Andmebaasipõhiste veebirakenduste arendamine Microsoft Visual Studio ja SQL Server'i baasil

C#

Tallinn
2011

Microsoft .NET | ASP.NET
Microsoft Visual Studio 2010
Microsoft SQL Server 2008

Sisukord

C# **4**
Põhivõimalused **5**
Käivitamine 8
Suhtlus arvutiga 9
Arvutamine 10
Valikud 11
Kordused 13
Korrutustabel 15
Alamprogramm 16
Massiivid 17
Käsud mitmes failis 23
Tekst 24
Tekstifailid 26
Juhuarv 29
Omaloodud andmestruktuur 30
Edasijõudnute osa: Objektorienteeritud programmeerimine **34**
Tutvustus 34
Dokumenteerivad kommentaarid 40
Pärilus 42
Ülekate 45
Liidesed 46
Abstraktne klass 48
Meetodite asendus 50
Omadused 52
Indekseering 54
Struktuurne andmestik 56
Operaatorite üledefineerimine 64
Abivahendid **73**
Erindid 73
Enum 77
Andmekollektsioonid 79
Mallid 83
Atribuudid 86
Andmebaasiliides **91**
Ühenduse loomine, päring 91
Andmete lisamine 95
SQL-parameeter 95
Salvestatud protseduur 96
Funktsiooni delegaadid **99**
Funktsioonide komplekt 100
Sündmused 100
Ilmajaamad 101
Graafiline liides 104
Visual Studio C# Expressi install **106**

LINQ - .NET Language-Integrated Query 110

Põhikonstruktsioonide näited 112

Kokkuvõte.. 113

C#

Mõnigi võib ohata, et jälle üks uus programmeerimiskeel siia ilma välja mõeldud. Teine jälle rõõmustab, et midagi uut ja huvitavat sünnib. Kolmas aga hakkas äsja veebilahendusi kirjutama ja sai mõnegi ilusa näite lihtsasti kokku. Oma soovide arvutile selgemaks tegemise juures läheb varsti vaja teada, "mis karul kõhus on", et oleks võimalik täpsemalt öelda, mida ja kuidas masin tegema peaks. Loodetavasti on järgnevatel lehekülgedel kõigile siia sattunute jaoks midagi sobivat. Mis liialt lihtne ja igav tundub, sellest saab kiiresti üle lapata. Mis esimesel pilgul paistab arusaamatu, kuid siiski vajalik, seda tasub teist korda lugeda. Ning polegi loota, et kõik kohe lennult külge jääks!?

Selle jaoks on teksti sees koodinäited, mida saab kopeerida ja arvutis tööle panna. Ning mõningase muutmise ja katsetamise peale avastada, mis mille jaoks on ning kuidas seda oma kasuks rakendada saab. Töötav näide on üks hea kindel tugipunkt nagu üks suur puu lagendikul, kuhu oskab alati tagasi minna. Juhul, kui muutmistega on õnnestunud oma koodilõik nii sõlme keerata, et see sugugi enam töötada ei taha, saab alati võtta materjalist taas algse töötava näite ning sealt juurest katsetama hakata.

Kes pikemalt mitmesuguseid rakendusi kirjutab, avastab mõne aja pärast, et samas keeles kirjutatud programm võib vähemalt esmapilgul mõnevõrra erinev välja näha sõltuvalt sellest, kas programm käivitatakse veebist, tegutsetakse nuppudega ja tekstiväljadega aknas, väljundiks on mobiiltelefon või piirdub kogu tegevus tekstiaknaga. Esimesel korral võib tunduda, et oleks nagu täiesti eri keeltes ja eri moodi kirjutamine. Ühes kohas on alati koodi juures salapärane `button1_click`, teises `public static void Main` ning kolmandas veel midagi muud. Aga sellest ei tasu ennast väga häirida lasta. Ehkki .NETi ja C# juures on püütud eri kohtades käivituvate rakenduste loomist sarnasemaks muuta, tuleb siiski kirjutamisel arvestada käivitumiskoha võimalustega. Siin materjalis keskendume C# keele ülesehitusega seotud teemadele, mis on ühised kõigi käivitumiskohtade puhul. Ning kasutajaliidesena pruugime programmeerimisõpikute traditsioonilist lihtsat ning väheste (eksimis)võimalustega tekstiakent - nii jääb rohkem aega tähelepanu pühendada keele enese konstruktsioonidele, mida siis edaspidi julgesti veebirakenduste juures ja soovi korral mujalgi pruukida. Kes aga tahab omale koodi kirjutamiseks rohkem abivalmis ning lisavõimalustega keskkonda, sellele soovitame lugeda peatükki nimega Visual Studio C# Expressi install – samm sammult juhis vastava keskkonna paigaldamiseks ning esimese rakenduse käivitamiseks. Edasine kirjutamine sarnaselt konspektis olevale.

