადი კლასის სახელი.

C#-ი უზრუნველყოფს მხოლოდ მხოლობით მემკვიდრეობითობას (არა მრავლობითს, ხისებრი მიმართება). ამიტომ თითოეული წარმოებული კლასისთვის შეგვიძლია განვსაზღვროთ მხოლოდ ერთი საბაზო კლასი. თუმცა C#-ი ინტერფეისების (*interface)* მრავლობითი მემკვიდრეობითობის (გრაფისებრი მიმართება) საშუალებას იძლევა. *Svilobili* კლასს აქვს ზუსტად იგივე შესაძლებლობები, რაც *MSobeliklasi* კლასს. უფრო მეტიც, შეიძლება ითქვას, რომ *Svilobili* არის *MSobeliklasi*-ი. ეს ნაჩვენებია *Svilobili*-ის *Main()* მეთოდში, როცა გამოიძახება *bewdva()* მეთოდი. *Svilobili*-ს არ გააჩნია საკუთარი *bewdva()* მეთოდი. ის იყენებს *MSobeliklasi*-ის *bewdva()* მეთოდს. შედეგები შეგიძლიათ იხილოთ პროგრამის გამოსავალის მესამე სტრიქონში.

საბაზო კლასის ეგზემპლარი ავტომატურად იქმნება წარმოებული კლასის ეგზემპლარამდე. შევნიშნოთ, რომ ამონაბეჭდის 8-1 გამოსავალში ჩანს, რომ *MSobeliklasi*-ის კონსტრუქტორი სრულდება *Svilobili*-ის კონსტრუქტორის წინ.

ამონაბეჭდი 8-2. წარმოებული კლასის კომუნიკაცია საბაზო კლასთან:

sabazosTansaubari.cs

```
using System;
 public class MSobeli
{   string mSobelistriqoni;
   public MSobeli()
   {       Console.WriteLine("მშობელი კონსტრუქტორი.");
   }
   public MSobeli(string striqoni)
   {       mSobelistriqoni = striqoni;
      Console.WriteLine(mSobelistriqoni);
   }
   public void bewdva()
   {       Console.WriteLine("მე ვარ მშობელი კლასი.");
   }
}
```

```
public class Svili : MSobeli
{   public Svili() : base("წარმოებულიდან")
    {       Console.WriteLine("შვილობილი კონსტრუქტორი.");
    }
    public new void bewdva()
    {       base.bewdva();
        Console.WriteLine("მე ვარ შვილი კლასი.");
    }
    public static void Main()
    {       Svili Svili = new Svili();
        Svili.bewdva();
        ((MSobeli)Svili).bewdva();
    }
}
```

პროგრამის გამოსავალი:

```
წარმოებულიდან
შვილობილი კონსტრუქტორი.
მე ვარ მშობელი კლასი.
მე ვარ შვილი კლასი.
მე ვარ მშობელი კლასი.
```

წარმოებულ კლასს (*Svili*) შეუძლია ურთიერთობა საბაზო კლასთან (*MSobeli*) ეგზემპლარის შექმნის პროცესში. ამონაბეჭდში 8-2 ნაჩვენებია, თუ როგორ ხდება ეს *Svili*-ის კონსტრუქტორის აღწერისას. ორი წერტილი ":" და საკვანძო სიტყვა *base* იძახებს საბაზო კლასის (*MSobeli*) კონსტრუქტორს შესაბამისი პარამეტრების სიით. გამოსავალის პირველი სტრიქონი წარმოგვიდგენს საბაზო კლასის (*MSobeli*)კონსტრუქტორს, რომელიც გამოიძახება სტრიქონით "წარმოებულიდან ".

ხანდახან შეიძლება დაგვჭირდეს საბაზო კლასში (*MSobeli*) არსებული მეთოდის შეცვლა საკუთარი მეთოდით. კლასი *Svili* ახდენს საკუთარი *bewdva()* მეთოდის აღწერას. წარმოებული კლასის *bewdva()* მეთოდი ფარავს საბაზო კლასის *bewdva()* მეთოდს, რის შედეგადაც საბაზო კლასის *bewdva()* მეთოდი არ გამოიძახება, თუ არ მივმართეთ სხვა სპეციალურ საშუალებას.

წარმოებული კლასის *bewdva()* მეთოდში შეგვიძლია ღიად გამოვიძახოთ საბაზო კლასის *bewdva()* მეთოდი. ეს ხორციელდება *"base"* პრეფიქსის გამოყენებით. საკვანძო სიტყვა *"base"* საშუალებას იძლევა, გამოვიყენოთ საბაზო კლასის *public* (საჯარო) ან *protected* (დაცული) წევრები. წარმოებული კლასის *bewdva()* მეთოდის შედეგი მოყვანილია პროგრამის გამოსავალში, მე-3 და მე-4 სტრიქონებში.

საბაზო კლასის (*MSobeli)* წევრთა მიმართვა შეიძლება დიად. ეს ხორციელდება *Svili* class *Main()* მეთოდის ბოლო ოპერატორის საშუალებით. შეგახსენებთ, რომ წარმოებული კლასი წარმოადგენს საბაზო კლასის სპეციალიზაციას (დაკონკრეტებას, დაზუსტებას, გაფართოებას). ეს ფაქტი საშუალებას იძლევა, მოვახდინოთ წარმოებული კლასის ტიპის გარდაქმნა ანუ გავხადოთ იგი საბაზო კლასის ეგზემპლარად. ამონაბეჭდის 8-2 ბოლო სტრიქონი გვიჩვენებს, რომ *MSobeli* კლასის *bewdva()* მეთოდი ნამდვილად შესრულდება.

Svili კლასის *bewdva()* მეთოდის მოდიფიკატორი *new* საშუალებას იძლევა, დავფაროთ *MSobeli* კლასის *bewdva()* მეთოდი. ეს აშკარად აგვაცილებს თავიდან ე. წ. პოლიმორფიზმს (ერთი და იმავე სახელით სხვადასხვა მეთოდის შესრულება). *new* მოდიფიკატორის გარეშე კომპილატორი უბრალოდ გამოიტანდა შემჩნევის შეტყობინებას.

თავი IX
პოლიმორფიზმი

ამ თავში განხილულია შემდეგი საკითხები:

- პოლიმორფიზმის ცნება;
- ვირტუალური მეთოდის შექმნა;
- ვირტუალური მეთოდის ჩანაცვლება;
- პოლიმორფიზმის გამოყენება პროგრამაში;
- პროგრამის სრუქტურა და დამუშავების პროცესი.

პოლიმორფიზმში იგულისხმება თანამოსახელე მეთოდის მიერ სხვადასხვა კოდის შესრულება. მაგალითად, "+" აღნიშნავს რიცხვების შეკრებასაც და სტრიქონების კონკატენაციასაც. პოლიმორფიზმის განხორციელება შესაძლებელია მეთოდების გადატვირთვის, ინტერფეისების და მემკვიდრეობითობის გამოყენებით. ეს უკანასკნელი საშუალებას გვაძლევს, დინამიურად (პროგრამის შესრულებისას) გამოვიყენოთ წარმოებული კლასის მეთოდები საბაზო კლასზე მითითების გზით. ეს ძალზე მოხერხებულია, როცა ვანიჭებთ ობიექტების ჯგუფს მასივს, ხოლო შემდგომ თითოეულისთვის ვიძახებთ შესაბამის მეთოდს. არ არის აუცილებელი, რომ მათ ჰქონდეთ ერთი და იგივე ტიპი. თუმცა, თუ ისინი დაკავშირებულნი არიან მემკვიდრეობითობით, შეგიძლიათ დაამატოთ ისინი მასივს, როგორც მემკვიდრეობითი ტიპი. თუ მათ ყველას მეთოდის საზიარო სახელი აქვთ, ყოველი ობიექტის შესაბამისი მეთოდი გამოიძახება. ამ თავში მოთხრობილია, თუ როგორ ხორციელდება ეს.

ამონაბეჭდი 9-1. საბაზო კლასი ვირტუალური მეთოდით.: Daxazeobieqti.cs

```
using System;
public class Daxazeobieqti
{   public virtual void Xazva()
    {       Console.WriteLine("მე მხოლოდ ხაზვადი საწყისი ობიექტი ვარ..");
    }
}
```

ამონაბეჭდი 9-1 გვიჩვენებს *Daxazeobieqti* კლასს. იგი არის საბაზო კლასი დანარჩენი კლასებისთვის. მას აქვს ერთადერთი მეთოდი სახელად *Xazva()*. *Xazva()* მეთოდს აქვს მოდიფიკატორი *virtual*. მოდიფიკატორი *virtual* გვიჩვენებს, რომ წარმოებულ კლასში შეიძლება მოხდეს ამ მეთოდის ჩანაცვლება. *Daxazeobieqti* კლასის *Xazva()* მეთოდი ახდენს მხოლოდ ერთადერთ ქმედებას – კონსოლზე ბეჭდავს წინადადებას "მე მხოლოდ ხაზვადი საწყისი ობიექტი ვარ.".

ამონაბეჭდი 9-2. წარმოებული კლასები ჩამნაცვლებელი მეთოდებით: Wrfe.cs, Wre.cs, და Kvadrati.cs

```
using System;
public class Wrfe : Daxazeobieqti
{   public override void Xazva()
    {       Console.WriteLine("მე ვარ წრფე.");
    }
}
public class Wre : Daxazeobieqti
{   public override void Xazva()
    {       Console.WriteLine("მე ვარ წრე.");
    }
}
public class Kvadrati : Daxazeobieqti
{   public override void Xazva()
    {       Console.WriteLine("მე ვარ კვადრატი.");
    }
}
```

ამონაბეჭდზე 9-2 ნაჩვენებია სამი კლასი. ეს კლასები წარმოებულია (მემკვიდრეობას იღებს) *Daxazeobieqti* კლასიდან. ყოველ კლასს აქვს *Xazva()* მეთოდი და ყოველ *Xazva()* მეთოდს აქვს *override* მოდიფიკატორი. *override* მოდიფიკატორი ნებართვას იძლევა, რომ მისი შესაბამისი საბაზო კლასის *virtual* მეთოდი ჩანაცვლებულ იქნეს *override* მოდიფიკატორიანი მეთოდით პროგრამის შესრულების დროს. ჩანაცვლება (*override*) მოხდება მხოლოდ იმ შემთხვევაში, თუ მითითება ხდება საბაზო კლასის საშუალებით. ჩამნაცვლებელ მეთოდებს უნდა ჰქონდეთ ერთი და იგივე სიგნატურა (პარამეტრები და ტიპები), სახელი, პარამეტრების რაოდენობა, როგორც საბაზო კლასის ჩასანაცვლებელ *virtual* მეთოდს.

ამონაბეჭდი 9-3. პოლიმორფიზმის განხორციელება: XazvaDemo.cs

```
using System;
public class XazvaDemo
{   public static int Main( )
    {       Daxazeobieqti[] dObj = new Daxazeobieqti[4];
      dObj[0] = new Wrfe();
      dObj[1] = new Wre();
```

```
dObj[2] = new Kvadrati();
dObj[3] = new Daxazeobieqti();
foreach (Daxazeobieqti xazvaObj in dObj)
{        xazvaObj.Xazva();      }
return 0;   }}
```

ამონაბეჭდი 9-3 წარმოგვიდგენს პროგრამას, რომელიც იყენებს ამონაბეჭდით 9-1 და ამონაბეჭდით 9-2 განსაზღვრულ კლასებს. ეს პროგრამა განახორციელებს პოლიმორფიზმს. *XazvaDemo* კლასის *Main()* მეთოდში იქმნება მასივი. ამ მასივის ტიპი არის *Daxazeobieqti* კლასი. მასივი *dObj* ინიციალიზაციისას შეიცავს *Daxazeobieqti* ტიპის ოთხ ობიექტს.

ვინაიდან ისინი მემკვიდრეობით დაკავშირებულნი არიან *Daxazeobieqti* საბაზო კლასთან, კლასები *Wrfe, Wre* და *Kvadrati* შეიძლება მივანიჭოთ *dObj* მასივს. ამ საშუალების გარეშე ვერ შევძლებდით განსხვავებული ტიპებისგან შედგენილი მასივის შექმნას. მემკვიდრეობითობა საშუალებას გვაძლევს, შევქმნათ საბაზო კლასის მსგავსი ობიექტები.

მასივის შექმნის შემდეგ *foreach* ციკლი გაივლის მასივის ყოველ ელემენტს. ამ ციკლში *Xazva()* მეთოდი გამოიძახება *dObj* მასივის ყოველი ელემენტისთვის. ვინაიდან ადგილი აქვს პოლიმორფიზმს, ყოველი შესრულების დროის მომენტში გააქტიურდება არსებული ტიპის ობიექტი. მითითების ტიპი *dObj* მასივისთვის არის *Daxazeobieqti*. თუმცა წარმოებული კლასი ყოველ მომენტში ჩაანაცვლებს *Daxazeobieqti* კლასის *virtual Xazva()* მეთოდს. ეს იწვევს ჩანაცვლებული *Xazva()* მეთოდის შესრულებას, როდესაც ვიყენებთ მითითებას *Daxazeobieqti* კლასზე *dObj* მასივის საშუალებით. ქვემოთ მოყვანილია პროგრამის გამოსავალი:

მე ვარ წრფე

მე ვარ წრე.

მე ვარ კვადრატი.

მე მხოლოდ ხაზვადი საწყისი ობიექტი ვარ.

override მოდიფიკატორით *Xazva()* მეთოდი სრულდება ყოველი წარმოებული კლასისთვის, როგორც ნაჩვენებია *XazvaDemo* პროგრამაში. ბოლო სტრიქონს ბეჭდავს *Daxazeobieqti* კლასის *virtual Xazva()* მეთოდი. ეს იმიტომ ხდება, რომ შესრულების დროის ტიპი მასივის ოთხივე ელემენტისთვის არის *Daxazeobieqti* ობიექტი.

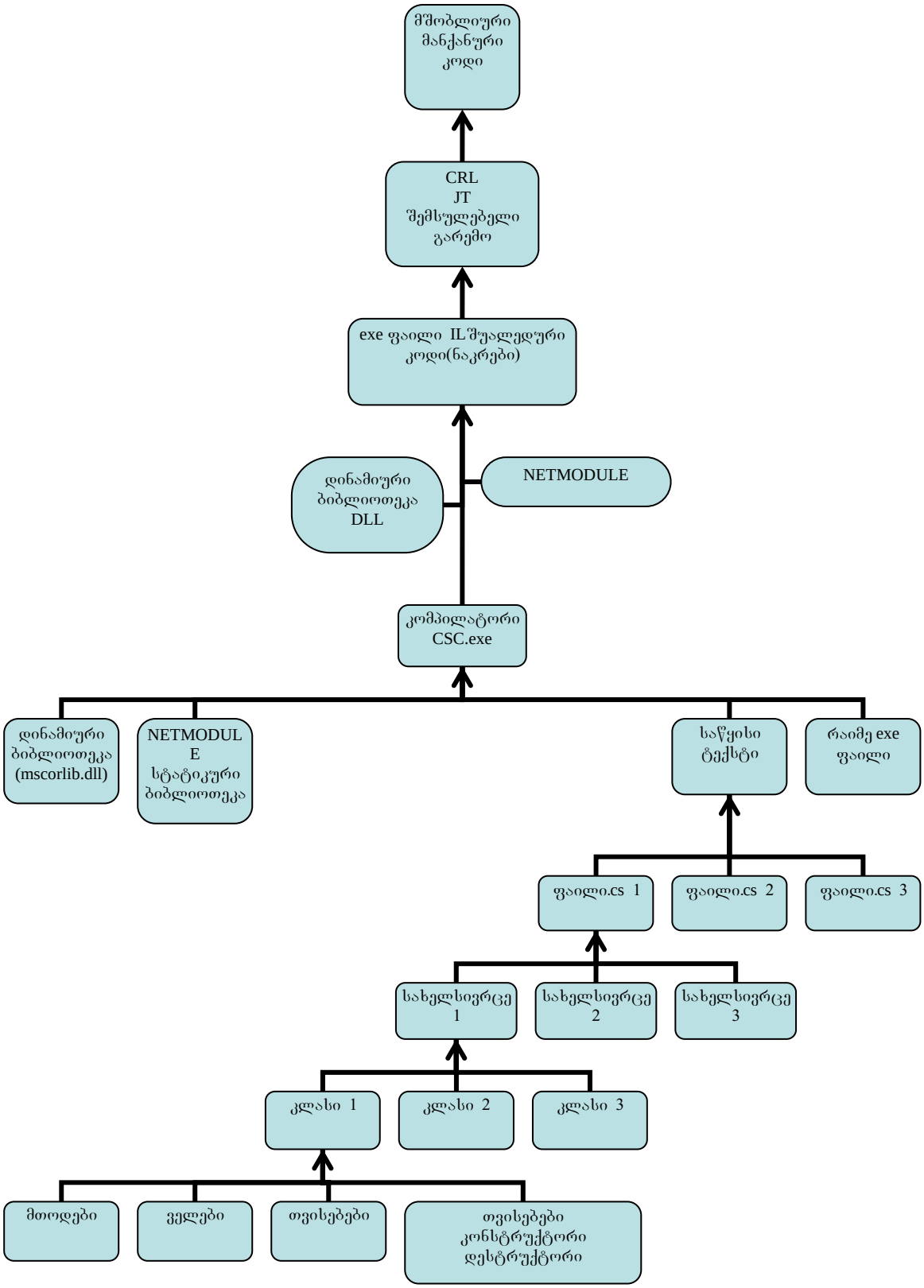
ამ თავში მოყვანილი მაგალითის კომპილირება შეიძლება შემდეგი სტრიქონის საშუალებით:

```
csc XazvaDemo.cs Daxazeobieqti.cs Wre.cs Wrfe.cs Kvadrati.cs
```

იგი ქმნის ფაილს *XazvaDemo.exe*, რომელიც შეთანხმებისამებრ ემთხვევა ბრძანების სტრიქონის პირველი ფაილის სახელს.

ზოგადად C# პროგრამის სრუქტურა და დამუშავების პროცესი წარმოდგენილია ნახ.-ზე 9.1.

ნახ. 9.1.



ერთი და იგივე სახელსივრცე შეიძლება სხვადასხვა ფაილებში იყოს განთავსებული. თუ სახელსივრცე მოცემული არ არის, იგულისხმება უსახელო გლობალური სახელსივრცე. უმარტივესი შემთხვევაა: ერთი კლასი, ერთი ფაილი, უსახელო სახელსივრცე.

თავი X

თვისებები (Properties)

განვიხილავთ შემდეგ საკითხებს:

- თვისებების ცნება;
- თვისებების აღწერა;
- მხოლოდ წაკითხვადი თვისება;
- მხოლოდ ჩაწერადი თვისება.

თვისებები საშუალებას იძლევა ვაკონტროლოთ კლასის ველებში ინფორმაციის ჩაწერა და წაკითხვა. C#-ი საშუალებას იძლევა თვისებები გარეგნულად ისევე გამოვიყენოთ, როგორც ველებს ვიყენებთ. იმისთვის, რომ უკეთ გავიგოთ, თუ რა როლს თამაშობენ თვისებები, განვიხილოთ, როგორ ხდება ველების დაცვა (ინკაფსულაცია) ტრადიციული მეთოდების გამოყენებით.

ამონაბეჭდი 10-1. კლასის ველებისადმი მიმართვის ტრადიციული ხერხი: Accessors.cs

```
using System;
public class TvisebebisSemcveli
{   private int raimeTviseba = 0;

   public int miiReRaimeTviseba()
   {       return raimeTviseba;
   }
   public void CarTeRaimeTviseba(int TvisebismniSvneloba)
   {       raimeTviseba = TvisebismniSvneloba;
   }}
public class Tvisebebistesti
{   public static int Main(string[] argumentebi)
   {  TvisebebisSemcveli TvisebSemc = new TvisebebisSemcveli();
      TvisebSemc.CarTeRaimeTviseba(5);
      Console.WriteLine("თვისების მნიშვნელობა: {0}", TvisebSemc.miiReRaimeTviseba());
      return 0;
   }}
```

ამონაბეჭდში 10-1 ნაჩვენებია კლასის ველების ცვლილების ტრადიციული მეთოდი. ასეთ ველს შეიცავს კლასი *TvisebebisSemcveli*. მას აქვს ორი მეთოდი: *miiReRaimeTviseba()* და *CarTeRaimeTviseba()*. მეთოდი *miiReRaimeTviseba()* აბრუნებს (იძლევა) ველის *raimeTviseba* მნიშვნელობას. მეთოდი *CarTeRaimeTviseba()* ანიჭებს მნიშვნელობას ველს *raimeTviseba*.