Põhivõimalused

Kui rakendus juba mingitki elumärki annab, on see tunduvalt rohkem, kui lihtsalt hulk koodi, mis peaks "midagi arukat" tegema. Tunne, et suutsin programmi ise, omade roosade kätega käima panna, on hea. Ja annab kindlustunde, et järgmisel korral saab asi ainult paremaks minna. Kui käima on lükatud, siis edasi võib mõtelda juba juurde panemise peale. Nii nagu talumees, kes omale krati oli ehitanud, sai hakata talle ülesandeid andma alles siis, kui kratt hinge sisse võttis. Muul juhul on tegemist palja põhuhunnikuga, millele teivas sisse löödud ja vanad kartulikorvid külge riputatud - olgu need nii suured ja vägevad tahes. Tööle hakkamiseks on hinge vaja. Aga hinge ei saa enne sisse puhuda, kui väikegi tervik olemas. Ning C# puhul näeb lühim tervikprogramm välja ligikaudu järgmine:

```
using System;
class Tervitus{
   public static void Main(string[] arg){
      Console.WriteLine("Tere");
   }
}
```

Esmapilgul võib tunduda, et suur hulk tundmatuid sõnu on ekraanile ritta seatud. Ning veel mõned sulud ka. Ning kui teada saada, et kõik see pikk jutt on vaid väikese programmi loomiseks, mis oma töö tulemusena ütleb "Tere", siis võib paista, et tegemist on ilmse raiskamisega. Eriti, kui mõni on varem tutvunud Basicu või Pythoniga, kus sama tulemuse saamiseks tuleb lihtsalt kirjutada

```
print "Tere"
```

Seetõttu veel 2000ndate aastate alguses tutvustati inimesi programmeerimisega mitmeski koolis just Basicu kaudu, sest algus on "nii lihtne". Ning kui programmid "väga suureks" - ehk siis tuhandete ridade pikkuseks - ei kasva, võib julgesti lihtsa algusega keele juurde jäädagi. Kõik vajalikud matemaatikaülesanded, kirjutamised ja arvutamised saavad tehtud.

Aga millegipärast on programmidel kombeks paisuda. Ning et suureks paisunud rakenduse seest sobiv toiming üles otsida, peab lisama vajalikule käsule üha uusi ja uusi kesti. Nii nagu üheainukese vajaliku sidekanali puhul võib korraldada nii, et telefonitoru tõstes ühendatakse rääkija kohe õigesse kohta. Kümnekonna puhul piisab telefoninumbritest, mis telefoni külge kleebitud või kiirklahvidele salvestatud. Kui aga raamatupidaja peab vajalikul hetkel kätte saama kõik temaga viimastel aastatel suhelnud kliendid, siis läheb juba tarvis korralikumat kataloogi.

Nõnda on C# ja ka mitme muu keele loojad otsustanud juba alguses programmikoodi osad korralikult süstematiseerida, et poleks vaja piltlikult öeldes pärast kataloogi tehes nuputada, millist värvi

paberilipikule kirjutatud Mati telefoninumber just selle õige Mati oma on. Mõnevõrra läheb tekst selle peale küll pikemaks, aga loodetavasti piisavalt vähe, et ikka julgete C# võlusid tundma õppida.