Tvisebebistesti კლასი იყენებს *TvisebebisSemcveli* კლასის მეთოდს, რათა მიიღოს *raimeTviseba* ველის მნიშვნელობა. *Main()* მეთოდი ქმნის *TvisebebisSemcveli* ტიპის ახალ ობიექტს *TvisebSemc*. შემდეგ *TvisebSemc*-ი *CarTeRaimeTviseba* მეთოდის საშუალებით ანიჭებს მნიშვნელობას 5 კლასის (*TvisebebisSemcveli)* საკუთარ ველს *raimeTviseba*. შემდეგ პროგრამა ბეჭდავს ამ თვისების მნიშვნელობას *Console.WriteLine()* მეთოდის საშუალებით. თვისების მნიშვნელობის მისაღებად გამოიყენება *TvisebSemc*-ობიექტის მეთოდი *miiReRaimeTviseba()*. კონსოლზე იბეჭდება მნიშვნელობა "Property Value: 5".

ველებიდან მონაცემთა მიღების ეს მეთოდი კარგია, ვინაიდან ეთანადება ინკაფსულაციის (ჰერმეტიზაციის, დაცვის) ობიექტზე ორიენტირებულ შეხედულებებს. თუ ველს (ცვლადს) *raimeTviseba*-ს შევცვლით *int* ტიპიდან *byte* ტიპზე, პროგრამა კვლავ იმუშავებს. იგივე უფრო იოლად შეიძლება განხორციელდეს თვისებების საშუალებით.

ამონაბეჭდში 10-2. კლასის ველებისადმი მიმართვა თვისებების გამოყენებით: Tvisebebi.cs

```
using System;
public class TvisebebisSemcveli
{   private int raimeTviseba = 0;
   public int RaimeTviseba
   {      get
     {
        return raimeTviseba;
     }
        set
     {   raimeTviseba = value;
     }}}
public class Tvisebebistesti
{   public static int Main(string[] argumentebi)
   {  TvisebebisSemcveli TvisebSemc = new TvisebebisSemcveli();
      TvisebSemc.RaimeTviseba = 5;
      Console.WriteLine("თვისების მნიშვნელობა: {0}", TvisebSemc.RaimeTviseba);
      return 0;
   }}
```

ამონაბეჭდში 10-2 ნაჩვენებია, თუ როგორ შევქმნათ და გამოვიყენოთ თვისებები. *TvisebebisSemcveli* კლასი აღწერს თვისებას *RaimeTviseba*. შევნიშნოთ, რომ პირველი ასო ასომთავრულია. მხოლოდ ამით განსხვავდება თვისების სახელი *RaimeTviseba* ველის სახელისგან *raimeTviseba*. თვისებას აქვს ორი აქსესორი: *get* და *set*. aqsesori *get* უკან აბრუნებს *raimeTviseba* ველის მნიშვნელობას, ხოლო *set* აქსესორი ანიჭებს *value*-ს

მნიშვნელობას *raimeTviseba* ველს. *set* აქსესორში გამოყენებული *value* წარმოადგენს C#-ის საკვანძო სიტყვას.

Tvisebebistesti კლასი იყენებს *RaimeTviseba* თვისებას *TvisebebisSemcveli* კლასში. *Main()* მეთოდის პირველი სტრიქონი ქმნის *TvisebebisSemcveli* ობიექტის ობიექტს სახელად *TvisebSemc*. შემდეგ *raimeTviseba* ველს ენიჭება მნიშვნელობა 5 *TvisebSemc* ობიექტის *RaimeTviseba* თვისების გამოყენებით. თვისებებისთვის მნიშვნელობის მინიჭება ველებისთვის მნიშვნელობის მინიჭების მსგავსია.

ამის შემდეგ *Console.WriteLine()* მეთოდი ბეჭდავს *TvisebSemc*-ის *raimeTviseba* ველის მნიშვნელობას. ეს ხდება *TvisebSemc* ობიექტის *RaimeTviseba* თვისების საშუალებით და ეს კვლავ უფრო ადვილია, ვინაიდან ვიყენებთ თვისებას ისე, თითქოს იგი ველი იყოს.

თვისებები მხოლოდ წაკითხვადი შეიძლება გავხადოთ. ამის განხორციელება შეიძლება, თუ მათ აღწერაში გამოვიყენებთ მხოლოდ *get* აქსესორს.

ამონაბეჭდი 10-3. მხოლოდ წაკითხვადი თვისება: MxolodwakiTxvadiTvis.cs

```
using System;
public class TvisebebisSemcveli
{   private int raimeTviseba = 0;
   public TvisebebisSemcveli(int TvisebismniSvneloba)
   {       raimeTviseba = TvisebismniSvneloba;
   }
   public int RaimeTviseba
   {  get
     {  return raimeTviseba;
      }}}
public class Tvisebebistesti
{
   public static int Main(string[] argumentebi)
   {  TvisebebisSemcveli TvisebSemc = new TvisebebisSemcveli(5);
      Console.WriteLine("თვისების მნიშვნელობა: {0}", TvisebSemc.RaimeTviseba);
      return 0;
   }}
```

ამონაბეჭდი 10-3 გვიჩვენებს მხოლოდ წაკითხვად თვისებებს. *TvisebebisSemcveli* კლასს აქვს თვისება *RaimeTviseba*, რომელიც მხოლოდ *get* აქსესორს იყენებს. იგი ტოვებს *set* აქსესორს. ამ განსაკუთრებულ *TvisebebisSemcveli* კლასს გააჩნია კონსტრუქტორი, რომელიც ღებულობს *int* პარამეტრს.

Tvisebebistesti კლასის *Main()* მეთოდი ქმნის ახალ *TvisebebisSemcveli* ობიექტს სახელად *TvisebSemc*. *TvisebSemc* ობიექტის შექმნისას გამოიყენება *TvisebebisSemcveli*ის კონსტრუქტორი, რომელიც ღებულობს *int* პარამეტრს. ამ შემთხვევაში ეს არის 5.

ვინაიდან *TvisebebisSemcveli* კლასის *RaimeTviseba* თვისება მხოლოდ წაკითხვადია, არ არსებობს სხვა გზა *raimeTviseba* ველის მნიშვნელობის შესაცვლელად. თუ ჩასვამთ პროგრამაში *TvisebSemc.RaimeTviseba = 7,* მისი კომპილირება არ მოხდება, ვინაიდან *RaimeTviseba* მხოლოდ წაკითხვადია. როცა *RaimeTviseba* თვისება გამოიყენება *Console.WriteLine()* მეთოდში, ის მშვენივრად მუშაობს. ეს ხდება იმის გამო, რომ წაკითხვის ოპერაცია ააქტიურებს თვისების *RaimeTviseba* მხოლოდ *get* აქსესორს.

ამონაბეჭდი 10-4. მხოლოდ ჩაწერადი თვისება: MxolodCaweradiTvis.cs

```
using System;
public class TvisebebisSemcveli
{   private int raimeTviseba = 0;
   public int RaimeTviseba
   {
      set
     {   raimeTviseba = value;
         Console.WriteLine("რაიმე  თვისება ტოლია {0}", raimeTviseba);
      }}}
public class Tvisebebistesti
{   public static int Main(string[] argumentebi)
   {  TvisebebisSemcveli TvisebSemc = new TvisebebisSemcveli();
      TvisebSemc.RaimeTviseba = 5;
      return 0;
   }}
```

ამონაბეჭდი 10-4 გვიჩვენებს, თუ როგორ შევქმნათ მხოლოდ ჩაწერადი თვისება. ამ შემთხვევაში *get* აქსესორი ამოიღება *TvisebebisSemcveli* კლასის *RaimeTviseba* თვისებიდან. *set* აქსესორი დაემატება. იბეჭდება *raimeTviseba* ველის მნიშვნელობა იმის შემდეგ, რაც იგი შეიცვლება.

Tvisebebistesti კლასის *Main()* მეთოდი ქმნის ამ კლასის ეგზემპლარს სტანდარტული კონსტრუქტორის საშუალებით. შემდეგ იგი იყენებს *TvisebSemc* ობიექტის *RaimeTviseba* თვისებას *raimeTviseba* ველისთვის მნიშვნელობის 5 მისანიჭებლად. იგი ააქტიურებს *TvisebSemc* ობიექტის *set* აქსესორს, რომელიც ანიჭებს *raimeTviseba* ველს მნიშვნელობას 5 და კონსოლზე ბეჭდავს: "**რაიმე თვისება ტოლია 5**".

თავი XI

ინდექსერები (Indexers)

განვიხილავთ შემდეგ საკითხებს:

- ინდექსერის ცნება;
- ინდექსერის აღწერა;
- ინდექსერის გადატვირთვა;
- მრავალპარამეტრიანი ინდექსერის გამოყენება.

ინდექსერები საშუალებას იძლევა, გამოიყენოთ კლასის ეგზემპლარები მასივების მსგავსად. კლასის შიგნით თქვენი სურვილისამებრ მართავთ მნიშვნელობათა კოლექციას. ეს ობიექტები შეიძლება იყოს კლასების სასრული სიმრავლეები, სხვა მასივები ან მონაცემთა სხვა კომპლესური სტრუქტურები. კლასის შიდა აღწერისგან მიუხედავად, მისი წევრების გამოყენება შეიძლება ინდექსერების საშუალებით, როგორც ნაჩვენებია ქვემოთ მაგალითში.

ამონაბეჭდი 11-1. ინდექსერის მაგალითი: indeqseri.cs

```
using System;
class MTeliindeqseri
{   private string[] monacemebi;
   public MTeliindeqseri(int zoma)//კლასის  კონსტრუქტორი
   {       monacemebi = new string[zoma];//ქმნის  სტრიქონების  მასივს
      for (int i=0; i < zoma; i++)
      {       monacemebi[i] = "ცარიელი";
      }
   }
   public string this[int pozicia]
   {
      get
     {    return monacemebi[pozicia];
      }
      set
     {  monacemebi[pozicia] = value;
      }
   }
   static void Main(string[] argumentebi)
   {   int zoma = 10;
```

```
        MTeliindeqseri indeqseri = new MTeliindeqseri(zoma);
        indeqseri[9] = "რაიმე მნიშვნელობა";
        indeqseri[3] = "სხვა მნიშვნელობა";
        indeqseri[5] = "ნებისმიერი მნიშვნელობა";
        Console.WriteLine("\nინდექსერის გამოსავალი\n");
        for (int i=0; i < zoma; i++)
        {   Console.WriteLine("ინდექსერი[{0}]: {1}", i, indeqseri[i]);
        }
    }
}
```

ამონაბეჭდი 11-1 გვიჩვენებს თუ, როგორ აღვწეროთ და გამოვიყენოთ ინდექსერი. *MTeliindeqseri* კლასს აქვს *string* მასივი სახელად *monacemebi*. ეს არის კერძო (შიდა მოხმარების) მასივი, რომელსაც გარე მომხმარებელი ვერ ხედავს. ამ მასივის ინიციალიზაცია ხდება კონსტრუქტორით, რომელიც ღებულობს *int zoma* პარამეტრს, ქმნის *monacemebi* მასივს და ყოველ ველს ავსებს სიტყვით "ცარიელი".

კლასის შემდეგი წევრია ინდექსერი, რომელიც განისაზღვრება საკვანძო სიტყვის *this* და კვადრატული ფრჩხილებით *this[int pozicia]*. იგი ღებულობს ერთადერთ პოზიციურ პარამეტრს *pozicia*. როგორც მიხვდებოდით, ინდექსერის აღწერა თვისების აღწერის მსგავსია. მას აქვს *get* და *set* აქსესორები, რომელთა გამოყენებაც ზუსტად ისეთივეა, როგორც თვისებებში. ინდექსერი უკან აბრუნებს *string*-ს, როგორც ნაჩვენებია ინდექსერის აღწერაში *string* დაბრუნების ტიპის საშუალებით.

Main() მეთოდი უბრალოდ ქმნის ახალ *MTeliindeqseri* ობიექტს, ამატებს მას რაღაც მნიშვნელობებს და ბეჭდავს შედეგებს. ქვემოთ მოყვანილია გამოსავალი:

```
ინდექსერი[0]: ცარიელი
ინდექსერი[1]: ცარიელი
ინდექსერი[2]: ცარიელი
ინდექსერი[3]: სხვა მნიშვნელობა
ინდექსერი[4]: ცარიელი
ინდექსერი[5]: ნებისმიერი მნიშვნელობა
ინდექსერი[6]: ცარიელი
ინდექსერი[7]: ცარიელი
ინდექსერი[8]: ცარიელი
ინდექსერი[9]: რაიმე მნიშვნელობა
```

მასივის ელემენტების განსაზღვრისთვის მრავალ ენაში ინდექსად მიღებულია *integer*-ის ტიპის გამოყენება, მაგრამ C#-ში ეს უფრო ფართოდაა წარმოდგენილი.

ინდექსერები შეიძლება აღიწერონ მრავალი პარამეტრით და ყოველ მათგანს შეიძლება ჰქონდეს სხვადასხვა ტიპი. დამატებითი პარამეტრები გამოიყოფა მძიმეებით ისევე, როგორც მეთოდის პარამეტრების სიაში. ინდექსერებისთვის დასაშვები პარამეტრების ტიპებია: *integers*, *enums*, და *strings*. გარდა ამისა, ინდექსერები შეიძლება იყოს გადატვირთული. ამონაბეჭდში 11-2 იმგვარად შევცვალეთ წინა პროგრამა, რომ ინდექსერებს მიეღოთ განსხვავებული ტიპები.

ამონაბეჭდი 11-2. ინდექსერების გადატვირთვა: indexergad.cs

```
using System;
class Indeqgad
{  private string[] monacemebi;
   private int        arrZoma;
   public Indeqgad(int zoma)
   {  arrZoma = zoma;
      monacemebi = new string[zoma];
      for (int i=0; i < zoma; i++)
      {   monacemebi[i] = "ცარიელი";
      }
   }
   public string this[int pozicia]
   {
      get
     {  return monacemebi[pozicia];
      }
      set
     {
         monacemebi[pozicia] = value;
      }
   }
   public string this[string monacemi]
   {
      get
     {  int mTvleli = 0;
         for (int i=0; i < arrZoma; i++)
         {  if (monacemebi[i] == monacemi)
```

```
            { mTvleli++;
            }
        }
        return mTvleli.ToString();
    }
    set
    {
        for (int i=0; i < arrZoma; i++)
        {  if (monacemebi[i] == monacemi)
            {  monacemebi[i] = value;
            }
        }
    }
  }
  static void Main(string[] argumentebi)
  {
    int zoma = 10;
    Indeqgad indeqseri = new Indeqgad(zoma);
    indeqseri[9] = "რაიმე  მნიშვნელობა";
    indeqseri[3] = "სხვა  მნიშვნელობა";
    indeqseri[5] = "ნებისმიერი მნიშვნელობა ";
    indeqseri["carieli"] = "მნიშვნელობის  არმქონე";
    Console.WriteLine("\nინდექსერის  გამოსავალი:\n");
    for (int i=0; i < zoma; i++)
    { Console.WriteLine("indeqseri[{0}]: {1}", i, indeqseri[i]);
    }
    Console.WriteLine("\nრაოდენობა \"  მნიშვნელობის  არმქონე \" წევრების: {0}",
indeqseri["მნიშვნელობის  არმქონე "]);
  }
}
```

ამონაბეჭდში 11-2 გვიჩვენებს, თუ როგორ გადავტვირთოთ ინდექსერები. პირველი ინდექსერი *int* ტიპის *pozicia* პარამეტრით იგივეა, რაც ამონაბეჭდში 11-1, თუმცა არის ახალი ინდექსერიც, რომელიც ღებულობს *string* პარამეტრს. ახალი ინდექსერის *get* აქსესორი უკან აბრუნებს რიცხვის *string* წარმოდგენას (სტრიქონის სახით წარმოდგენას), რომელიც ეთანადება *monacemi* პარამეტრის მნიშვნელობას. *set* აქსესორი ცვლის მასივის

ყოველ ელემენტს, რომელსაც ინდექსერის მინიჭებაში ეთანადება *monacemi* პარამეტრის მნიშვნელობა.

ინდექსერის, რომელიც ღებულობს *string* პარამეტრს, გადატვირთვის ქცევა წარმოდგენილია *Main()* მეთოდის ამონაბეჭდში 11-2. იგი ააქტიურებს *set* აქსესორს, რომელიც ანიჭებს "მნიშვნელობის არმქონე" მნიშვნელობას *indeqseri* კლასის ყოველ წევრს, რომელსაც აქვს მნიშვნელობა "carieli". იგი იყენებს შემდეგ ოპერატორს: *indeqseri["carieli"] = "*მნიშვნელობის არმქონე";. როცა დაიბეჭდება *indeqseri* კლასის ყოველი ელემენტი, ამონაბეჭდის ბოლო სტრიქონი გვიჩვენებს იმ ელემენტთა რაოდენობას, რომელთა მნიშვნელობაცაა "მნიშვნელობის არმქონე". ეს ხდება *get* აქსესორის შემდეგი ოპერატორის გამოძახებით. ქვემოთ მოყვანილია გამოსავალი:

ინდექსერის გამოსავალი:

ინდექსერი[0]: მნიშვნელობის არმქონე

ინდექსერი[1]: მნიშვნელობის არმქონე

ინდექსერი[2]: მნიშვნელობის არმქონე

ინდექსერი[3]: სხვა მნიშვნელობა

ინდექსერი[4]: მნიშვნელობის არმქონე

ინდექსერი[5]: ნებისმიერი მნიშვნელობა

ინდექსერი[6]: მნიშვნელობის არმქონე

ინდექსერი[7]: მნიშვნელობის არმქონე

ინდექსერი[8]: მნიშვნელობის არმქონე

ინდექსერი[9]: რაიმე მნიშვნელობა

რაოდენობა "მნიშვნელობის არმქონე" წევრების: 0

ამონაბეჭდში 11-2 ორივე ინდექსერის ერთსა და იმავე კლასში თანაარსებობის მიზეზი მდგომარეობს იმაში, რომ მათ აქვთ სხვადასხვა სიგნატურები. ინდექსერის სიგნატურა განისაზღვრება პარამეტრების რაოდენობითა და ტიპით ინდექსერის პარამეტრების სიაში. კლასს შეუძლია, გაარკვიოს სიგნატურის მიხედვით, თუ რომელი ინდექსერი გამოიძახოს. ინდექსერები მრავალი პარამეტრით შესაძლებელია აღიწეროს ქვემომოყვანილის მსგავსად:

```
public object this[int param1, ..., int paramN]
{
   get
{ // გარკვეული პროცესი და კლასის რაიმე მონაცემების მნიშვნელობის  მიღება
}
   set
{ // გარკვეული  პროცესი და კლასის რაიმე მონაცემების მნიშვნელობის
დაყენება   }   }
```

თავი XII

სტრუქტურები (Structs)

განვიხილავთ შემდეგ საკითხებს:

- სტრუქტურის ცნება და დანიშნულება;
- სტრუქტურის აღწერა;
- სტრუქტურის გამოყენება.

სტრუქტურები საშუალებას იძლევა, შევქმნათ მნიშვნელობის ტიპის ახალი ობიექტები, რომლებიც მსგავსია ტიპების: *int*, *float*, *bool* და ა. შ. როდის გამოიყენება სტრუქტურები კლასების მაგივრად? დავფიქრდეთ, როდის ვიყენებთ მნიშვნელობის ზემოჩამოთვლილ ტიპებს. მათ გააჩნიათ მნიშვნელობანი, რომლებზეც შესაძლებელია განსხვავებული ოპერაციების ჩატარება. ზოგადად შეიძლება ითქვას, რომ თუ ობიექტი დიდი არ არის და მას ძალიან ხშირად ვხმარობთ, ეფექტურობიდან გამომდინარე, სჯობს გამოვიყენოთ სტრუქტურები და არა კლასები. ქვემოთ მოყვანილია სტრუქტურის მაგალითი.

ამონაბეჭდი 12-1. სტრუქტურის მაგალითი: StruqturismagaliTi.cs

```
using System;
struct Wertili
{   public int x;
    public int y;
    public Wertili(int x, int y)
    {   this.x = x;
        this.y = y;
    }
    public Wertili Daamate(Wertili pt)
    {   Wertili axaliPt;
        axaliPt.x = x + pt.x;
        axaliPt.y = y + pt.y;
        return axaliPt;
    }
}
class Struqturis magaliTi
{   static void Main(string[] argumentebi)
    {   Wertili pt1 = new Wertili(1, 1);
        Wertili pt2 = new Wertili(2, 2);
```

```
    Wertili pt3;
    pt3 = pt1.Daamate(pt2);
    Console.WriteLine("pt3: {0}:{1}", pt3.x, pt3.y);
  }
}
```

ამონაბეჭდი 12-1 გვიჩვენებს, თუ როგორ აღვწეროთ და გამოვიყენოთ სტრუქტურა. სტრუქტურის ძირითადი ფორმატი კლასის მსგავსია, თუმცა არის განსხვავებები, რომელთაც შევეხებით მოგვიანებით. სიტყვა *struct* საკვანძოა და ამ სიტყვით იწყება სტრუქტურის აღწერა. *Wertili* სტრუქტურას აქვს კონსტრუქტორი, რომელიც ველებს *x* და *y* ანიჭებს ფორმალური არგუმენტების მნიშვნელობებს. მას აგრეთვე აქვს მეთოდი *Daamate(),* რომელიც ღებულობს სხვა Wertili ტიპის სტრუქტურას, უმატებს მას ამ სტრუქტურას და უკან აბრუნებს, როგორც ახალ სტრუქტურას.

Daamate() მეთოდში არის *Wertili* სტრუქტურის აღწერა. მსგავსად კლასისა, აქაც არ არის new ოპერატორის გამოყენების საჭიროება. როცა ეს ხდება, იქმნება სტრუქტურა არაცხადი კონსტრუქტორის საშუალებით. არაცხადი კონსტრუქტორი სტრუქტურის თითოეულ ველს ანიჭებს სტანდარტით განსაზღვრულ მნიშვნელობებს, მაგალითად, მთელებს 0-ს, მცურავწერტილიანებს 0.0-ს, ხოლო ლოგიკურებს - *false*. უპარამეტრებო კონსტრუქტორების ცხადად გამოყენება სტრუქტურაში დაშვებული არ არის.

თუმცა არ არის სავალდებულო *new* ოპერატორის სტრუქტურის ინიციალიზაცია, ამონაბეჭდში 12-1 *pt1* და *pt2 Wertili* ტიპის სტრუქტურები ინიციალიზირებულია *Wertili* სტრუქტურაში აღწერილი კონსტრუქტორით. მესამე *Wertili* სტრუქტურა *pt3* აღწერილია და იყენებს არაცხად კონსტრუქტორს, ვინაიდან არა აქვს მნიშვნელობა, რა მონაცემები იქნება ამ სტრუქტურაში. *pt1* სტრუქტურის *Daamate()* მეთოდი გააქტიურებისას აგზავნის *pt2* სტრუქტურას, როგორც პარამეტრს. შედეგი მიენიჭება *pt3*-ს, რაც გვიჩვენებს იმას, რომ სტრუქტურა შეიძლება სხვა *value* (მნიშვნელობის) ტიპის ცვლადების მსგავსად იქნეს გამოყენებული. ქვემოთ მოყვანილია ამონაბეჭდის 12-1 შედეგები:

```
pt3: 3:3
```

შემდეგი განსხვავება სტრუქტურასა და კლასს შორის იმაში მდგომარეობს, რომ სტრუქტურას არ შეიძლება ჰქონდეს დესტრუქტორი, ასევე სტრუქტურას არ შეიძლება ჰქონდეს მემკვიდრეობითობა სხვა კლასებთან და სტრუქტურებთან. რა თქმა უნდა, სტრუქტურას შეიძლება ჰქონდეს მემკვიდრეობითობა ინტერფეისებთან (მსგავსია კლასების, რომელთაც არ აქვთ მეთოდის რეალიზაცია, აქვთ მხოლოდ მეთოდის სახელი და გამოყენებული პარამეტრების სია). ნებისმიერ კლასი ან სტრუქტურა, რომელიც მემკვიდრეობით ღებულობს ინტრფეისიდან ვალდებულია მოახდინოს ამ ინტერფეისში აღწერილი მეთოდების აღწერა (რეალიზაცია). ინტერფეისები უფრო დაწვრილებით განხილულია შემდეგ თავში.

თავი XIII

ინტერფეისები (Interfaces, ურთიერთშეკავშირებები)

განხილულია შემდეგი საკითხები:

- ინტერფეისის დანიშნულება;
- ინტერფეისის აღწერა;
- ინტერფეისის გამოყენება;
- ინტერფეისის მემკვიდრეობითობის გამოყენება.

ინტერფეისი კლასის მსგავსად გამოიყურება. იგი შეიცავს მხოლოდ მეთოდების, თვისებების, ინდექსერების, მოვლენების სათაურებს, მათი რეალიზაციის გარეშე (ანუ 100% აბსტრაქტული კლასია, ამ უკანასკნელში ზოგიერთი მეთოდი შეიძლება რეალიზებული იყოს). მათი რეალიზაცია (დაკონკრეტება) ხდება იმ სტრუქტურებსა და კლასებში, რომლებიც მათ იღებენ, როგორც მემკვიდრეობას. იბადება კითხვა, თუ რა სარგებლობა მოაქვს ინტერფეისებს, თუ მათში არ არის ფუნქციონალურობა (მეთოდები). ეს სასარგებლოა ე. წ. თავსებადი არქიტექტურების დამუშავებისას მაგალითად, როგორიცაა *plug-n-play*. ეს მიდგომა აიძულებს ზედა დონის მოდულებს გამოიყენონ ინტერფეისებით დადგენილი ურთიერთობის სტანდარტული წესები. ამით შესაძლებელი ხდება ერთი დონის პროგრამული კომპონენტების გაცვლა და ცვლილება ისე, რომ არ შევეხოთ სხვა კომპონენტებს და არ დავარღვიოთ სისტემის მთლიანი ფუნქციონირება. ყველა ურთიერთშეცვლადი კომპონენტი ერთსა და იმავე ინტერფეისს განახორციელებს. ინტერფეისი ყოველ კომპონენტს აიძულებს იქონიოს განსაზღვრული საჯარო (*public*) წევრები, რომლებიც სათანადოდ იქნება გამოყენებული.

ვინაიდან ინტერფეისები უნდა იყოს რეალიზებული კლასებსა და სტრუქტურებში, ისინი განსაზღვრავენ გარკვეულ შეთანხმებას. მაგალითად, თუ კლასი მემკვიდრეობით ღებულობს ინტერფეისს, მაშინ ის აუცილებლად უნდა უზრუნველყოფდეს მეთოდის რეალიზაციას ინტერფეისით დადგენილი სიგნატურით. ამონაბეჭდში 13-1 ნაჩვენებია, თუ როგორ განისაზღვრება ინტერფეისი.

ამონაბეჭდი 13-1. ინტერფეისის განსაზღვრა: Interfasi.cs

```
interface IinterfeisiX
{  void MeTodisdanergva();
}
```

ამონაბეჭდი 13-1 გვიჩვენებს ინტერფეის *IinterfeisiX* განსაზღვრებას. სახელების საერთაშორისო კონვენციის თანახმად ინტერფეისების სახელები უნდა იწყებოდეს ასომთავრული "I"–ით. ამ ინტერფეისს აქვს ერთადერთი მეთოდი სახელად

MeTodisdanergva(). ინტერფეისს შესაძლებელია ჰქონდეს ნებისმიერი ტიპის მეთოდების განსაზღვრება სხვადასხვა პარამეტრების სიით და დაბრუნების ტიპით (სიგნატურით). სიმარტივისთვის მოცემულია უპარამეტრებო და მნიშვნელობის დაბრუნების გარეშე მეთოდი. შევნიშნოთ, რომ ამ მეთოდს არ გააჩნია ფიგურულ ფრჩხილებში {} მოთავსებული ოპერატორები, მაგრამ სამაგიეროდ მთავრდება წერტილ-მძიმით ";". ეს იმიტომ ხდება, რომ ინტერფეისი ახდენს მხოლოდ მეთოდის სიგნატურის სპეციფიცირებას (აღწერას, განსაზღვრას), რომლის რეალიზაცია უნდა მოხდეს იმ მეთოდსა და კლასში, რომელიც მემკვირეობას მიიღებს ამ ინტერფეისიდან. ამონაბეჭდი 13-2 გვიჩვენებს თუ, როგორ გამოვიყენოთ ეს ინტერფეისი.

ამონაბეჭდი 13-2 . ინტერფეისის გამოყენება: Interfeisisganxorcieleba.cs

```
class Interfeisisganxorcieleba : IinterfeisiX
{   static void Main()
    {   Interfeisisganxorcieleba iImp = new Interfeisisganxorcieleba();
        iImp.MeTodisdanergva();
    }
    public void MeTodisdanergva()
    {   Console.WriteLine("MeTodisdanergva() გამოძახება.");
    }
}
```

ამონაბეჭდში 13-.2 კლასი *Interfeisisganxorcieleba* ახდენს ინტერფეისის *IinterfeisiX* რეალიზაციას. ინტერფეისის მემკვიდრეობითობის აღწერა მსგავსია კლასის მემკვიდრეობითობის აღწერისა. ამ შემთხვევაში გამოიყენება შემდეგი სინტაქსი:

```
class Interfeisisganxorcieleba : IinterfeisiX
```

ეს კლასი მემკვიდრეობით ღებულობს ინტერფეისს *IinterfeisiX*, და რეალიზაციას უკეთებს მის წევრებს. ამას ახორციელებს *MeTodisdanergva()* მეთოდის აღწერით. აღვნიშნოთ, რომ ნებისმიერი განსხვავება ინტერფეისში მოცემულ და კლასში აღწერილ მეთოდის სიგნატურებს შორის იწვევს შეცდომის შეტყობინებას კომპილაციის დროს. ამონაბეჭდში 13-3 ნაჩვენებია, თუ როგორ ხდება ინტერფეისის მემკვიდრეობითობა და რეალიზაცია.

ამონაბეჭდი 13-3. ინტერფეისის მემკვიდრეობითობა: InterfaceInheritance.cs

```
using System;
interface IMSobeliInterface
{   void MSobeliInterfaceMethod();
}
interface IinterfeisiX : IMSobeliInterface
```

```
{   void MeTodisdanergva();
}
class Interfeisisganxorcieleba : IinterfeisiX
{   static void Main()
   { Interfeisisganxorcieleba iImp = new Interfeisisganxorcieleba();
     iImp.MeTodisdanergva();
     iImp.MSobeliInterfaceMethod();
   }
   public void MeTodisdanergva()
   {  Console.WriteLine("MeTodisdanergva() გამოძახებულია.");
   }
   public void MSobeliInterfaceMethod()
   {  Console.WriteLine("MSobeliInterfaceMethod() გამოძახებულია.");
   }
}
```

ამონაბეჭდში 13-3 მოთავსებული კოდი ორ ინტერფეისს შეიცავს: *IinterfeisiX* და *IMSobeliInterface*, რომლიდანაც ღებულობს მემკვიდრეობას *IinterfeisiX* ინტერფეისი. როცა ერთი ინტერფეისი ღებულობს მემკვიდრეობას მეორისგან, ყოველი კლასი ან სტრუქტურა, რომელიც ამ ინტერფეისის რეალიზაციას ახდენს, ვალდებულია გაითვალისწინოს ინტერფეისის მემკვიდრეობითობის ჯაჭვში შემავალი ყველა წევრი. ვინაიდან კლასი *Interfeisisganxorcieleba* ამონაბეჭდში 13-3 მემკვიდრეობას ღებულობს *IinterfeisiX*, იგი აგრეთვე ღებულობს მემკვიდრეობას *IMSobeliInterface* ინტერფეისიდანაც, ამიტომ კლასი *Interfeisisganxorcieleba* უნდა ახდენდეს მეთოდის *MeTodisdanergva()* რეალიზაციას, რომელიც განსაზღვრულია *IinterfeisiX* ინტერფეისში და მეთოდის *MSobeliInterfaceMethod()* რეალიზაციას, რომელიც განსაზღვრულია *IMSobeliInterface* ინტერფეისში.

თავი XIV

შესავალი დელეგატებსა და მოვლენებში

ამ თავში განხილულია შემდეგი საკითხები:

- დელეგატის ცნება;
- მოვლენის ცნება;
- დელეგატების გამოყენება;
- მოვლენის გაშვება.

დელეგატები

დელეგატების საშუალებით ხდება ისეთი ტიპის ცვლადების აღწერა, რომელთა მნიშვნელობებიც მეთოდების სახელებია. სხვა სიტყვებით რომ ვთქვათ, ცლადის მნიშვნელობა რაიმე მეთოდის სახელია, რომელიც შესაძლებელია გამოვიყენოთ, როგორც მეთოდის ფაქტიური და ფორმალური პარამეტრები.

დელეგატები გამოიყენება ფუნქციის (მეთოდის) სახელის გადასაცემად მეთოდში ფაქტიური პარამეტრის სახით ანუ მეთოდებში ფაქტიური არგუმენტის გამოსაყენებლად ფუნქციის სახელად. გარდა ამისა დელეგატები გამოიყენება გამონაკლისი შემთხვევების (მოვლენების) დამუშავებისას. დელეგატები წარმოადგენენ მითითების *(reference)* ახალი ტიპის ცვლადების აღწერის საშუალებას.

დელეგატები C#-ში არის ენის ელემენტები, რომლებიც საშუალებას იძლევა, განვახორციელოთ მითითება მეთოდზე.

როგორ ვიყენებდით მეთოდებს აქამდე? რეალიზებას ვუკეთებდით ალგორითმს, რომელიც ახდენდა ცვლადების მნიშვნელობებით მანიპულირებას და მეთოდებს ვიძახებდით სახელის პირდაპირი მითითებით. რა მოხდება, თუ გვაქვს ალგორითმი, რომელიც ახორციელებს რაიმე ტიპის მონაცემთა სტრუქტურის დახარისხებას (სორტირებას). ასევე დავუშვათ, სხვადასხვა შემთხვევაში მონაცემთა ეს სტრუქტურა განსხვავებული ტიპების იყოს. თუ არ არის ცნობილი მათი ტიპები, როგორ მოახდენთ პროგრამის შესრულების პროცესში მათ შედარებას? საყოველთაოდ ცნობილი ტიპებისთვის შეიძლება ეს განახორციელოთ, მაგალითად, *if/then/else* ან *switch* ოპერატორებით, მაგრამ ეს ცნობილი ტიპებით იქნება შეზღუდული და მოითხოვს დამატებით ქმედებებს ტიპების განსაზღვრისთვის. განსხვავებული მიდგომა იქნება ყველა ტიპისთვის ინტერფეისის რეალიზაცია საერთო მეთოდის გამოძახების საშუალებით. დელეგატების გამოყენება ამის გადასაწყვეტად სავსებით დახვეწილი საშუალებაა.

ამ პრობლემას ვწყვეტთ დელეგატის გაგზავნით ჩვენს ალგორითმში, რომელიც მოიცავს მეთოდს, რომელზეც მიუთითებს დელეგატი. ეს მეთოდი ჩვენს შემთხვევაში ახორციელებს შედარების ოპერაციას. ეს ოპერაციები ნაჩვენებია ამონაბეჭდში 14-1.

ამონაბეჭდი 14-1. დელეგატების აღწერა და გამოყენება : Ubralodelegati.cs

```
using System;
// ეს არის დელეგატის აღწერა.
public delegate int Semdarebeli(object obj1, object obj2);
public class Saxeligvari
{       public string Saxeli = null;// null, ნიშნავს, რომ არც ერთ ობიექტზე არ მიუთითებს
        public string Gvari = null;// null  ენის საკვანძო სიტყვაა
        public Saxeligvari(string pirveli, string bolo)
        {       Saxeli = pirveli;
                Gvari = bolo;
        }
        // ეს არის დელეგატის დამმუშავებელი მეთოდი
        public static int SaxelebisSedareba(object saxeligvari1, object saxeligvari2)
        {       string n1 = ((Saxeligvari)saxeligvari1).Saxeli;
                string n2 = ((Saxeligvari)saxeligvari2).Saxeli;
                if (String.Compare(n1, n2) > 0)
                {return 1;
                }
                else if (String.Compare(n1, n2) < 0)
                {return -1;
                }
                else
                {                       return 0;
                }
        }
        public override string ToString()
        {       return Saxeli + " " + Gvari;
        }
}
class Ubralodelegati
{Saxeligvari[]saxelebigvarebi = new Saxeligvari[5];
        public Ubralodelegati()
        {       saxelebigvarebi[0] = new Saxeligvari("ნინო", "ბახტაძე");
                saxelebigvarebi[1] = new Saxeligvari("თენგიზ"," ბახტაძე ");
                saxelebigvarebi[2] = new Saxeligvari("ნიკა", "ჩიტიძე");
                saxelebigvarebi[3] = new Saxeligvari("ხათუნა", "იმნაძე");
                saxelebigvarebi[4] = new Saxeligvari("ქსენია", "უგულავა");
        }
```

```
        static void Main()
        {
// დელეგატის განსაზღვრა
                Ubralodelegati sd = new Ubralodelegati();
// ეს არის დელეგატის ეგზემპლარის შექმნა
Semdarebeli cmp = new Semdarebeli(Saxeligvari.SaxelebisSedareba);
                Console.WriteLine("\nდაწყობამდე: \n");
                sd.BewdvaSaxelebigvarebi();
                // დელეგატის არგუმენტად გამოყენება
                sd.Dalageba(cmp);
                Console.WriteLine("\nდაწყობის  შემდეგ: \n");
                sd.BewdvaSaxelebigvarebi();
        }
                public void Dalageba(Semdarebeli Seadare)
        {       object droebiTi;
                for (int i=0; i < saxelebigvarebi.Length; i++)
                {       for (int j=i; j < saxelebigvarebi.Length; j++)
                        {if ( Seadare(saxelebigvarebi[i], saxelebigvarebi[j]) > 0 )
                                {       droebiTi = saxelebigvarebi[i];
                                        saxelebigvarebi[i] = saxelebigvarebi[j];
                                        saxelebigvarebi[j] = (Saxeligvari)droebiTi;
                                }
                        }
                }
        }
        public void BewdvaSaxelebigvarebi()
        {       Console.WriteLine("სახელებიგვარები: \n");
                foreach (Saxeligvari sg in saxelebigvarebi)
                {Console.WriteLine(sg.ToString());
                }}}
```

გამოსავალი:

```
დაწყობამდე:
სახელები გვარები:
ნინო ბახტაძე
თენგიზ  ბახტაძე
ნიკა ჩიტიძე
ხათუნა იმნაძე
ქსენია უგულავა
```

დაწყობის შემდეგ:

სახელები გვარები:

ნიკა ჩიტიძე

ნინო ბახტაძე

ქსენია უგულავა

თენგიზ ბახტაძე

ხათუნა იმნაძე

ამონაბეჭდში 14-1 განისაზღვრება დელეგატი. დელეგატის აღწერა გამოიყურება მეთოდის მსგავსად. განსხვავება იმაშია, რომ იგი შეიცავს *delegate* მოდიფიკატორს, მთავრდება წერტილ-მძიმით (;) და არა აქვს რეალიზაცია. ქვემოთ მოყვანილია დელეგატის აღწერა ამონაბეჭდიდან 14-1.

```
public delegate int Semdarebeli(object obj1, object obj2);
```