C# kood jaotatakse üksteise sees olla võivatesse plokkidesse. Iga ploki ümber on looksulud. Siinses näites on välimiseks plokiks klass nimega Tervitus ning tema sees alamprogramm nimega Main. Plokke võib vahel tunduvalt rohkem olla. Omaette terviklikud toimingud paigutatakse üldjuhul alamprogrammidesse. Nende sees võivad olla plokid tingimuste ja korduste tarbeks. Klassi moodustab üheskoos toimivate või sarnaste alamprogrammide komplekt (näiteks matemaatikafunktsioonid) - sellest pikemalt aga hiljem objektinduse juures. Suuremate rakenduste juures jagatakse klassid veel omakorda nimeruumidesse. Nii on lootust ka pikemate rakenduste puhul midagi hiljem koodist üles leida.

Mõned salapärased kohad on veel jäänud. Esimene rida

```
using System;
```

teatab, et nimeruumist System pärinevaid klasse saab kergesti kasutada - piisab vaid klassi nimetamisest. C# standardpaketis leidub tuhandeid kasutamisvalmis klasse. Lisaks veel loendamatu hulk lisapakette, mis muud programmeerijad on valmis kirjutanud. Kindlasti leidub nende hulgas korduvaid klassinimesid - sest mõttekaid ja kõigile arusaadavaid klassinimetusi lihtsalt ei saa olla nii palju. Kui aga samanimelised klassid on eri nimeruumides, siis nad üksteist üldjuhul ei sega. Nii võib igaüks oma nimeruumis nimetada klasse kuidas tahab. Ja using-lausega märgitakse, millist klasside komplekti parajasti kasutatakse. Edasi siis

```
class Tervitus{
```

mis praegusel juhul annab programmile ka nime ning

```
   public static void Main(string[] arg){
```

näitab alamprogrammi `Main`, kust käsurearakendus oma tööd alustab. Muudest sõnadest siin rea peal ei tasu end veel liialt häirida lasta. Kiirülevaatena: `public` näitab, et pole seatud käivitusõiguse tõkkeid, `static` - et alamprogramm Main on olemas ja käivitusvalmis kohe rakenduse töö alguses. `void` näitab, et ei raporteerita operatsioonisüsteemile programmi töö edukuse kohta. Ja ümarsulgudes oleva abil saab mõnes rakenduses kasutaja omi andmeid ette anda - siin seda võimalust aga ei kasutata.

Järgneb kasutajale nähtav toiming, ehk

```
      Console.WriteLine("Tere");
```

`Console` klass asub nimeruumis `System` ja on üleval märgitud using lause tõttu kasutatav. Klassi käsklus `WriteLine` lubab kirjutada konsoolile ehk tekstiekraanile. Praegu piirdutakse ühe väikese

teretusega. Jutumärgid on ümber selleks, et arvuti saaks aru, et tegemist on tekstiga - mitte näiteks käskluse või muutuja (märksõna) alla salvestatud andmetega.

```
    }
}
```

Kaks sulgu lõpus lõpetamas eespool avatud sulgusid. Iga sulg, mis programmikoodi sees avaneb, peab ka kusagil lõppema - muidu ei saa arvuti asjast aru, hing ei tule sisse ja programm ei hakka tööle. Tühikud ja reavahetused on üldjuhul vaid oma silmailu ja pildi korrastuse pärast. Kompilaatori jaoks võiks kõik teksti rahumeeli ühte ritta jutti kirjutada, enesele kasvaks aga selline programm varsti üle pea. Siin näites paistab, et alamprogramm `Main`' i sulg on sama kaugel taandes kui alamprogrammi alustav rida ise. Ning klassi sulg on sama kaugel kui klassi alustava rea sulg. Nõnda saab programmi pikemaks kasvamisel kergemini järge pidada, millises plokis või millistes plokkides vaadatav käsk asub.