დელეგატის ეს აღწერა განსაზღვრავს იმ მეთოდის სიგნატურას (პარამეტრების სახეს), რომელსაც მომავალში მიუთითებს Semdarebeli ტიპის ცვლადი ანუ *Semdarebeli* არის ჩვენს მიერ განსაზღვრული ტიპი. დელეგატის მომსახურე მეთოდს *Semdarebeli* დელეგატისთვის შეიძლება ჰქონდეს ნებისმიერი სახელი, მაგრამ პირველ პარამეტრად უნდა ჰქონდეს *object* ტიპი, მეორე პარამეტრადაც *object* ტიპი, ხოლო უკან დასაბრუნებლად - *int* ტიპი. ფაქტიურად სიტყვა *Semdarebeli* განსაზღვრავს ტიპის სახელს, ხოლო დანარჩენი მონაცემები იმ ფუნქციის მახასიათებლებს აზუსტებენ, რომელზედაც მიუთითებს ამ ტიპით აღწერილი ცვლადი. სხვა სიტყვებით რომ ვთქვათ, ზედა სტრიქონი წარმოადგენს დელეგატის ტიპის აღწერას, რომლის სახელია *Semdarebeli*. ანუ *Semdarebeli* არის ტიპი. ამონაბეჭდიდან 14-1 შემდეგი წინადადება განსაზღვრავს დელეგატის დამუშავების მეთოდს, რომელიც *Semdarebeli* დელეგატის სიგნატურას ეთანადება:

```
public static int SaxelebisSedareba(object saxeligvari1, object saxeligvari2)
  {     ...   }
```

დელეგატის გამოსაყენებლად უნდა შექმნათ მისი ეგზემპლარი. იგი იქმნება კლასის ეგზემპლარის მსგავსად ერთადერთი პარამეტრით, რომელიც განსაზღვრავს დელეგატის მომსახურე მეთოდის სიგნატურას, როგორც ნაჩვენებია ქვემოთ.

```
Semdarebeli cmp = new Semdarebeli(Saxeligvari.SaxelebisSedareba);
```

ამ ოპერატორით აღიწერება Semdarebeli ტიპის cmp ცვლადი, რომელსაც ენიჭება SaxelebisSedareba მეთოდის სახელი `Saxeligvari` კლასიდან. დელეგატი *cmp* შემდგომში გამოიყენება, როგორც *Dalageba()* მეთოდის პარამეტრი, რომელიც გამოიყენება ჩვეულებრივი მეთოდის მსგავსად, რაც ნაჩვენებია ქვემოთ.

```
sd.Dalageba(cmp);
```

ამ ტექნიკის გამოყენებით შესაძლებელია დელეგატის ნებისმიერი მომსახურე მეთოდის გაგზავნა *Dalageba()* მეთოდში პროგრამის შესრულების დროს ანუ შეგიძლიათ

განსაზღვროთ მომსახურე მეთოდი სახელად *GvarisSedareba()*, შექმნათ *Semdarebeli* დელეგატის ახალი ეგზემპლარი და გააგზავნოთ *Dalageba()* მეთოდში.

ქვემოთ მოყვანილია დელეგატის გამოყენების კიდევ ორი მაგალითი. ამონაბეჭდები 14.1.1 და 14.1.2, რომლებიც უფრო ნათელს ფენენ ამ საკითხს.

ამონაბეჭდი 14.1.1.

```
using System;
// ეს არის დელეგატის დეკლარაცია
public delegate double funqciissaxeli(double x);
class martividelegati
{static void FX(funqciissaxeli F,Double x)// F –მეთოდის სახელი
        {Console.WriteLine(F(x));
        }
        static void Main()
        {// ეს არის დელეგატის დაკონკრეტება
        funqciissaxeli Fsin = new funqciissaxeli(Math.Sin);
        funqciissaxeli Fcos = new funqciissaxeli(Math.Cos);
        FX(Fsin,0.5);FX(Fcos,0.5);
        funqciissaxeli Funs = new funqciissaxeli(Math.Sin);
        Funs+=new funqciissaxeli(Math.Cos);
Console.WriteLine("Funs(0.5)={0}",Funs(0.5));//მეთოდების მიმდევრობის გამოძახება
Funs=null;
FX(new funqciissaxeli(Math.Sin),0.5);
FX(new funqciissaxeli( Math.Cos),0.5);
                double d;
                d=(new funqciissaxeli(Math.Sin))(0.5);
                Console.WriteLine("d={0}",d);
Console.WriteLine("(new funqciissaxeli(Math.Sin))(0.5)={0}",(new
funqciissaxeli(Math.Sin))(0.5));
        }
}
```

გამოსავალი:

0.479425538604203

0.877582561890373

Funs(0.5)=0.877582561890373

0.479425538604203

0.877582561890373

d=0.479425538604203

(new funqciissaxeli(Math.Sin))(0.5)=0.479425538604203

ზედა მაგალითში დელეგატი განსაზღვრავს ერთი ცვლადის ორმაგი სიზუსტის არგუმენტის მეთოდებს. მეთოდების სახელებად გამოიყენება *Math* კლასის წევრები. დელეგატის ცვლადს შეიძლება რამდენიმე ფუნქცია მივანიჭოთ: `Funs+=new funqciissaxeli(Math.Cos);`
და შემდეგ გამოვიძახოთ მიმდევრობით: `Funs(0.5)`, დაბრუნდება მხოლოდ ბოლოს მინიჭებული ფუნქციის შედეგი. *FX* მეთოდი ბეჭდავს ნებისმიერი *F* ფუნქციის მნიშვნელობას.

ამონაბეჭდი 14.1..2

```
using System;
// ეს არის დელეგატის განსაზღვრა
public delegate double funqciisdelegati(double x);
class SimpleDelegate
{       static double integrali(funqciisdelegati F,double a,double b,int nstep)
        {       double f1,f2,S,step;
                S=0;
                step=(double)((b-a)/nstep);
                f1=F(a);
                for(double x=a; x<b; x+=step)
                {  f2=F(x+step);S+= (f1 +f2)/2* step; f1=f2;}
                return S;
        }
        static double funqcia(double x)// ინტეგრალქვეშა ფუნქცია
        {       return 10*x*x+45;
        }
        static void Main()
        {       double a=0;
                double b=1;
```

```
            int nstep=100;// საწყის წერტილთა რაოდენობა
            double int2;
            double int1;
//კონკრეტული ფუნქციის განსაზღვრა და ინტეგრალის გამოთვლა
int2=integrali(new funqciisdelegati(funqcia),a, b,nstep);
            do
{       int1= int2;
                    nstep=nstep*2;
        int2=integrali(new funqciisdelegati(funqcia),a, b,nstep);
            }
while(System.Math.Abs(int1-int2)>0.0001);
            Console.WriteLine("integrali={0}",int2);
        }
}
```

გამოსავალი:

ინტეგრალი=48.3333333334092

ზედა მაგალითში განხორციელებულია a- დან b- მდე განსაზღვრული ინტეგრალის გამოთვლის მეთოდი ტრაპეციის ფორმულით. დელეგატის გამოყენებით ინტეგრალის რეალიზაცია განხორციელებულია ნებისმიერი ერთი ცვლადის F ფუნქციისთვის. გამოთვლის სიზუსტე განისაზღვრება ერთმაგ და ორმაგ წერტილთა რაოდენობაზე გათვლის სხვაობით და წერტილთა რაოდენობის გაორმაგებით.

მოვლენები (Events)

ტრადიციულად კონსოლ-აპლიკაცია (გამოყენებითი პროგრამა) შემდეგნაირად ურთიერთობს მომხმარებელთან: იგი ელოდება, სანამ დაბეჭდავთ სტრიქონს და დააჭერთ ღილაკს *enter*, შემდეგ ასრულებს პროგრამის ოპერატორებს და მთავრდება ან კვლავ ელოდება მომხმარებელს. მაგრამ ამგვარი ფუნქციონირება ხისტია და მოუქნელი, აპარატურასთან მჭიდროდ დაკავშირებული. ამისგან დიამეტრალურად განსხვავდება მომხმარებლის გრაფიკული ინტერფეისის (*GUI- graphical user interface*) პროგრამების ფუნქციონირება, რომელიც დაფუძნებულია მოვლენების (based event) მოდელზე. როცა რაიმე მოვლენა ხდება სისტემაში, ხდება ამის შეტყობინება დაინტერესებული მოდულებისადმი და ისინი შესაბამისად რეაგირებენ. *Windows Form*-ებში ეს ხდება ავტომატურად, არ გჭირდებათ გამოკითხვისა და მოლოდინის კოდის დაწერა, ის წინასწარ მომზადებულია ამ მოდელში.

C# მოვლენები კლასის წევრებია, რომლებიც ყოველთვის აქტიურდება, როდესაც შესაბამისი მოვლენა ხდება. ვინც დაინტერესებულია კონკრეტული მოვლენით, ახდენს მის

რეგისტრირებას და როდესაც ის წარმოიშვება, მას ატყობინებენ ამის შესახებ და შესაბამისი მეთოდი აქტიურდება (მოვლენის ფართოდ გავრცელებული მაგალითია მაუსის ღილაკის დაჭერა, როცა კურსორი ეკრანზე მიყვანილია ფორმის ღილაკთან).

მოვლენები და დელეგატები ერთობლივად გამოიყენება, რაც უზრუნველყოფს პროგრამის სწორ ფუნქციონირებას. იგი დასაბამს იწყებს იმ კლასში, რომელიც აღწერს მოვლენას. ნებისმიერ კლასში იმ კლასის ჩათვლით, რომელშიც აღწერილია მოვლენა, შეიძლება დარეგისტრირდეს მეთოდი მოვლენისთვის. ეს ხდება დელეგატების საშუალებით, რომლებშიც მოცემულია იმ მეთოდის სიგნატურა, რომელიც რეგისტრირდება მოვლენისთვის. დელეგატი შესაძლებელია იყოს *.NET*-ის (სისტემის) წინასწარ განსაზღვრული ან ჩვენს მიერ აღწერილი. დელეგატისადმი მოვლენის მინიჭებისას ხდება მეთოდის რეგისტრაცია. ეს მეთოდი გამოიძახება ამ მოვლენის წარმოშობისას. თვით მოვლენის გამოწვევა ხდება სისტემურად, მაგალითად, მაუსის ღილაკის დაჭერით ან მოვლენის ცვლადის გამოყენებით მსგავსად მეთოდის გამოძახებისა. ამონაბეჭდი 14-2 გვიჩვენებს მოვლენების განხორციელების 2 სხვადასხვა გზას. ერთი სისტემური, ხოლო მეორე - საკუთრივ ჩვენი დაწერილი. სხვა სიტყვებით, რომ ვთქვათ პირველი სისტემის შემქმნელების მიერ დაწერილი (რაც ჩვენთვის წინასწარ გამზადებულია) და მეორე ჩვენს მიერ დაწერილი.

ამონაბეჭდი 14-2. მოვლენების დეკლარაცია და განხორციელება: movlenebisdemonstrireba.cs

```
using System;
using System.Drawing;
using System.Windows.Forms;
// საკუთარი დელეგატი
public delegate void Sawyisidelegati();
class Movlenismsvleloba : Form
{     // საკუთარი მოვლენა
    public event Sawyisidelegati Sawyisimovlena;
    public Movlenismsvleloba()
    {   Button damaWire = new Button();
        damaWire.Parent = this;
        damaWire.Text = "დააჭირე!";
  damaWire.Location = new Point((ClientSize.Width - damaWire.Width) /2,
   (ClientSize.Height - damaWire.Height)/2);
// EventHandler დელეგატი მიენიჭა ღილაკს Click მოვლენას OnClickMeClicked მეთოდის
სახით
        damaWire.Click += new EventHandler(OnClickMeClicked);
       // საკუთარი დელეგატი "Sawyisidelegati" მიენიჭა
   // საკუთარ მოვლენას "Sawyisimovlena"
```

```
    Sawyisimovlena += new Sawyisidelegati(Rocasawyisimovlena);
        // საკუთარი მოვლენის გაშვება
        Sawyisimovlena();
    }
    // ეს მეთოდი გამოიძახება როცა დააჭერთ " დამაჭირე "  ღილაკს
    public void OnClickMeClicked(object sender, EventArgs ea)
    {   MessageBox.Show("დააჭირეთ!");
    }
    // ეს მეთოდი გამოიძახება როცა გაიშვება " Sawyisimovlena "
    public void Rocasawyisimovlena()
    {   MessageBox.Show("მე დავიწყე!");
    }
    static void Main()
    {   Application.Run(new Movlenismsvleloba());
    }
}
```

შეამჩნევდით, რომ ამონაბეჭდში 14-2 გამოიყენება Windows ფორმა. თუმცა ამ სახელძღვანელოში არ იყო განხილული ფორმები (რაც პრინციპში ფორმის დიზაინით vizual studio net-ში ავტომატურად მიიღება პროგრამირების გარეშეც) ჩვენ უკვე საკმაოდ ვიცით C# შესახებ, რომ გავარჩიოთ ეს მაგალითი. აქ *Movlenismsvleloba* კლასი მემკვიდრეობას ღებულობს *Form* კლასიდან, რომელიც არსებითად ქმნის Windows ფორმას. ეს ავტომატურად გვაძლევს ფორმის ყველა შესაძლებლობას "Title Bar, Minimize/Maximize/Close buttons, System Menu, Borders" ჩათვლით. აქ სწორედ თავს იჩენს მემკვიდრეობის ძალა, ე. ი. თავიდან არ გვიხდება სხვის მიერ კარგად გაკეთებული საქმის კეთება.

Windows Form-ის აპლიკაციის დაწყება ხდება სტატიკური *Application* ობიექტის *Run( )* მეთოდის გამოყენებით, რომელიც მიუთითებს *Form* ტიპის (კლასის) ობიექტზე, როგორც პარამეტრზე. ეს გამოიწვევს სრულყოფილი Windows ფანჯრის შექმნას სამომხმარებლო გრაფიკული ინტერფეისით და უზრუნველყოფს მოვლენების შესაბამისად ფუნქციონირებას (ფაქტიურად მომხმარებლის ყველა ქმედება აღიქმება, როგორც მოვლენა, რომელზეც შესაძლებელია რეაგირება).

შევხედოთ ჩვენს მიერ შექმნილ მოვლენას (*event*). ქვემოთ მოყვანილია ამ მოვლენის აღწერა, რომელიც არის *Movlenismsvleloba* კლასის წევრი. იგი აღიწერება საკვანძო სიტყვით *event* , *delegate* ტიპით Sawyisidelegati და მოვლენის სახელით Sawyisimovlena.

```
public event Sawyisidelegati Sawyisimovlena;
```

პროგრამის ელემენტი, რომელიც დაინტერესებულია ამ მოვლენით ახდენს დელეგატის მიბმას მოვლენასთან. შემდეგ სტრიქონში ვხედავთ *Sawyisidelegati* ტიპის დელეგატს, რომელიც ებმება მოვლენას *Sawyisimovlena* .

Sawyisimovlena += new Sawyisidelegati(Rocasawyisimovlena);

აქ Sawyisimovlena-მოვლენის სახელია, Sawyisidelegati- დელეგატის ტიპის სახელია, Rocasawyisimovlena- იმ მეთოდის სახელია, რომელმაც უნდა დაამუშაოს ეს მოვლენა. დასარეგისტრირებლად გამოიყენება ოპერაცია "+=", რეგისტრაციის მოსახსნელად "-=".

მოვლენის გამოწვევა ხდება მეთოდის გამოძახების მსგავსად, რაც ნაჩვენებია ქვემოთ:

Sawyisimovlena();

ეს არის საკუთარი მოვლენის აღწერისა და გამოყენების მაგალითი. რა თქმა უნდა უმრავლეს შემთხვევაში გამოვიყენებთ სისტემაში არსებულ წინასწარ განსაზღვრულ მოვლენებსა და დელეგატებს, რომელთა გამოყენებაც აღწერილია ქვემოთ. ამონაბეჭდში 14-2 *EventHandler* დელეგატს გადავაბამთ *Button Click* მოვლენას.

damaWire.Click += new EventHandler(OnClickMeClicked);

მოვლენა Click აგრეთვე ეკუთვნის კლასს *Button*-ს. დელეგატი *EventHandler* არსებობს .NET Frameworks-ის (სისტემის დასახელება, რომელიც უზრუნველყოფს C#-ის ფუნქციონირებას) კლასების ბიბლიოთეკის *System* სახელსივრცეში. ერთადერთი, რის გაკეთებაც გვჭირდება, ეს არის საკუთარი მეთოდის განსაზღვრა, რომელიც დაამუშავებს ამ მოვლენას. მეთოდი *OnClickMeClicked()* დელეგატს *EventHander* ეთანადება თავისი სიგნატურით.

```
public void OnClickMeClicked(object sender, EventArgs ea)
{       MessageBox.Show("დააჭირეთ!");
}
```

ყოველთვის, როცა *damaWire* ღილაკის დაჭერისას (თაგვის (mouse-ის) მარცხენა ღილაკით) იგი იწვევს *Click* მოვლენას, რომელიც გააქტიურებს *OnClickMeClicked()* მეთოდს. კლასი *Button* ზრუნავს *Click* მოვლენის გააქტიურებაზე და ჩვენგან არავითარი ძალისხმევა არ არის საჭირო. ვინაიდან წინასწარ განსაზღვრული (სისტემური) მოვლენებისა და დელეგატების გამოყენება ადვილია, ამიტომ მაქსიმალურად უნდა ვეცადოთ საჭიროებისამებრ გამოვიყენოთ არსებული მოვლენები, ვიდრე შევქმნათ საკუთარი. ამონაბეჭდში 14.3 მოყვანილია მოვლენის გამოყენების მაგალითი, რომელიც უფრო მარტივია გასაგებად.

ამონაბეჭდში 14.3. მოვლენისა და დელეგატის გამოყენება. EventDelegate.cs

```
using System;
// საკუთარი დელეგატი
public delegate void Sawyisidelegati();
   class Movlenismsvleloba1
{       // საკუთარი მოვლენა
        public event Sawyisidelegati Sawyisimovlena;// იგივე შედეგს იძლევა
        // public  Sawyisidelegati  Sawyisimovlena;EVENT-ის გარეშე
```

```
        public void Movlenismsvleloba1meTodi()
        {                   // საკუთარი დელეგატი "Sawyisidelegati" მიენიჭა
                // საკუთარი მოვლენას"Sawyisimovlena"
                Sawyisimovlena += new Sawyisidelegati(Rocasawyisimovlena1);
                Sawyisimovlena += new Sawyisidelegati(movlenismimRebi1.Rocasawyisimovlena2);
                Sawyisimovlena += new Sawyisidelegati(movlenismimRebi1.Rocasawyisimovlena3);
                Sawyisimovlena();
      Sawyisimovlena-= new Sawyisidelegati(Rocasawyisimovlena1);
}               // საკუთარი მოვლენის გაშვება
         static void Rocasawyisimovlena1()
                {Console.WriteLine("მე დავიწყე1!");
       }
                public void gaSveba()
        {       Sawyisimovlena();
        }
                } //დასასრული Movlenismsvleloba1
        class movlenismimRebi1
        {// ეს მეთოდი გამოიძახება, როცა გაიშვება " Sawyisimovlena "
                                public static void Rocasawyisimovlena2()
                {Console.WriteLine("მე დავიწყე2!");
                }
                public static void Rocasawyisimovlena3()
                {Console.WriteLine("მე დავიწყე3!");
                }
                public  void Rocasawyisimovlena4()
                {Console.WriteLine("საჯარო 1!");
                }
        }
        class mTavari
        { public static void Main()
        {       Movlenismsvleloba1 mf1=new Movlenismsvleloba1();
                mf1.Movlenismsvleloba1meTodi();
                movlenismimRebi1 mm=new movlenismimRebi1();
mf1.Sawyisimovlena-= new Sawyisidelegati(movlenismimRebi1.Rocasawyisimovlena2);
mf1.Sawyisimovlena-= new Sawyisidelegati(movlenismimRebi1.Rocasawyisimovlena3);
mf1.Sawyisimovlena+= new Sawyisidelegati(movlenismimRebi1.Rocasawyisimovlena3);
mf1.Sawyisimovlena+= new Sawyisidelegati(mm.Rocasawyisimovlena4);
                        mf1.gaSveba();
```

```
        // mf1.Sawyisimovlena();// მიდის EVENT-ის  გარეშე
        // mf1.Sawyisimovlena=null;// მიდის EVENT-ის  გარეშე
                Console.ReadLine();
        }
}
```

გამოსავალი:

მე დავიწყე1!

მე დავიწყე2!

მე დავიწყე3!

მე დავიწყე3!

საჯარო 1!

ამონაბეჭდში 14.3 შედგება 3 კლასისგან. `Movlenismsvleloba1`-რომელშიც აღწერილია მოვლენა და ხდება მისი გაშვება. ასევე დასაშვებია იქვე მისი დამუშავებაც. `movlenismimRebi1` – რომელშიც ხდება მოვლენის მიღება და დამუშავება. `mTavari` – ამ კლასში ხდება პროგრამის შესრულების დაწყება. უნდა აღინიშნოს, რომ საკვანძო სიტყვა event-ის გარეშეც პროგრამა იგივე შედეგს იძლევა. ამ სიტყვის გამოყენება უფრო დაცულს ხდის პროგრამას, კერძოდ, დაუშვებელი ხდება შემდეგი ოპერატორების ხმარება:

`mf1.Sawyisimovlena();mf1.Sawyisimovlena=null;`

ანუ მოვლენას ვერ გავუშვებთ და ვერც გავანულებთ, მისი აღმწერი კლასის გარეთ.

თავი XV

გამონაკლისი შემთხვევების დამუშავება

განხილულია შემდეგი საკითხები:

- გამონაკლისი შემთხვევის ცნება;
- ბლოკი *try/catch*;
- რესურსების გამოთავისუფლება *finally* ბლოკში.

გამონაკლისები ანუ განსაკუთრებული შემთხვევები (Exceptions)

გამონაკლისები არის გაუთვალისწინებელი შეცდომები, რომლებიც შეიძლება მოხდეს პროგრამის შესრულებისას. ჩვენ უნდა შეგვეძლოს მათი გამოვლენა და პროგრამული დამუშავება. მაგალითისთვის მომხმარებლის მიერ არაადეკვატური მონაცემების შეყვანა, ნულოვან ობიექტებზე შემოწმება, მეთოდების მიერ დაბრუნებული მონაცემების შემოწმება.

რა თქმა უნდა არის მომენტები, როდესაც წინასწარ შეუძლებელია ყველა მოსალოდნელი შეცდომის თავიდან აცილება. მაგალითად, შეუძლებელია შეყვანა-გამოყვანის შეცდომის ან მეხსიერების დასაშვები არიდან გასვლის ყველა შესაძლო შემთხვევის გათვალისწინება. ამგვარი შემთხვევები ნაკლებად ალბათურია, მაგრამ ჩვენ უნდა შეგვეძლოს მათი დამუშავება, რათა შევინარჩუნოთ პროგრამის მდგრადობა და არ მივიღოთ ოპერაციული სისტემის შეტყობინება. ასეთ შემთხვევებში გვეხმარება გამონაკლისების დამუშავების ოპერატორები და პროგრამული ბლოკები.

როდესაც გამონაკლისი მოხდება, ვამბობთ "გამოვარდება, გაგორდება, წარმოიშვება, აღმოცენდება" (thrown). ფაქტიურად გამოვარდება *System.Exception* კლასიდან წარმოებული ობიექტი. კლასი *System.Exception* უზრუნველყოფს მეთოდებსა და თვისებებს, რომლის საშუალებითაც შესაძლებელია დადგინდეს, რა იყო არასწორად. მაგალითად, თვისება *Message* შეიცავს შემაჯამებელ ინფორმაციას შეცდომის შესახებ, თვისება *StackTrace* ინფორმაციას სტეკიდან, სადაც წარმოიშვა პრობლემა, მეთოდი *ToString()* მთლიანად გამონაკლისის სიტყვიერ აღწერას.

გამონაკლისის განსაზღვრა გამომდინარეობს იმ ამოცანიდან, რომელსაც ვაპროგრამებთ. მაგალითად, თუ პროგრამა ხსნის ფაილს *System.IO.File.OpenRead()* მეთოდის გამოყენებით, მაშინ მოსალოდნელია შემდეგი სახის გამონაკლისები:

- *SecurityException* (დაცვის)
- *ArgumentException* (არგუმენტის)
- *ArgumentNullException* (ნულოვანი არგუმენტის)
- *PathTooLongException* (გზა ზედმეტად გრძელია)
- *DirectoryNotFoundException* (დირექტორი არ იქნა ნაპოვნი)
- *UnauthorizedAccessException* (არაავტორიზებული მიმართვა)
- *FileNotFoundException* (ფაილი არ იქნა ნაპოვნი)

- *NotSupportedException* (არ აქვს მხარდაჭერა)

სიაში მოყვანილი გამონაკლისების აღწერა შეიძლება ინახოს სახელსივრცე *System.IO*–ს კლასს *File* მეთოდში *OpenRead()*. როდესაც გარკვეულია, თუ რა გამონაკლისი შემთხვევა შეიძლება წარმოიშვას ამის შემდეგ გვჭირდება მექანიზმი მისი დამუშავების კოდის ჩასაწერად. დასამუშავებული გამონაკლისების სია ასევე შეიძლება დადგინდეს პროგრამის გამართვის პროცესში ეტაპობრივად. წარმოიშვა გამონაკლისი გავითვალისწინეთ, კიდევ წარმოიშვა გავითვალისწინეთ და ა.შ. პროგრამის მომხმარებელთან ჩაბარებამდე. შემდეგი შეცდომების ჩასწორება, რომლებიც გაცილებით იშვიათად წარმოიშვება, ხდება პროგრამის თანხლების პროცესში.

ბლოკები try/catch

გამონაკლისის დასამუშავებლად გამოიყენება ბლოკი *try/catch*. ვინაიდან *OpenRead()* მეთოდმა შეიძლება გამოაგდოს ზემოჩამოთვლილი გამონაკლისები, იგი უნდა მოვათავსოთ *try* ბლოკში. თუ გამონაკლისი გამოვარდა, მაშინ მისი "დაჭერა" შეიძლება *catch* ბლოკში. ამონაბეჭდში 15-1 პროგრამა ბეჭდავს შეტყობინებებს და ტრასირების ინფორმაციას სტეკიდან კონსოლზე, თუ გამონაკლისი შემთხვევა გამოვარდა. ამ შემთხვევაში გამონაკლისის პროვოცირება ხდება გამიზნულად, სასწავლო მიზნებით, რათა დავინახოთ მის მიერ გამოწვეული შედეგები.

ამონაბეჭდი 15-1. try/catch ბლოკის გამოყენება: ScadedaiWire.cs

```
using System;
using System.IO;
class ScadedaiWire
{   static void Main()
  {     try
    {
      File.OpenRead("არარსებული ფაილის სახელი");
    }
    catch(Exception ex)
    {         Console.WriteLine(ex.ToString());
    }
  }
}
```

თუმცა ამონაბეჭდში 15-1 მხოლოდ ერთი *catch* ბლოკია. იგი ყველა გამონაკლისს მოიცავს ვინაიდან ბაზური ტიპი "*Exception*" არის გამოყენებული. გამონაკლისების დამუშავებისას უფრო სპეციფიური (წარმოებული) გამონაკლისები დამუშავდება უფრო

ადრე, ვიდრე მათი მშობელი გამონაკლისები. პროგრამის შემდეგი მონაკვეთი გვიჩვენებს, თუ რა მიმდევრობით განვათავსოთ მრავალდამჭერი catch ბლოკები:

```
catch(FileNotFoundException fnfex)
{       Console.WriteLine(fnfex.ToString());
}
catch(Exception ex)
{       Console.WriteLine(ex.ToString());
}
```

თუ ფაილი არ არსებობს, გამოვარდება (წარმოიშვება) გამონაკლისი *FileNotFoundException*, რომელსაც დაიჭერს და დაამუშავებს პირველი (ზემოდან ქვემოთ) *catch* ბლოკი. რა თქმა უნდა, თუ გამოვარდება *PathTooLongException* გამონაკლისი მას დაიჭერს და დაამუშავებს მეორე *catch* ბლოკი. ეს ხდება იმიტომ, რომ არ არის *PathTooLongException* გამონაკლისის ბლოკი და მხოლოდ ერთი არჩევანია, მისი ზემდგომი მშობლიური *Exception* ტიპი.

გამონაკლისები დამუშავებამდე თავსდება სტეკში, სანამ მათ არ დაამუშავებს ამისთვის განკუთვნილი კოდი. თუ არ არსებობს პროგრამაში შესაბამისი *try/catch* ბლოკი და მოხდა გამონაკლისი, მაშინ პროგრამის მიმდინარეობა წყდება და ვღებულობთ შეცდომის შეტყობინებას გამონაკლისის შესახებ, რაც ფრიად არასასურველია პროგრამის მომხმარებლისთვის.

დამასრულებელი (Finally) ბლოკი

გამონაკლისები ტოვებენ პროგრამას რესურსების გამოთავისუფლების გარეშე. რესურსების გამოთავისუფლება და სხვა შესასწორებელი სამუშაოები შესაძლებელია ჩატარდეს *catch* ბლოკში, მაგრამ არ არის გარანტია, რომ იგი აუცილებლად შესრულდება. ამისთვის გამოიყენება ზემოხსენებული *Finally* ბლოკი, რომელიც აუცილებლად სრულდება დამოუკიდებლად იმისა, გამოვარდა თუ არა გამონაკლისი.

ამონაბეჭდი 15-2 გვიჩვენებს ამ ბლოკის გამოყენების მაგალითს. როგორც ცნობილია, შეყვანის შემდეგ ფაილი უნდა დაიხუროს. ამ შემთხვევაში ფაილური ნაკადი არის რესურსი, რომელიც უნდა გამოთავისუფლდეს. აქ *gamomavalinakadi* იხსნება ნორმალურად და პროგრამას ხელი მიუწვდება ფაილის რესურსებამდე. როდესაც ვცდილობთ *Semavalinakadi* გახსნას, წარმოიშვება *FileNotFoundException* გამონაკლისი, რაც იწვევს უშუალოდ მართვის გადაცემას *catch* ბლოკისადმი.

მაგალითად, შესაძლებელია *gamomavalinakadi*-ის დახურვა შესაბამის *catch* ბლოკში, მაგრამ თუ გამონაკლისი არ მოხდა? წარმატებული გახსნის შემთხვევაში *catch* ბლოკი არასდროს შესრულდება და ფაილი არ დაიხურება. ამ შემთხვევის გასათვალისწინებლად ვიყენებთ *finally* ბლოკს ამონაბეჭდში 15-2. დამოუკიდებლად იმისა, გამონაკლისი შემთხვევა წარმოიშვა თუ არა, ეს ბლოკი ყოველთვის სრულდება, რის შემდეგაც მართვა გადაეცემა

მომდევნო ოპერატორებს, თუკი ასეთები არსებობენ. ჩვენ შემთხვევაში მეთოდის შესრულება მთავრდება *finally* ბლოკში.

ამონაბეჭდი 15-2. finally ბლოკის განხორციელება: Damasrulebeli.cs

```
using System;
using System.IO;
class Damasrulebeli
{   static void Main()
  { FileStream gamomavalinakadi = null;
    FileStream Semavalinakadi = null;
    try
    { gamomavalinakadi = File.OpenWrite("Saboloofaili.txt");
      Semavalinakadi = File.OpenRead("Sawyisifaili.txt");
    }
    catch(Exception ex)
    {    Console.WriteLine(ex.ToString());
    }
    finally
    {        if (gamomavalinakadi != null)
      {
        gamomavalinakadi.Close();
        Console.WriteLine("გამომავალინაკადი დაიხურა.");
      }
      if (Semavalinakadi != null)
      {            Semavalinakadi.Close();
        Console.WriteLine("შემავალინაკადი  დაიხურა.");
      }
    }  }
}
```

finally ბლოკში სავალდებულო არ არის. ჩვეულებრივ *catch* ბლოკის შემდგომი ოპერატორები სრულდება. თუ *catch* ბლოკში მოთავსებულია ოპერატორი, რომელიც გამოაგდებს იმავე ან სხვა გამონაკლის შემთხვევას, მაშინ ამ ბლოკის მომდევნო ოპერატორები აღარ სრულდება. *finally* ბლოკში მოთავსებული ოპერატორები გარანტირებულად სრულდება, როგორც წესი, ისინი კონკრეტული სიტუაციიდან გამომდინარე ახდენენ კრიტიკულ ქმედებებს.

თავი XVI

ატრიბუტების გამოყენება

განხილულია შემდეგი საკითხები:

- რა არის ატრიბუტები და რატომ გამოიყენება ისინი;
- სხვადასხვა ატრიბუტების გამოყენება პარამეტრებით და უპარამეტრებოდ;
- სხვადასხვა დონის ატრიბუტების გამოყენება *(assembly, type member type level).*

ატრიბუტები დეკლარაციული (აღწერილობითი) ელემენტებია, რომლებიც საშუალებას იძლევა დავამატოთ დეკლარაციული ინფორმაცია პროგრამაში. ეს ინფორმაცია შეიძლება გამოყენებულ იქნეს სხვადასხვა გზით. მაგალითად, თვით ამ პროგრამის შესრულებისას ან სხვა პროგრამის მიერ. მაგალითად, აპლიკაციის დიზაინერის ინსტრუმენტების (გადასაწყვეტი ამოცანის კონსტრუირების პროგრამა) ან კომპილატორის და ა. შ. მიერ. მაგალითად, ატრიბუტი *DllImportAttribute*, რომელიც იძლევა *Win32* ბიბლიოთეკასთან ურთიერთობის საშუალებას, ატრიბუტი ObsoleteAttribute უზრუნველყოფს გაფრთხილებას კომპილაციის დროს, რომ მეთოდი პროგრამული პროდუქტის შემდგომ ვერსიებში აღარ გამოიყენება.

მიზეზი იმისა, რომ ატრიბუტების გამოყენება არის საჭირო, ისაა, რომ მრავალი სერვისი, რომლებსაც ისინი იძლევიან ძნელი განსახორციელებელი იქნებოდა პროგრამულ კოდში. როგორც ხედავთ, ატრიბუტები პროგრამებს აძლევენ დამატებით მონაცემებს. როდესაც ხდება C# პროგრამის კომპილაცია, იქმნება ნაკრები (*assembly*) სახის ფაილი, რომელიც ჩვეულებრივ არის შესრულებადი ან DLL (დინამიურად დაკავშირებადი) ბიბლიოთეკა. ნაკრები არის თვითაღწერადი, ვინაიდან შეიცავს ატრიბუტებში აღწერილ მონაცემებს. ეს ინფორმაცია შესაძლებელია მოვიძიოთ პროგრამის შესრულების დროს რეფლექციის *(reflection)* პროცესის გამოყენებით. ატრიბუტები არის კლასები, რომლებიც დაწერილია და დამატებით შეიძლება დაწერილ იქნეს C# და გამოიყენება თქვენი კოდის აღწერილობითი ინფორმაციით უზრუნველსაყოფად. მაგალითად, ატრიბუტი:

```
[Author("T. BakhtaZe", version = 1.1)]
Class Author
```

შინაარსობრივად ეკვივალენტურია:

```
Author anonymousAuthorObject = new Author("T. BakhtaZe "); anonymousAuthorObject.version
= 1.1;
```

ანუ ატრიბუტის სახელი კლასის სახელია, პოზიციური პარამეტრი კონსტრუქტორის ფაქტიური არგუმენტი, ხოლო სახელდებული პარამეტრი კი - კლასის ეგზემპლარის თვისების კონკრეტული მნიშვნელობა.

ეს კონცეფცია იძლევა შესაძლებლობას გააფართოვოთ ენა ინდივიდუალური სინტაქსის აღწერის საშუალებით.

ამ თავში ნაჩვენებია, თუ როგორ გამოვიყენოთ წინასწარ განსაზღვრული ატრიბუტები C# პროგრამებში. კონცეფციის გაგება და ატრიბუტების გამოყენების მაგალითები დაგეხმარებათ საჭიროების შემთხვევაში მოიძიოთ და გამოიყენოთ სხვა ატრიბუტები .NET კლასების ბიბლიოთეკაში.

ატრიბუტების საფუძვლები

ატრიბუტები, როგორც წესი, გამოიყენება ტიპის წინ და ტიპის წევრის წინ. ისინი აღიწერება კვადრატული ფრჩხილებით **"["** და **"]".** მაგალითად:

```
[ObsoleteAttribute]
```

"Attribute" ნაწილი არასავალდებულოა, ამდენად ქვემოთ მოყვანილი ზედას ეკვივალენტურია:

```
[Obsolete]
```

როგორც შენიშნეთ, ამ შემთხვევაში ატრიბუტი მოცემულია მხოლოდ სახელით. მრავალ ატრიბუტს აქვს პარამეტრების სია, რომელთა საშუალებით აღვწერთ დამატებით მონაცემებს. ამონაბეჭდი 16.1 გვიჩვენებს *ObsoleteAttribute* ატრიბუტის გამოყენების სხვადასხვა გზებს.

ამონაბეჭდი 16-1. როგორ გამოვიყენოთ ატრიბუტი: ZiriTadiatributi.cs

```
using System;
class ZiriTadiatributi
{   [Obsolete]
   public void PirveliSewinaaRmdegeba()
   {      Console.WriteLine(" PirveliSewinaaRmdegeba() გამოძახება.");
   }
   [ObsoleteAttribute]
   public void MeoreSewinaaRmdegeba()
   {      Console.WriteLine("MeoreSewinaaRmdegeba() გამოძახება.");
   }
   [Obsolete("თქვენ მეტი აღარ შეგიძლიათ ამ მეთოდის გამოძახება.")]
   public void MesameSewinaaRmdegeba()
   {      Console.WriteLine("MesameSewinaaRmdegeba() გამოძახება.");
   }  // ამ პროგრამას ხდის COM –ისგან დაცულს
   [STAThread]
   static void Main()
   {      ZiriTadiatributi atributi = new ZiriTadiatributi();
      atributi.PirveliSewinaaRmdegeba();
      atributi.MeoreSewinaaRmdegeba();
```

```
    atributi.MesameSewinaaRmdegeba();
  }}
```

პროგრამის გამოსავალი:

PirveliSewinaaRmdegeba() გამოძახება.

MeoreSewinaaRmdegeba() გამოძახება.

MesameSewinaaRmdegeba() გამოძახება.

როგორც მაგალითიდან ჩანს, ამონაბეჭდში 16-1 *ObsoleteAttribute* ატრიბუტი გამოიყენება სხვადასხვა გზით. პირველად გამოიყენება *PirveliSewinaaRmdegeba()* მეთოდის წინ, ხოლო შემდგომ - *MeoreSewinaaRmdegeba()* მეთოდთან, რაც ნაჩვენებია ქვემოთ:

```
  [Obsolete]
  public void PirveliSewinaaRmdegeba()
  ...
  [ObsoleteAttribute]
  public void MeoreSewinaaRmdegeba()
  ...
```

ეს ორივე შემთხვევა ეკვივალენტურია, მხოლოდ გარეგნულად განსხვავდებიან ერთმანეთისგან. ატრიბუტებს შეიძლება ჰქონდეს პარამეტრები, რაც ნაჩვენებია ქვემოთ:

```
  [Obsolete("თქვენ მეტი აღარ შეგიძლიათ ამ მეთოდის გამოძახება.")]
  public void MesameSewinaaRmdegeba()   ...
```

ეს თავისებურებას მატებს *ObsoleteAttribute* ატრიბუტს, რომელიც იძლევა *ObsoleteAttribute* ატრიბუტისგან განსხვავებულ შედეგს. სამივე *ObsoleteAttribute* ატრიბუტის მოქმედების შედეგი ნაჩვენებია ქვემოთ. ეს არის კომპილატორის მიერ ამობეჭდილი გაფრთხილება:

warning CS0612: 'ZiriTadiatributi.PirveliSewinaaRmdegeba()' is obsolete

warning CS0612: 'ZiriTadiatributi.MeoreSewinaaRmdegeba()' is obsolete

warning CS0618: 'ZiriTadiatributi.MesameSewinaaRmdegeba()' is obsolete: თქვენ მეტი აღარ შეგიძლიათ ამ მეთოდის გამოძახება.