Nõnda on esimene väike programm üle vaadatud ja loodetavasti tekib natuke tuttav tunne, kui vaadata järgnevaid ridu:

```
using System;
class Tervitus{
   public static void Main(string[] arg){
      Console.WriteLine("Tere");
   }
}
```

Käivitamine

Edasi tuleb tervitusrakendus käima saada. Töö tulemusel tekkinud pildi võib arenduskeskkonnas (näiteks Visual Studio 2010) ette saada kergesti - vajutad lihtsalt käivitusnuppu ja midagi ilmubki. Kuidas rakendus paigaldada, seda saab lugeda C# osa lõpust. Et aga "karu kõhust" tekiks parem ülevaade, püüame tervitaja käsurealt tööle saada. Sealt paistab paremini välja, millised etapid enne läbi käiakse, kui töötava programmini jõutakse. See on kasulik näiteks vigade otsimisel ja neist aru saamisel.

C# programmi kood salvestatakse laiendiga cs. Faili nime võib panna loodud programmi klassiga samasuguseks, kuigi otsest kohustust ei ole. Aga nõnda on kataloogist omi programme ehk kergem üles leida, kui klass ja fail on sama nimega.

Käsureale pääsemiseks on tarvilik see avada. Tavaliseks mooduseks on Start -> run ja sinna sisse käsklus cmd. Et sealtkaudu kompilaatorile ligi pääseda, on lisaks vaja panna otsinguteesse kompilaatori csc.exe kataloogi aadress, mis ühe konkreetse installatsiooni puhul on näiteks

```
c:\Windows\Microsoft.NET\Framework\v4.0.30319\.
```

Edasi tuleb minna sinna kausta, kus asub loodud koodifail.

```
cd C:\Users\windows 7\Documents
```

teeb praegu selle töö meie eest.

Faili loomiseks sobib kasvõi lihtsaim redaktor ehk Notepad käsuga

```
notepad algus.cs
```

Sisuks endiselt lihtne tervitus.

```
using System;
public class algus{
  public static void Main(String[] arg){
     Console.WriteLine("Tere");
  }
}
```

Siit saab juba kompilaatori välja kutsuda ning olemasolevat koodi kompileerida.

```
C:\Users\windows 7\Documents>c:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe
algus.cs
```

```
Microsoft (R) Visual C# 2010 Compiler version 4.0.30319.1
Copyright (C) Microsoft Corporation. All rights reserved.
```

Töö tehti ära, vigu sel korral ei leidunud.

Tulemusena tekkis kataloogi fail nimega algus.exe, mille võib rõõmsasti käivitada.

```
C:\Users\windows 7\Documents>algus.exe
Tere
```

Ülesandeid

- Muuda väljatrükitavat teksti
- Kirjuta ekraanile kaks rida (kaks järjestikust Console.WriteLine käsklust, kumbki omal real)
- Tekita programmi sisse väikeseid vigu, ja vaata, mis kompilaator selle peale teatab. Paranda tagasi ja veendu, et programm töötab jälle. Näiteks kaota ära n-täht sõnast Console, üks jutumärk "Tere" ümbert, üks sulg, semikoolon, muuda klassi nime. Ja igal korral jäta meelde või soovi korral suisa märgi üles, kas ja milline viga oli tegelikult ning millise veateate andis kompilaator. Sellised on kõige tüüpilisemad kompileerimisel tekkivad vead. Kord väikese programmi juures läbi katsetatuna on kergem sellistest veateadetest ka suures rakenduses aru saada.

Suhtlus arvutiga

Enamik programme vajavad andmeid kasutajalt - muul juhul ei teaks ju arvuti, mida meil vaja on. Kui just programmi ainsaks ülesandeks polegi kellaaja teatamine, sest sellisel juhul tõesti piisab vaid programmi enese käivitamisest.