როგორც ხედავთ, `MesameSewinaaRmdegeba()` მეთოდთან გამოყენებული ატრიბუტი იძლევა კომპილატორის არასტანდარტულ გაფრთხილებას. დანარჩენი ატრიბუტები უბრალოდ სტანდარტულ შეტყობინებებს იძლევა.

ამონაბეჭდი `16-1` შეიცავს სხვა ატრიბუტსაც `STAThreadAttribute.` ეს ატრიბუტი *STA* (*Single Thread Appartment*) ხშირად არის გამოყენებული `Main()` მეთოდის წინ, რაც უჩვენებეს, რომ `C#` პროგრამა შეიძლება ურთიერთობდეს ე. წ. არამართვად `COM(Component Object Model)`კოდთან ერთ პროგრამაში მრავალი განცალკევებადი ნაკადით (ანუ ერთდროულად შეიძლება რამდენიმე ოპერატორის გაშვება მთავარი პროგრამიდან, მაგრამ თითოეულ ნაკადში ბრძანებები სრულდება მიმდევრობით). ამ

ატრიბუტის გამოყენება უფრო უსაფრთხოს ხდის პროგრამას. ქვემოთ ნაჩვენებია ***STAThreadAttribute*** **ატრიბუტის გამოყენება:**

```
[STAThread]
static void Main(string[] argumentebi)   ...
```

ატრიბუტების პარამეტრები

ატრიბუტებს ხშირად აქვთ პარამეტრები, რაც იძლევა კონკრეტულ მოთხოვნილებებზე მორგების საშუალებას. არსებობს პოზიციური და სახელდებული პარამეტრები. პოზიციური გამოიყენება იმ შემთხვევაში, როდესაც ატრიბუტის შემქმნელს სურს, რომ პარამეტრი იყოს მოთხოვნილი. რა თქმა უნდა ეს არ არის ხისტი და მტკიცე მოთხოვნა, ვინაიდან ატრიბუტს აქვს მეორე პოზიციური error პარამეტრიც, რომელიც ჩვენ გამოვტოვეთ, რაც ნაჩვენებია ამონაბეჭდში 16-1. ეს ატრიბუტი შესაძლებელია გამოყენებულ იქნეს მეორე პოზიციურ პარამეტრთან, რაც იწვევს კომპილატორის მიერ გაფრთხილების მაგივრად შეცდომის შეტყობინებას, როგორც ნაჩვენებია ქვემოთ:

```
[Obsolete("თქვენ მეტი აღარ შეგიძლიათ ამ მეთოდის გამოძახება.", true)]
public void MesameSewinaaRmdegeba()   ...
```

პოზიციურ და სახელდებულ პარამეტრებს შორის განსხვავება იმაში მდგომარეობს, რომ ეს უკანასკნელი განსაზღვრულია სახელით და ყოველთვის არჩევითია. ატრიბუტს *DllImportAttribute* აქვს როგორც პოზიციური, ასევე სახელდებული პარამეტრები, რაც ნაჩვენებია ამონაბეჭდში 16-2.

ამონაბეჭდი 16-2. პოზიციური და სახელდებული ატრიბუტების გამოყენება:

Atributisparametrebi.cs

```
using System;
using System.Runtime.InteropServices;
class Atributisparametrebi
{   [DllImport("User32.dll", EntryPoint="MessageBox")]
    static extern int MessageDialog(int hWnd, string msg, string caption, int msgType);
    [STAThread]
    static void Main(string[] argumentebi)
    {       MessageDialog(0, "გამოძახებულია MessageDialog!", "DllImport დემონსტრირება", 0);
    }}
```

პროგრამის გამოსავალი არის შეტყობინების ფანჯარა, რომელშიც ჩაწერილია:

DllImport დემონსტრირება

გამოძახებულია MessageDialog!

ატრიბუტს *DllImportAttribute* ამონაბეჭდში 16-2 აქვს ერთი პოზიციური *"User32.dll"*, და ერთი სახელდებული *EntryPoint="MessageBox"* პარამეტრი. პოზიციური პარამეტრები

ყოველთვის სახელდებულის წინ იწერება. სახელდებული პარამეტრები შესაძლებელია ნებისმიერი თანამიმდევრობით ჩავწეროთ.

ატრიბუტების გამოყენების სამიზნეები

ატრიბუტები აქამდე გამოყენებული იყო მეთოდების მიმართ, მაგრამ მათი გამოყენება ენის სხვა ელემენტებთანაც შეიძლება. სამიზნეები განსაზღვრავს ატრიბუტის მოქმედების გავრცელების საზღვრებს. ცხრილში 16-1 მოყვანილია საკვანძო სიტყვების სია, რომლებიც გამოიყენება სამიზნეებად, და მათი დანიშნულება.

ცხრილში 16-1

ატრიბუტის სამიზნე	შეიძლება გამოყენებულ იქნეს
all	ყველგან
assembly	მთელი ნაკრების მიმართ
class	კლასების მიმართ
constructor	კონსტრუქტორების მიმართ
delegate	დელეგატების მიმართ
enum	ენუმერატორების მიმართ
event	მოვლენების მიმართ
field	ველების მიმართ
interface	ინტერფეისების მიმართ
method	მეთოდების მიმართ
module	მოდულების მიმართ
parameter	პარამეტრების მიმართ
property	თვისებების მიმართ
returnvalue	დაბრუნების ტიპების მიმართ
struct	სტრუქტურების მიმართ

ატრიბუტების სამიზნე იწერება ატრიბუტის თავსართის სახით, რომელიც გამოყოფილია ":" ნიშნით.

ამონაბეჭდში 16-3 ნაჩვენებია *CLSCompliantAttribute* ატრიბუტის გამოყენების მაგალითი. ეს ატრიბუტი აიძულებს კომპილატორს დაიცვას ე. წ. *CLS (Common Language Specification)* სპეციფიკაცია.

ამონაბეჭდი 16-3. სახელდებული და პოზიციური პარამეტრების გამოყენება.: Atributissamizneebi.cs

```
using System;
[assembly:CLSCompliant(true)]
public class Atributissamizneebi
{   public void Cls_specifikaciasTanaraTavsebadi(uint nclsParam)
   {      Console.WriteLine("გამოძახებულია Cls_specifikaciasTanaraTavsebadi().");    }
   [STAThread]
   static void Main(string[] argumentebi)
   {      uint myUint = 0;
      Atributissamizneebi tgtDemo = new Atributissamizneebi();
      tgtDemo.Cls_specifikaciasTanaraTavsebadi(myUint);
   }}
```

კომპილატორის შეტყობინებები:

Argument type 'uint' is not CLS-compliant

ამონაბეჭდში 16-3 მოცემული კოდი მოგვცემს კომპილაციის შეცდომას, ვინაიდან *Cls_specifikaciasTanaraTavsebadi()* მეთოდში *uint* ტიპი არ არის გათვალისწინებული ზემოხსენებული სპეციფიკაციით (მოთხოვნათა წესები). თუ [assembly:CLSCompliant(true)] ატრიბუტში *true*-ს შევცვლით *False*-ით ან *Cls_specifikaciasTanaraTavsebadi()* მეთოდში აღწერას *uint*-ს *int*-ით, მაშინ კომპილირება წარმატებით ჩაივლის. ამონაბეჭდში 16-3 გამოყენებულია სამიზნე *"assembly"*. ეს ნიშნავს, რომ ატრიბუტის მოქმედება ვრცელდება მთელ ნაკრებზე. ამ ატრიბუტის მოქმედების ზონა შესაძლებელია შევცვალოთ სამიზნეებით class და method. პირველ შემთხვევაში გავრცელდება *Atributissamizneebi* კლასზე, ხოლო მეორე შემთხვევაში - *Cls_specifikaciasTanaraTavsebadi()* მეთოდზე.

ჩვენ შეგვიძლია შევქმნათ და გამოვიყენოთ საკუთარი ატრიბუტი. ამონაბეჭდში 16-4 მოცემულია საკუთარი ატრიბუტის შექმნისა და გამოყენების მაგალითი:

ამონაბეჭდი 16-4. ატრიბუტის აღწერა და და გამოყენება. `Attribute.cs`

```
[System.AttributeUsage(System.AttributeTargets.Class | System.AttributeTargets.Struct,
AllowMultiple = true)] // მიუთითებს, რომ avtori ატრიბუტის გამოყენება შეიძლება რამდენიმე
// კლასის  ან სტრუქტურის წინ
public class avtori : System.Attribute
{       string saxeli;
        public double versia;
        public avtori(string saxeli)
        {this.saxeli = saxeli; versia = 1.0; //ნაგულისხმები მნიშვნელობა
        }
                public string GetSaxeli() { return saxeli; }
 [avtori("T. baxtaZe")]
```

```
 private class pirveliklasi
{ /* ...*/ }
 // ავტორის ატრიბუტის გარეშე
 private class meoreklasi { }
[avtori("T. baxtaZe"), avtori("x. imnaZe", versia = 2.0)]
  private class mesameklasi
{ /* ... */}
class TestavtoriAttribute
        {  static void Main()
           { PrintavtoriInfo(typeof(pirveliklasi));
   PrintavtoriInfo(typeof(meoreklasi));
PrintavtoriInfo(typeof(mesameklasi)); }
        private static void PrintavtoriInfo(System.Type t)
         {        System.Console.WriteLine("ავტორი ინფორმაცია:{0}", t);
          System.Attribute[] attrs = System.Attribute.GetCustomAttributes(t); // //რეფლექცია
          foreach (System.Attribute attr in attrs)
                  {                    if (attr is avtori)
                  {        avtori a = (avtori)attr;
System.Console.WriteLine(" {0}, ვერსია {1:f}", a.GetSaxeli(), a.versia);
                          }} }}
}
```

გამოსავალი:

```
        ავტორი ინფორმაცია: avtori+pirveliklasi
          თ. ბახტაძე, ვერსია 1.00
        ავტორი ინფორმაცია: avtori+meoreklasi
        ავტორი ინფორმაცია: avtori+mesameklasi
          თ. ბახტაძე, ვერსია 1.00
          ხ. იმნაძე, ვერსია 2.00
```

როგორც მაგალითიდან ჩანს, საკუთარი ატრიბუტის შესაბამისი კლასი მემკვიდრეობას ღებულობს `System.Attribute` კლასიდან. საკუთარი ატრიბუტის შექმნისას ასევე აუცილებელია `System.AttributeUsage` ატრიბუტის, `AttributeTargets` ჩამოთვლის და `AllowMultiple` თვისების გამოყენება. *Main* მეთოდში ნაჩვენებია საკუთარი ატრიბუტის გამოყენება, ე.ი. თუ კლასის ავტორმა ატრიბუტის საშულებით ჩაწერა თავისი მონაცემები, ჩვენ მას წავიკითხავთ, თუ დაგვჭირდა. ***typeof-***ოპერატორი განსაზღვრავს არგუმენტის ტიპს პროგრამის შესრულების მომენტში. *is*-ოპერატორი გამოიყენება გამოსახულების კონკრეტული ტიპისადმი თავსებადობის დასადგენად. შედეგი ჭეშმარიტია თავსებადობის შემთხვევაში.

თავი XVII

ჩამოთვლები (Enums)

განხილულია შემდეგი საკითხები:

- ჩამოთვლის (ენუმერაციის) ცნება;
- ახალი ენუმერაციული ტიპის შექმნა;
- ენუმერაციის გამოყენება;
- *System.Enum* ტიპის მეთოდების გაცნობა.

Enums-ის განსაზღვრა

Enums არის მკაცრად ტიპიზირებული კონსტანტანტების ჯგუფები. იგი გამორჩეულია იმით, რომ საშუალებას იძლევა სიმბოლურ სახელებს მივანიჭოთ მთელი მნიშვნელობანი. C#-ის ტრადიციის მიხედვით იგი მკაცრად ტიპიზირებულია, რაც იმას ნიშნავს, რომ სხვადასხვა ენუმერაციის ტიპის კონსტანტებს გარდაქმნის გარეშე ვერ მივანიჭებთ ერთმანეთს, ქვემდებარე ელემენტების ტიპები ერთნაირი რომც იყოს. მთელი ტიპის და ენუმერაციული ტიპის მინიჭებაც აშკარა გარდაქმნების გარეშე შეუძლებელია. ენუმერაციის გამოყენება პროგრამას აძლევს უფრო თვალსაჩინო სახეს, ვინაიდან ციფრების მაგივრად გამოიყენება შინაარსობრივი დატვირთვის მქონე სახელები (მაგალითად, "ჩრდილოეთი", "სამხრეთი", "აღმოსავლეთი" და "დასავლეთი" სიტყვების გამოყენება უფრო მრავლის მთქმელია, ვიდრე ციფრები 0, 1, 2, 3). გარდა ამისა, მათი მნიშვნელობის უნებლიედ შეცვლა რთულია, რაც ზრდის პროგრამის საიმედოობას. *Enums* არის მნიშვნელობის ტიპი, რაც ნიშნავს, რომ იგი შეიცავს მხოლოდ საკუთარ მნიშვნელობას და არაფერს არც ღებულობს და არც გასცემს მემკვიდრეობით და მისი ასლის მინიჭება შეიძლება ამ ტიპის სხვა კონსტანტისთვის. ქვემოთ გამოიყენება ორი საკვანძო სიტყვა *enum* და *Enum*. პირველი ეკუთვნის C# ტიპს, ხოლო მეორე - BCL (Base Class Library, ძირითადი კლასების ბიბლიოთეკა) ტიპს. პირველი (*enum)* მემკვიდრებას იღებს მეორისგან (*Enum*). პირველს ვიყენებთ ახალი ტიპის აღსაწერად, ხოლო მეორეს - სტატიკურ მეთოდებთან სამუშაოდ.

Enum-ის შექმნა

.NET Framework-ის BCL-ი შეიცავს BCL მრავალ ენუმერაციულ ტიპს, რომლებიც თავიანთი წარმოებული კლასებით აღწერილია დოკუმენტაციაში. მაგალითად, ყოველთვის, როდესაც ათავსებთ *MessageBox*-ში იკონას, ხედავთ *MessageBoxIcon* ენუმერაციას.

ამონაბეჭდში 17-1 მოცემულია ენუმერაციის გამოყენების მაგალითი. *enum* ტიპი გამოიყენება *switch* ოპერატორში. 0, 1, 2 ციფრების მაგივრად *switch* ოპერატორში

გამოიყენება *Xma* ენუმერაციული ტიპი, რაც პროგრამას უფრო თვალსაჩინოსა და დაცულს ხდის.

ამონაბეჭდი 17-1. ენუმერაციის შექმნა და გამოყენება: EnumSwitch.cs

```
using System;
public enum Xma
{ Dabali,
  SaSualo,
  MaRali
}
class EnumSwitch
{  static void Main()
  {    // შექმნა და ინიციალიზაცია
    Xma Cemixma = Xma.SaSualo;
    // გადაწყვეტილების მიღება ენუმერაციული ტიპის მნიშვნელობით
    switch (Cemixma)
    {    case Xma.Dabali:
         Console.WriteLine("ხმამ დაიწია.");
         break;
       case Xma.SaSualo:
         Console.WriteLine("ხმა საშუალოა.");
         break;
       case Xma.MaRali:
         Console.WriteLine("ხმამ აიწია.");
         break;
    }
    Console.ReadLine();
  }
}
```

ამონაბეჭდი 17-1 შეიცავს *enum* ტიპის აღწერას. ტიპის სახელია *Xma,* რომელსაც ფიგურულ ფრჩხილებში მოსდევს მნიშვნელობათა სია. ტიპი *Xma* გამოიყენება *Cemixma* ცვლადის აღსაწერად *Main* მეთოდში. ვინაიდან *enum* არის მნიშვნელობის ტიპი, მისი მნიშვნელობა (*Xma.SaSualo*) შესაძლებელია მივანიჭოთ სხვა მნიშვნელობის (*int* და *double)* ტიპების მსგავსად. ვინაიდან *Cemixma* ცვლადი აღწერილია და ინიციალიზირებული, მისი გამოყენება შეიძლება *switch* ოპერატორში. ყოველი *case* წინადადება წარმოადგენს *Xma* ცვლადის უნიკალურ მნიშვნელობას. სრულად კვალიფიცირებული სახელის გამოყენება გაუგებრობის თავიდან აცილების გარანტიაა.

ენუმერაციის გამოყენება

ამონაბეჭდში 17-1 მოყვანილი ენუმერაცია შესაძლებელია შეცვლილ იქნეს საბაზო ტიპისა და წევრების მნიშვნელობის შეცვლის გზით. თანდაყოლილად *enum* ტიპის ქვემდებარე ტიპი არის *int*. ენუმერაციის ტიპის აღწერისას შესაძლებელია საბაზო ტიპის შეცვლა, მაგალითად, მეხსიერების დაზოგვის მიზნით. სხვა მიზეზი შეიძლება იყოს ქვემდებარე ტიპის შეცვლა ამოცანიდან გამომდინარე, რათა ტიპებს შორის ცხადად გარდაქმნისას არ დაიკარგოს სიზუსტე. დასაშვები საბაზო ტიპებია: *byte, sbyte, short, ushort, int, uint, long,* და *ulong*.

შემდეგი მოდიფიკაცია, რომელიც შესაძლებელია ჩავატაროთ არის ტიპის ნებისმიერი წევრის მნიშვნელობის შეცვლა. თანდაყოლილად *enum* ტიპის პირველი წევრის მნიშვნელობა ნულია. თუ ეს არ გაკმაყოფილებთ, იგი შეიძლება შეიცვალოს სხვა მნიშვნელობით, ასევე სხვა წევრების შემთხვევაშიც. მიუნიჭებელი წევრების მნიშვნელობა ავტომატურად განისაზღვრება, როგორც ერთით მეტი წინა წევრის მნიშვნელობაზე. ამონაბეჭდში 17-2 ნაჩვენებია, თუ როგორ მოვახდინოთ საბაზო ტიპისა და წევრების მნიშვნელობის შეცვლა.

ამონაბეჭდი 17-2. Enum –ის ბაზური ტიპის და წევრების განსაზღვრა:

CamonaTbazadawevrebi.cs

```
using System;
// declares the enum
public enum Xma : byte
{   Dabali = 1,
    SaSualo,
    MaRali
}
class CamonaTbazadawevrebi
{   static void Main()
    {   Xma Cemixma = Xma.Dabali;
        switch (Cemixma)
        {     case Xma.Dabali:
                Console.WriteLine("ხმამ დაიწია.");
                break;
          case Xma.SaSualo:
                Console.WriteLine("ხმა საშუალოა.");
                break;
          case Xma.MaRali:
```

```
            Console.WriteLine("ხმამ აიწია.");
            break;
        }
        Console.ReadLine();
    }
}
```

ენუმერაცია *Xma* ამონაბეჭდში 17-2 გვიჩვენებს, თუ როგორ შევცვალოთ საბაზო ტიპი და წევრების მნიშვნელობანი. საბაზო ტიპის შეცვლა *byte* ტიპად ხდება ":" შემდეგ საკვანძო სიტვის *byte* გამოყენებით (*: <type>*). ეს იძლევა იმის გარანტიას, რომ *Xma* ტიპის ქვემდებარე წევრების ტიპი მხოლოდ *byte* შეიძლება იყოს. პირველი *Dabali* წევრის მნიშვნელობა იცვლება 1-ზე. მსგავსი სინტაქსი *<member> = <value>* შესაძლებელია გამოყენებულ იქნეს ნებისმიერი წევრის მიმართ. შეზღუდვა მდგომარეობს წინმსწრებ მითითებაში (ანუ, როცა განმსაზღვრელს ჯერ არ აქვს კონკრეტული მნიშვნელობა), მნიშვნელობათა დუბლირებაში, წრიულ მითითებებში (იგივეა, რაც პირველი შემთხვევა, ოღონდ განსაზღვრებაში მიმდევრობა მონაწილებს, რომლის მიხედვით კონკრეტული მნიშვნელობის დადგენა მაინც ვერ ხერხდება, მსგავსად *EXCEL*-ისა). *Xma* ტიპის თანდაყოლილი მნიშვნელობებია *Dabali*=0, *SaSualo*=1 და *MaRali*=2. ამონაბეჭდში 17-2 *Dabali* მნიშვნელობა შეცვლილია 1-ზე, რის გამოც *SaSualo*=2 და *MaRali*=3.

ენუმერაციული ტიპი მემკვიდრეობას არაცხადად (":" მითითების გარეშე) ღებულობს BCL ტიპიდან, ამიტომ შეგვიძლია ამ ტიპის წევრების გამოყენება. ამონაბეჭდში 17-3 ნაჩვენებია, თუ როგორ ჩავატაროთ გარდაქმნა ენუმერაციულსა და მისი ბაზური ტიპის მნიშვნელობებს შორის და როგორ გამოვიყენოთ მემკვიდრეობით მიღებული ზოგიერთი წევრი.

ამონაბეჭდი 17-3. Enum გარდაქმნები და System.Enum ტიპის გამოყენება: EnumTricks.cs

```
using System;
// enum-ის აღწერა
public enum Xma : byte
{  Dabali = 1,
   SaSualo,
   MaRali
}
class EnumTricks
{   static void Main(string[] argumentebi)
   {      EnumTricks enumTricks = new EnumTricks();
      // ტიპის ცხადი გარდაქმნა
```

```
        enumTricks.EnumisSeyvana();
        // იტერაცია saxeli-s    მიხედვით
 enumTricks.Saxelebissia();
       იტერაცია Xma -s    მიხედვით
        enumTricks.MniSvnelobebissia();
        Console.ReadLine();
    }
    // ცხადი  გარდაქმნა  int –ის  Xma-ში
    public void EnumisSeyvana()
    {   Console.WriteLine("\n----------------");
        Console.WriteLine("ხმის  პარამეტრები:");
        Console.WriteLine("----------------\n");

        Console.Write(@"
1 - Dabali
2 - SaSualo
3 - MaRali

Please select one (1, 2, or 3): ");
        // მომხმარებლის  მონაცემის  შეყვანა
        string volString = Console.ReadLine();
        int volInt = Int32.Parse(volString);
          // ცხადი  გარდაქმნა  int –ის  Xma-ში
        Xma Cemixma = (Xma)volInt;
        Console.WriteLine();
        switch (Cemixma)
        {     case Xma.Dabali:
              Console.WriteLine("ხმამ  დაიწია.");
              break;
           case Xma.SaSualo:
              Console.WriteLine("ხმა  საშუალოა.");
              break;
           case Xma.MaRali:
              Console.WriteLine("ხმამ  აიწია.");
              break;
        }
        Console.WriteLine();
    }
```

```
  public void Saxelebissia()
  {  Console.WriteLine("\n---------------------------- ");
     Console.WriteLine("Xma Enum Members by Saxeli:");
     Console.WriteLine("----------------------------\n");
     // იტერაცია xma ტიპის სიის წევრებზე
     foreach (string xma in Enum.GetNames(typeof(Xma)))
     {    Console.WriteLine("Xma Member: {0}\n Value: {1}",
          xma, (byte)Enum.Parse(typeof(Xma), xma));
     }
  }
  // იტერაცია მნიშვნელობის მიხედვით
  public void MniSvnelobebissia()
  {  Console.WriteLine("\n---------------------------- ");
     Console.WriteLine("Xma Enum წევრები მნიშვნელობის მიხედვით:");
     Console.WriteLine("----------------------------\n");
     // ყოველი რიცხვითი მნიშვნელობის მიღება და გამოყვანა
     foreach (byte val in Enum.GetValues(typeof(Xma)))
 {  Console.WriteLine("Xma Value: {0}\n Member: {1}", val, Enum.GetName(typeof(Xma), val));
     }
  }
}
```

ამონაბეჭდი 17-3 შეიცავს სამ მეთოდს *EnumisSeyvana*, *Saxelebissia* და *MniSvnelobebissia*. თითოეული მათგანი გვიჩვენებს *System.Enum* გამოყენების სხვადასხვა თვალსაზრისს ჩამონათვალთან მუშაობისას. მეთოდი *EnumisSeyvana* გვიჩვენებს, თუ როგორ შევიყვანოთ მთელი რიცხვი და გარდავქმნათ შესაბამის ჩამოთვლის ტიპად. ქვემოთ მოყვანილია ამონაბეჭდში 17-3 ამოღებული ნაწილი, რომელიც ამ გარდაქმნას ახორციელებს:

```
     string volString = Console.ReadLine();
     int volInt = Int32.Parse(volString);      Xma Cemixma = (Xma)volInt;
```

მას შემდეგ, რაც პროგრამა გვიჩვენებს მენიუს, იგი შეახსენებს მომხმარებელს, რომ შეიყვანოს რომელიმე რიცხვთაგანი (1, 2, ან 3). მომხმარებლის მიერ აკრეფილი რიცხვი შეყვანის ღილაკის დაჭერის შემდეგ შეიყვანება *Console.ReadLine* ოპერატორით, რომელიც აბრუნებს სტრიქონის ტიპის მნიშვნელობას. შემდგომ საჭიროა ამ მნიშვნელობის გარდაქმნა მთელი ტიპის მნიშვნელობად, რაც ხორციელდება *Int32.Parse* მეთოდით. *int* ტიპის *Xma* ტიპად გარდაქმნა ხდება ტიპების ცხადი გარდაქმნის გამოყენებით მინიჭებისას. ტიპის ყოველი წევრის მისაღებად გამოიყენება *System.Enum*-ტიპის *GetNames* მეთოდი. ამონაბეჭდში 17-3 ამოღებულია შესაბამისი ტექსტი, რომელიც ზემოთქმულს ახორციელებს:

```
foreach (string xma in Enum.GetNames(typeof(Xma)))
{     Console.WriteLine("Xma Member: {0}\n Value: {1}",
      xma, (byte)Enum.Parse(typeof(Xma), xma));
}
```

ვინაიდან *GetNames* აბრუნებს სტრიქონების მასივს, ამიტომ მოსახერხებელია *foreach* ციკლის გამოყენება. ზემომოყვანილ მაგალითში მოცემულია ტიპი და წევრის სახელის სტრიქონული წარმოდგენა. ჩვენ ვიყენებთ მეთოდს ქვემდებარე წევრის მნიშვნელობის მისაღებად. ვინაიდან *Xma* ტიპის საბაზო ტიპი არის *byte*, ამიტომ *Enum.Parse (ეს მეთოდი გამოიყენება სტრიქონული ტიპის სხვა ტიპად გარდასაქმნელად, format მეთოდის შებრუნებულია)* მეთოდის მიერ დაბრუნებული მნიშვნელობა საჭიროა გარდაიქმნას ცხადად. რომ არ გაგვეკეთებინა (*bye*) გარდაქმნა, გამოსავალი იქნებოდა *Xma* ტიპის წევრი, რომელიც გარდაიქმნებოდა *Xma* წევრის სახელის სტრიქონულ წარმოდგენად და მოსალოდნელ შედეგს ვერ მივიღებდით.

ჩამოსათვლელი ტიპის ყველა წევრის სახელების მაგივრად შეიძლება დაგვჭირდეს ყველა მნიშვნელობის მიღება. ზემოთქმულს ახორციელებს ამონაბეჭდში 17-3 *MniSvnelobebissia* მეთოდში ქვემოთ მოყვანილი კოდი:

```
foreach (byte val in Enum.GetValues(typeof(Xma)))
 {Console.WriteLine("Xma Value: {0}\n Member: {1}",val, Enum.GetName(typeof(Xma), val));
}
```

მოცემულია ჩამოსათვლელი ტიპი, *GetValues* მეთოდი აბრუნებს ამ ტიპით განსაზღვრულ საბაზო მნიშვნელობებს მასივის სახით (ამ შემთხვევაში *byte* ტიპის). პროგრამული ციკლის დროს იბეჭდება თითოეული წევრის მნიშვნელობა და სახელი. სახელების მიღება ხდება *GetName* მეთოდის გამოყენებით. საკვანძო სიტყვა *typeof* ადგენს ფრჩხილებში მოთავსებული გამოსახულების ტიპს.

თავი XVIII

ოპერაციათა გადატვირთვა

განხილულია შემდეგი საკითხები:

- ოპერაციათა გადატვირთვის ცნება;
- გადატვირთვის მიზანშეწონილობის განსაზღვრა;
- როგორ გადავტვირთოთ ოპერატორი;
- ოპერაციების გადატვირთვის წესები.

ოპერაციათა გადატვირთვის შესახებ

მე-2 თავში განხილული იყო ოპერაციები ”+" ,"-" ,"^" და სხვა. ოპერაციები განსაზღვრულია თანდაყოლილი (წინასწარ განსაზღვრული) ტიპებისთვის, მაგრამ ეს ყველაფერი არ არის. შეგიძლიათ დაამატოთ ოპერაციები თქვენი საკუთარი ტიპებისთვის, რაც საშუალებას მოგცემთ გამოიყენოთ ისინი მსგავსად თანდაყოლილი (სტანდარტული) C# ოპერაციებისა.

ოპერაციათა გადატვირთვის საჭიროების გასაგებად წარმოიდგინეთ, რომ გჭირდებათ მათემატიკური მატრიცული ოპერაციები. შეგიძლიათ გამოიყენოთ ორი ორგანზომილებიანი მატრიცა და გააკეთოთ, რაც გჭირდებათ. ამ კოდის სხვა პროგრამაში გამოსაყენებლად შესაძლებელია ახალი ტიპის შექმნა. ამგვარად შექმნით მატრიცის ტიპს, რომელიც იქნება კლასი ან სტრუქტურა.

ახლა განვიხილოთ, თუ როგორ გამოვიყენებთ ამას. შექმნით ორ Matrix ტიპის ეგზემპლარს მონაცემებით და მოახდენთ მათზე ოპერაციებს. მოახდენთ *Daamate(), DotProduc()* მეთოდების რეალიზაციას. კლასის გამოყენება შეიძლება გამოიყურებოდეს შემდეგნაირად:

Matrix result = mat1.Daamate(mat2);
ან

Matrix result = Matrix.Daamate(mat1, mat2);
ან უფრო უარესად

Matrix result = mat1.DotProduct(mat2).DotProduct(mat3);

მეთოდების გამოყენებისას პრობლემა მდგომარებს იმაში, რომ გამოსახულებები ტლანქია, ჭარბსიტყვიანი და არაბუნებრივი იმ პრობლემის მიმართ, რომელსაც ვხსნით. გაცილებით ადვილი იქნებოდა მატრიცების შესაკრებად გამოგვეყენებინა "+", ხოლო გადასამრავლებლად "*" ოპერაციები. ქვემოთ ნაჩვენებია თუ, როგორ გამოიყურება ეს ამ ოპერაციების გამოყენებით:

Matrix result = mat1 + mat2;
ან

Matrix result = mat1 * mat2;
ან უფრო უკეთესად

Matrix result = mat1 * mat2 * mat3 * mat4;

ეს გაცილებით დახვეწილია და იოლი სამუშაოდ, განსაკუთრებით მაშინ, როდესაც ამგვარი ოპერაციები მრავლად გამოიყენება მათემატიკურ გამოსახულებებში. გარდა ამისა მათემატიკოსებისთვის ინტუიციურად უფრო ადვილად აღსაქმელი და ბუნებრივი.

ძირითადი რჩევა მდგომარეობს შემდეგში: გამოიყენეთ ოპერაციები, როდესაც ისინი უფრო გასაგებს და მარტივს ხდის ტიპის გამოყენებას.

ოპერაციათა გადატვირთვის აღწერა

რამდენიმე გამონაკლისის გარდა ეს მსგავსია სტატიკური მეთოდის აღწერის სინტაქსისა. თქვენ უნდა გამოიყენოთ საკვანძო სიტყვა *operator* და გადასატვირთი ოპერატორის სიმბოლო. ქვემოთ მოყვანილია მონახაზი სკალარული ნამრავლის განსახორციელებლად:

```
public static Matrix operator *(Matrix mat1, Matrix mat2)
{
   // რეალიზაცია
}
```

შევნიშნოთ, რომ მეთოდი სტატიკურია. აქ Matrix, რომელიც მოსდევს operator-ს ოპერაციის დაბრუნების ტიპია, მრგვალ ფრჩხილებში მოთავსებულია ოპერაციის ოპერანდები თავიანთი ტიპებით. ამონაბეჭდში 18-1 მოყვანილია სრული მაგალითი.

ამონაბეჭდი 18-1. ოპერაციათა გადატვირთვის გამოყენება: Matrix.cs

```
using System;
class Matrica
{   // ეს არის 3x3 მატრიცა
   public const int GANZOMILEBA = 3;
   private double[,] matrix = new double[GANZOMILEBA, GANZOMILEBA];
   // ნებას აძლევს გამომძახებელს მოახდინოს მატრიცის ინიციალიზაცია
   public double this[int x, int y]
   {  get { return matrix[x, y]; }
      set { matrix[x, y] = value; }
   }
   // ნებას აძლევს მომხმარებელს მოახდინოს მატრიცის დამატება
   public static Matrica operator +(Matrica mat1, Matrica mat2)
   {  Matrica newMatrix = new Matrica();
```

```
      for (int x=0; x < GANZOMILEBA; x++)
         for (int y=0; y < GANZOMILEBA; y++)
            newMatrix[x, y] = mat1[x, y] + mat2[x, y];
      return newMatrix;
   }
}
class MatrixTest
{   //გამოიყენება  InitMatrix მეთოდში
   public static Random rand = new Random();
   //  Matrica ტესტირება
   static void Main()
   {  Matrica mat1 = new Matrica();
      Matrica mat2 = new Matrica();
      // მატრიცების შემთხვევითი სიდიდეებით ინიციალიზაცია
      InitMatrix(mat1);
      InitMatrix(mat2);
     //  მატრიცების ბეჭდვა
      Console.WriteLine("Matrix 1: ");
      MatricisbeWdva(mat1);
      Console.WriteLine("Matrix 2: ");
      MatricisbeWdva(mat2);
      //  ოპერაციების შესრულება და შედეგების ბეჭდვა
      Matrica mat3 = mat1 + mat2;
      Console.WriteLine();
      Console.WriteLine("Matrix 1 + Matrix 2 = ");
      MatricisbeWdva(mat3);
   }
   // მატრიცის ელემენტების შევსება შემთხვევითი მნიშვნელობებით
   public static void InitMatrix(Matrica mat)
   {       for (int x=0; x < Matrica.GANZOMILEBA; x++)
         for (int y=0; y < Matrica.GANZOMILEBA; y++)
 mat[x, y] = rand.NextDouble();  //NextDouble()  შემთხვევითი რიცხვების გენერირების ფუნქცია
   }
   //   მატრიცის ბეჭდვა კონსოლზე
   public static void MatricisbeWdva(Matrica mat)
   {   Console.WriteLine();
      for (int x=0; x < Matrica.GANZOMILEBA; x++)
      {  Console.Write("[ ");
```

```
        for (int y=0; y < Matrica.GANZOMILEBA; y++)
        {   // გამოსავალის ფორმატი
          Console.Write("{0,8:#.000000}", mat[x, y]);
          if ((y+1 % 2) < 3)
            Console.Write(", ");
        }
        Console.WriteLine(" ]");
      }
      Console.WriteLine();
    }
}
```

სკალარული ნამრავლის ოპერაციის მსგავსად ამონაბეჭდში 18-1 მოცემულია "+" ოპერაციის გადატვირთვა. უფრო მეტი თვალსაჩინოებისთვის ამ ოპერაციის გადატვირთვა გამოტანილია ცალკე:

```
public static Matrica operator +(Matrica mat1, Matrica mat2)
{  Matrica newMatrix = new Matrica();
   for (int x=0; x < GANZOMILEBA; x++)
      for (int y=0; y < GANZOMILEBA; y++)
         newMatrix[x, y] = mat1[x, y] + mat2[x, y];
   return newMatrix;
}
```

ოპერაცია არის *static*, რაც ხაზს უსვამს იმ გარემოებას, რომ ოპერაცია მიეკუთვნება ტიპს და არა რომელიმე ეგზემპლარს. დაბრუნების ტიპია *Matrica*. ოპერაციის რეალიზაცია ქმნის დაბრუნების ტიპის ახალ ეგზემპლარს (ობიექტს) და აწარმოებს მატრიცების შეკრებას.

ძირითადად ეს ადვილია, უბრალოდ უნდა დავიცვათ რამდენიმე წესი, როდესაც ვახორციელებთ ოპერაციების გადატვირთვას.

ოპერაციების გადატვირთვის წესები

პირველი წესი: უნდა მოახდინოთ იმ ოპერაციის გადატვირთვა იმ ტიპში, რომელსაც გამოიყენებთ ოპერაციაში ოპერანდის ტიპად.

მეორე წესი: უნდა გადატვირთოთ ყველა თავსებადი ოპერაცია. მაგალითად, თუ ტვირთავთ ==, ასევე უნდა გადატვირთოთ !=. იგივე ეხება <= და >=.

როდესაც აღწერთ რაიმე ოპერაციას, მისი შედგენილი ვარიანტის გამოყენებაც შეიძლება. მაგალითად, + ოპერაციის აღწერის შემდეგ *Matrica* ტიპისთვის შესაძლებელია += ოპერაციის გამოყენება იმავე *Matrica* ტიპის მონაცემებისთვის.

ოპერაციების გადატვირთვის გამოყენება ტიპების ცხადი და არაცხადი გარდაქმნისთვის

ოპერაციათა გადატვირთვა შესაძლებელია გამოყენებულ იქნეს ცვლადების მნიშვნელობის საკუთარი გარდამქმნელების შესაქმნელად მინიჭების დროს. ამონაბეჭდში 18-2 მოყვანილია შესაბამისი მაგალითი:

ამონაბეჭდი 18-2. ტიპის გარადაქმნის ოპერაციის აღწერა და გამოყენება

cxadigardaqmna.cs

```
using System;
namespace cxadigardaqmna
{  class  Wiladi
            {int x, y;
        public static implicit operator Wiladi (int x) { return new Wiladi(x, 1); }
         public static explicit operator int (Wiladi f) { return f.x / f.y; }
                public Wiladi (int x, int y) {this.x = x; this.y = y; }
                     static void Main(string[] argumentebi)
            {Wiladi  f = 3;// არაცხადი  გარდაქმნა,f.x  ==  3,  f.y  ==  1
                          int i = (int)f; //  ცხადი  გარდაქმნა,i ==  3
                          System.Console.WriteLine("{0},{1}",f,i);
            }
            }
        }
```

ამონაბეჭდში 18-2 *implicit* და *explicit* საკვანძო სიტყვებია. კლასში Wiladi აღწერილია ტიპის გარდაქმნის ცხადი და და არაცხადი ოპერაციები, შესაბამისად. *Main*-მეთოდში მოყვანილია არაცხადი და ცხადი გარდაქმნების მაგალითები.

თავი XIX

დაუცველი კოდი

მოცემულ თავში განხილულია შემდეგი საკითხები:

- პოინტერების მოკლე მიმოხილვა და გამოყენება;
- პოინტერული ნოტაცია;
- დაუცველი კოდი;
- მასივების პოინტერები.

პოინტერების გამოყენება გამართლებულია მხოლოდ მაშინ, როდესაც გვჭირდება პროგრამის შესრულების ძალზე მაღალი სიჩქარე.

პოინტერების ჩაწერის წესები

პოინტერები არის ცვლადები, რომელთა მნიშვნელობა არის სხვა ტიპის ცვლადის მისამართი. ისტორიულად ამ ტიპის ცვლადები გამოიყენებოდა ობიექტზე არაორიენტირებულ პროგრამირების ენებში. პოინტერის ტიპის ცვლადები შეიძლება გამოყენებულ იქნეს მხოლოდ მნიშვნელობისა და მასივის ტიპის ცვლადების მიმართ. პოინტერის ტიპის ცვლადების აღწერა ხდება ცხადად * სიმბოლოს გამოყენებით, როგორც ნაჩვენებია მომდევნო მაგალითში:

int *p; ეს ეკვივალენტურია : int* p; ანუ განმთითებელი სიმბოლო * შესაძლებელია უშუალოდ ტიპის დასახელებას მოსდევდეს. ეს აღწერს პოინტერს p, რომელიც უთითებს მთელი ტიპის ცვლადის მეხსიერების დასაწყისზე, რომელსაც უკავია 4 ბაიტი. ოპერატორებში ცვლადის წინ * სიმბოლოს გამოყენება გულისხმობს იმ მეხსიერების მნიშვნელობას, რომელსაც უთითებს p. მაგალითად:

*p = 5; ნიშნავს, რომ p-ს მნიშვნელობის მეხსიერების მისამართზე ჩაიწერება მთელი მნიშვნელობა 5. p = 5; კი ნიშნავს, რომ იცვლება p-ს მნიშვნელობა ანუ მთელი ტიპის ცვლადის მეხსიერების მისამართი. ამ ოპერატორის შესრულების შემდეგ ეს ცვლადი მიუთითებს მე-5 მისამართზე, სადაც ინახება მთელი ტიპის 4 ბაიტი. პირველი ოპერატორის შესრულების შედეგი არ იცვლება.

შემდეგი მნიშვნელოვანი სიმბოლო არის &, მისი გამოყენება ცვლადის წინ იძლევა ამ ცვლადის მეხსიერების მისამართს. მაგალითად:

```
int i = 5;
int *p;
p = &i;
```

ზემოთ მოცემული კოდის შემთხვევაში:

```
*p = 10;
```

ცვლის i-ს მნიშვნელობას 10-ზე, ვინაიდან '*p' იკითხება, როგორც მთელი რიცხვი, რომელიც მოთავსებულია p მისამართზე.

დაუცველი კოდი

C# -ში პოინტერების გამოყენების მთავარი პრობლემა მდგომარეობს იმაში, რომ პროგრამის შესრულებისას წარმოებს არასაჭირო მონაცემებისგან მეხსიერების განთავისუფლების უხილავი პროცესი (*garbage collection process*). ამ დროს მონაცემების გადაჯგუფებისას შესაძლებელია შეიცვალოს ობიექტის მისამართი მეხსიერებაში გაფრთხილების გარეშე. ამგვარად, რაიმე პოინტერი შეიძლება აღარ მიუთითებდეს საჭირო ობიექტს. ამ შემთხვევაში შესაძლებელია არასწორად იფუნქციონიროს თვით ამ პროგრამამ ან გავლენა მოახდინოს სხვა პროგრამების მთლიანობაზე. ამის გამო პოინტერების გამოყენებისას აშკარად საჭიროა საკვანძო სიტყვა *unsafe* მითითება. პროგრამები, რომლებიც შეიცავს დაუცველ კოდს, გაიშვება მხოლოდ იმ შემთხვევაში, თუ მათ მინიჭებული აქვთ სრული ნდობის სტატუსი. იმისთვის, რომ არ შეიცვალოს პოინტერის მნიშვნელობა უხილავი ფონური პროცესით, გამოიყენება საკვანძო სიტყვა *fixed*. აქვე აღვნიშნოთ, რომ მნიშვნელობის ტიპთან პოინტერის გამოყენება ავტომატურად ნიშნავს მის ფიქსირებულ მნიშვნელობას. ამონაბეჭდში 19-1 მოყვანილია მეთოდი, რომელიც მონიშნულია, როგორც დაუცველი 'unsafe'. აქ არ არის საჭირო fixed გამოყენება, ვინაიდან სტრუქტურა მნიშვნელობის ტიპისაა.

ამონაბეჭდი 19-1.

```
using System;
public struct Koordinatebi
{int x;
   int y;
   unsafe public static void Main()
{      Koordinatebi c = new Koordinatebi();
       Koordinatebi *p = &c;
      {   p->y = 6;   // სტრუქტურის ელემენტებზე მითითებისთვის ორივე
         (*p).x = 5; //  ჩანაწერი დასაშვებია
      }
      Console.WriteLine(c.y);   Console.WriteLine(c.x);
      }}
```

ამონაბეჭდში 19-2 აუცილებელია 'fixed' ოპერატორის გამოყენება, ვინაიდან პოინტერი მიუთითებს ობიექტზე, რომელიც გამოცხადებულია დაუცველი კოდის გარეთ.

ამონაბეჭდი 19-2.

```
using System;
public struct Koordinatebi
{   int x;
```

```
int y;
   unsafe public static void araMain(ref Koordinatebi c)
   {fixed (Koordinatebi *p = &c)
      {   p->y = 6;
         (*p).x = 5;
 }
Console.WriteLine(c.y);
      Console.WriteLine(c.x);
   }
public static void Main()
      {Koordinatebi c = new Koordinatebi();
        araMain(ref  c); Console.WriteLine(c.y);Console.WriteLine(c.x);
      }}
```

პროგრამის გამოსავალი:

6

5

6

5

ზემომოყვანილ მაგალითებში საკვანძო სიტყვა 'unsafe' გამოყენებულია, როგორც მეთოდის მოდიფიკატორი. ქვემომოყვანილ მაგალითში იგი შეიძლება გამოყენებულ იქნეს, როგორც პროგრამული კოდის ბლოკი:

ამონაბეჭდი 19-3.

```
using System;
public static void Main()
{   unsafe
 {
Koordinatebi c = new Koordinatebi();
      [...]
   }
}
```

მასივების პოინტერები

პოინტერები შესაძლებელია გამოყენებულ იქნეს მასივების მიმართებაში, როგორც ნაჩვენებია ქვემოთ:

ამონაბეჭდი 19-4.

```
using System;
public class Tester
```

```
{   public static void Main()
   {      int[] a = {4, 5};
      mniSvnelobebisSecvla(a);
Console.WriteLine(a[0]);
Console.WriteLine(a[0]);
   }
public unsafe static void mniSvnelobebisSecvla(int[] a)
 {      fixed (int *b = a)
      {   *b = 5;
         *(b + 1) = 7;
      }
      }
}
```

პროგრამის გამოსავალი:

5

7

კომპიუტერის არქიტექტურის თავისებურებების გათვალისწინებითა და პოინტერების გამოყენებით შესაძლებელია პროგრამის წარმადობის გაზრდა. ქვემოთ მოყვანილია სწრაფი კოპირების პროგრამის მაგალითი. ამ მაგალითში გათვალისწინებულია, რომ ოთხი ბაიტის ერთდოული კოპირება ხდება უფრო სწრაფად, ვიდრე ოთხჯერ თითო-თითო ბაიტისა.

ამონაბეჭდი 19-5. სწრაფი კოპირების პროგრამა. swrafikopireba.cs

```
using System;
 class Test
{   static unsafe void Kopireba(byte[] sawyisi, int sawyisiIndeqsi,
      byte[] Sedegi, int SedegiIndeqsi, int mTvleli)
   {      if (sawyisi == null || sawyisiIndeqsi < 0 ||
         Sedegi == null || SedegiIndeqsi < 0 || mTvleli < 0)
      {  throw new ArgumentException();
      }
      int sawyisiLen = sawyisi.Length;
      int SedegiLen = Sedegi.Length;
      if (sawyisiLen - sawyisiIndeqsi < mTvleli ||
         SedegiLen - SedegiIndeqsi < mTvleli)
      {  throw new ArgumentException();
      }
```

```
        fixed (byte* pSawyisi = sawyisi, pSedegi = Sedegi)
        {   byte* ps = pSawyisi;
            byte* pd = pSedegi;
        // ერთდროულად 4x4 ბაიტის კოპირება

        for (int n =0 ; n < mTvleli/4 ; n++)
        {  *((int*)pd) = *((int*)ps);
           pd += 4;
           ps += 4;
        }
         // დარჩენილი ბაიტების კოპირება, რომლებიც 4x4- ად ვერ გაიგზავნა:
        for (int n =0; n < mTvleli%4; n++) // % განაყოფის ნაშთის ოპერაცია
        {  *pd = *ps;
           pd++;
           ps++;
        }
        }
    }

    static void Main()
    {  byte[] a = new byte[100];
       byte[] b = new byte[100];
       for(int i=0; i<100; ++i)
       a[i] = (byte)i;
       Kopireba(a, 0, b, 0, 100); Console.WriteLine("პირველი 10 ელემენტია:");
       for(int i=0; i<10; ++i)
       Console.Write(b[i] + " ");  Console.WriteLine("\n");
    }}
```

პროგრამის გამოსავალი:

```
პირველი 10 ელემენტია:
0 1 2 3 4 5 6 7 8 9
```

ამ მაგალითში გამოიყენება პოინტერები ბაიტების მასივის კოპირებისთვის *sawyisi* ცვლადიდან Sedegi ცვლადში. კომპილაციისთვის გამოიყენება /unsafe ოფცია. Visual Studio Net -ში კი საჭიროა: პროექტის სახელზე მარჯვენა კლავიშის კონტექსტური მენიუ, შემდეგ Properties=>Configuration Properties=>Build=>Allow Unsafe Code Bloks=>True.

თავი XX
მოდიფიკატორები

მოდიფიკატორები მნიშვნელოვან გავლენას ახდენენ პროგრამის მსვლელობაზე. მათი გამოყენება ინკაფსულაციის თვისებას სძენს ობიექტებს. ინკაფსულაცია გულისხმობს ინტერფეისისა და რეალიზაციის დამოუკიდებლობას, რის გამოც რეალიზაცია შეიძლება იცვლებოდეს ინტერფეისის შეუცვლელად ისე, რომ მთლიანად სისტემის ფუნქციონირება არ ირღვეოდეს. ობიექტი გარე სამყაროსთან ურთიერთქმედებს *public* ველების, თვისებისა და მეთოდების საშუალებით. გარედან შემთხვევით შეუძლებელია private ველების შეცვლა. აქამდე ძირითადად ვიყენებდით წვდომის *public* და მეთოდების მოდიფიკატორებს *static, virtual* და *override*. ისინი განხილული იყო წინა თავებში, კერძოდ, მე-9 თავში. ამ თავში უფრო დეტალურად განვიხილავთ მოდიფიკატორებთან დაკავშირებულ საკითხებს. მოდიფიკატორები გამოიყენება როგორც კლასის, ისე მისი წევრების მიმართ.

არსებობს წვდომის ანუ ხედვის 5 მოდიფიკატორი. თუ რაიმე ელემენტის, მაგალითად, ცვლადის გამოყენება შეიძლება მოცემულ ადგილას, ვამბობთ, რომ იგი ჩანს აქ ანუ ეს ადგილი ხვდება ამ ცვლადის ჭვრეტის არეში. წვდომის მოდიფიკატორები განსაზღვრავენ ელემენტების ჭვრეტის არეს. ცხრილი 20.1 ასახავს მათ დანიშნულებას. ჩვეულებრივ *public* მოდიფიკატორი გამოიყენება კლასის იმ წევრებისადმი, რომლებიც გვსურს დავინახოთ კლასის ეგზემპლარში. *protected* ვიყენებთ იმ შემთხვევაში, თუ კლასში და მის წარმოებულ კლასებში გვინდა დავინახოთ ცვლადი და არა მის ეგზემპლარში. *internal* ვიყენებთ პროექტის შექმნისას და მას ვხედავთ ყველგან მოცემულ ნაკრებში, როგორც კლასებში, ასევე ეგზემპლარებში, ხოლო ჩვენს მიერ შექმნილი კლასების ბიბლიოთეკაში ჩაწერისას, განმეორებითი გამოყენების შემთხვევაში მოცემული წევრები არ ჩანს. *protected internal* არის *internal* მოცემულ ნაკრებში და *protected* მის გარეთ. სასარგებლოა იმ შემთხვევაში, თუ ერთ კლასს ქმნით თავისი წარმოებული კლასებით და გსურთ, რომ პროექტის მონაწილემ დაინახოს ყველგან, ხოლო შემდგომმა მომხმარებელმა დაინახოს მხოლოდ წარმოებულ კლასებში. *private* გამოიყენაბა მაშინ, როდესაც კლასის გარეთ არ გვსურს მისი დანახვა.

ცხრილი 20.1

წვდომის მოდიფიკატორები	**დანიშნულება**
public	წვდომა შეუზღუდავია, ჩანს ყველგან.
protected	წვდომა შეზღუდულია შემცველი და წარმოებული კლასებით.
internal	წვდომა შეზღუდულია მოცემული ნაკრებით.
protected internal	წვდომა შეზღუდულია მოცემული ნაკრებით ან წარმოებული კლასებით.

private	წვდომა შეზღუდულია შემცველი ტიპით..

ამონანაბეჭდში 20.1 მოყვანილია პროგრამა ზემოთქმულის საილუსტრაციოდ.

ამონაბეჭდი 20.1 წვდომის მოდიფიკატორები: wvdomismodific.cs

```
using System;
 class klasi1
{   private int  zprivate=8;
   protected internal int wprotectedinternal=9;
        protected int xprotected=1;
        internal int yinternal=2;
        public void f()
System.Console.WriteLine("klasi1:this.zprivate={0},this.wprotectedinternal={1}",this.zprivat
e,this.wprotectedinternal);
System.Console.WriteLine("klasi1:this.xprotected={0},this.yinternal={1}",this.xprotected,this
.yinternal);
}
}
class klasi2: klasi1
{public void f1()
        {Console.WriteLine("klasi2:
klasi1:this.xprotected={0},this.yinternal={1},this.wprotectedinternal={2}",this.xprotected,this.
yinternal,this.wprotectedinternal);
klasi1 mC3 = new klasi1();
Console.WriteLine("klasi2:
klasi1:mC3.wprotectedinternal={0},mC3.yinternal={1}",mC3.wprotectedinternal,mC3.yinter
nal);
        }
}
class mTavari
{       public static void Main()
        {                klasi1 mC = new klasi1();
      mC.f();
                mC.wprotectedinternal=25;
                mC.yinternal = 15;
```

```
    Console.WriteLine("mTavari:winternal = {0},yinternal =
{1}",mC.wprotectedinternal,mC.yinternal);
    mC.f();
klasi2 mC2 = new klasi2();
Console.WriteLine("mTavari:wprotectedinternal = {0},yinternal =
{1}",mC2.wprotectedinternal,mC2.yinternal);
mC2.f();mC2.f1();
        }
}
```

გამოსავალი:

```
klasi1:this.zprivate=8,this.wprotectedinternal=9
klasi1:this.xprotected=1,this.yinternal=2
mTavari:winternal = 25,yinternal = 15
klasi1:this.zprivate=8,this.wprotectedinternal=25
klasi1:this.xprotected=1,this.yinternal=15
mTavari:wprotectedinternal = 9,yinternal = 2
klasi1:this.zprivate=8,this.wprotectedinternal=9
klasi1:this.xprotected=1,this.yinternal=2
klasi2: klasi1:this.xprotected=1,this.yinternal=2,this.wprotectedinternal=9
klasi2: klasi1:mC3.wprotectedinternal=9,mC3.yinternal=2
```

როგორც პროგრამიდან გამომდინარეობს, *zprivate* მხოლოდ *klasi1*-ში ჩანს. *xprotected klasi1* და *klasi2-ში ჩანს* და არ ჩანს *Main* მეთოდში. *winternal, wprotectedinternal, yinternal* ჩანან *Main* მეთოდში და არ გამოჩნდებიან კომპილაციის შემდეგ სხვის მიერ ამ კლასების გამოყენებისას.

ამონაბეჭდში 20.2 მოყვანილია *internal* მოდიფიკატორის საილუსტრაციო ტექსტი.

ამონაბეჭდი 20.2 შიდა მოხმარების მოდიფიკატორი. *internal.cs*

```
//Nakrebi1.cs:
// კომპილირდება ოფციით: /target:library
internal class Sabazoklasi
{   public static int IntM = 0;
}
//Nakrebi2.cs
// კომპილირდება ოფციით: /reference:Nakrebi1.dll
// მოსალოდნელი შეტყობინება: CS0122
class TestAccess
```

```
{   public static void Main()
    {
// შეცდომა, საბაზო კლასი არ ჩანს Nakrebi1-ის გარეთ:
 Sabazoklasi myBase = new Sabazoklasi();
    }}
```

ზედა მაგალითში ჯერ ხდება *Sabazoklasi* კლასის კომპილირება და ბიბლიოთეკაში ჩაწერა, ხოლო შემდგომ ბიბლიოთეკიდან მისი გამოყენების მცდელობა. მაგალითი განკუთვნილია *Net Framewok*-ში გასაშვებად.

კლასს გააჩნია მოდიფიკატორები *new,abstract,seald.* new მოდიფიკატორი გამოიყენება საბაზო კლასის ჩაწყობილი კლასის დასაფარად წარმოებულ კლასში. ეს მოდიფიკატორი ასევე გამოიყენება საბაზო კლასის მეთოდის დასაფარად. *abstrac* გამოიყენება აბსტრაქტული კლასის აღსაწერად. ამგვარი კლასის ეგზემპლარის შექმნა შეუძლებელია. აუცილებელია წარმოებულ კლასებში მოხდეს ყველა აბსტრაქტული მეთოდის რეალიზაცია. ამის შემდეგ შესაძლებელია წარმოებული კლასის ეგზემპლარის შექმნა. *seald* მოდიფიკატორი კრძალავს მოცემული კლასის წარმოებული კლასის შექმნას. ამონანაბეჭდში 20.3 ნაჩვენებია *new* მოდიფიკატორის გამოყენება ჩაწყობილი კლასის დასაფარად.

ამონანაბეჭდი 20.3.

```
using System;
public class Cemisabazo
{       public class Cemiklasi
   {public int x = 200;
                public int y;
        }
}
public class warmoebuli : Cemisabazo
{// ჩაწყობილი ტიპი ფარავს საბაზო ტიპის წევრებს:
new public class Cemiklasi
        { public int x = 100;
                public int y;
                public int z;
        }
        public static void Main()
        {// ობიექტის შექმნა დამფარავი კლასიდან :
                Cemiklasi S1  = new Cemiklasi();
        // ობიექტის შექმნა დაფარული კლასიდან:
                Cemisabazo.Cemiklasi S2 = new Cemisabazo.Cemiklasi();
```

```
                Console.WriteLine(S1.x);Console.WriteLine(S2.x);    }
}
```

ზედა დონის ტიპების მოდიფიკატორები შეიძლება იყოს მხოლოდ *internal, public.*ნაგულისხმევი მოდიფიკატორი არის *internal.* ჩაწყობილი ტიპებისთვის სამართლიანია

ცხრილი 20.2

ჩაწყობილი ტიპები	ნაგულისხმები მოდიფიკატორი	წევრის ნებადართული მოდიფიკატორი
enum	public	არ არსებობს
class	private	public protected internal private protected internal
interface	public	არ არსებობს
struct	private	public internal private

ლიტერატურა

1. http://www.microsoft.com/learning/syllabi/en-us/2609Afinal.mspx
2. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cscon/html/vcoriCStartPage.asp
3. http://zone.ni.com/devzone/conceptd.nsf/webmain/80AC25B00E13CEE386256D1A00648A54#1
4. http://www.softsteel.co.uk/tutorials/cSharp/contents.html
5. http://www.devarticles.com/c/a/C-Sharp/Visual-C-Sharp.NET-Part-1-Introduction-to-Programming-Languages/
6. http://www.e-trainonline.com/html/visual_c__net.html
7. http://knowledge.learnitonline.com/educate/onlinelearning/marketing/course/syllabus.jsp?productId=16760&productType=1