Sisendite võimalused sõltuvad programmi kasutuskohast. Veebis näiteks saame pruukida veebilehel töötavaid graafikakomponente - nuppe, tekstivälju, rippmenüüsid jne. Windows Formide ehk arvutis iseseisvalt (mitte veebi kaudu) töötavate graafiliste rakenduste puhul on tavalised graafikakomponendid suhteliselt sarnased veebis nähtavatega. Vaid mõnevõrra vabamalt pääseb oma komponente juurde tegema ning mitmekülgsemaid võimalusi kasutama. Tekstiakna rakenduste juures piirdub suhtlus arvutiga loetava ja trükitava tekstiga. Lihtsaim dialoogi pidav programm näeb välja järgmine:

```
using System;
class Sisend{
   public static void Main(string[] arg){
      Console.WriteLine("Palun eesnimi:");
      string eesnimi=Console.ReadLine();
      Console.WriteLine("Tere, "+eesnimi);
   }
}
```

Ning töö paistab välja nii:

```
C:\Projects\oma\naited>Sisend
Palun eesnimi:
Juku
Tere, Juku
```

Esmane "Palun eesnimi" trükitakse välja sarnaselt nagu lihtsaimaski tervitavas programmis. Edasine `Console.ReadLine()` jääb kasutajalt sisestust ootama. Kõik, mis kasutaja kuni reavahetuseni kirjutab, püütakse kokku üheks tekstiks ning selle saab arvutisse meelde jätta. Märksõnaks ehk muutuja nimeks sai "eesnimi" ning andmetüübiks "string", mis inimkeeli tähendab teksti. Järgmisel real trükitakse tulemus välja. Nõnda sõltub programmi vastus küsimise peale sisestatavast nimest.

Arvutamine

Arvutamine teadupärast arvuti põhitöö - vähemalt arvutustehnika algaastatel. Et siin lahkesti kasutaja antud arve liita/lahutada saaks, tuleb kõigepealt hoolitseda, et need ka arvuti jaoks arvud ja mitte sümbolite jadad oleksid. Kõigepealt annab `ReadLine` kätte numbriliste sümbolitega teksti. Ning käsklus `int.Parse` muudab selle arvutuste jaoks kõlbulikuks. Tüüp `int` (sõnast integer) tähistab täisarvu. Kui on vaja komakohtadega ümber käia, siis sobib selleks tüüp `double`. Teise arvu puhul on andmete lugemine ning arvuks muundamine ühte käsklusesse kokku pandud. Nii võib ka.

Väljatrüki juures näete kolme looksulgudesse paigutatud arvu. Nõnda on võimalik andmeid trükkides algul määrata ära trükkimise kohad ning alles pärast loetellu kirjutada tegelikud väärtused. Juhul, kui väärtuste arvutamine on pikk (näiteks `arv1*arv2`), aitab see programmikoodi pilti selgemana hoida. Muul juhul tuleks hulk pluss- ja jutumärke väljatrüki juurde. Jutumärgid tekstide eristamiseks ning plussmärgid üksikute osade kokku liitmiseks. Samuti on sellisest asukohanumbritega paigutamisest kasu juhul, kui rakendust tõlgitakse. Keele lauseehituste tõttu võib sõnade järjestus lauses muutuda. Selliselt looksulgude vahel olevate arvudega mängides aga saab lihtsamalt tõlkida ilma, et peaks selleks programmikoodis märgatavaid muutusi tegema.

```
using System;
class Arvutus{
   public static void Main(string[] arg){
      Console.WriteLine("Esimene arv:");
      string tekst1=Console.ReadLine();
      int arv1=int.Parse(tekst1);
      Console.WriteLine("Teine arv:");
      int arv2=int.Parse(Console.ReadLine());
      Console.WriteLine("Arvude {0} ja {1} korrutis on {2}",
              arv1, arv2, arv1*arv2);
   }
}
C:\Projects\oma\naited>Arvutus
Esimene arv:
3
Teine arv:
5
Arvude 3 ja 5 korrutis on 15
```

Ülesandeid

- Küsi kahe inimese nimed ning teata, et täna on nad pinginaabrid
- Küsi ristkülikukujulise toa seinte pikkused ning arvuta põranda pindala
- Leia 30% hinnasoodustusega hinna põhjal alghind

Valikud

Ehk võimalus otsustamiseks, kui on vaja, et programm käituks kord üht-, kord teistmoodi. Allpoololev näide koos väljundiga võiks näidata, kuidas tingimuslause abil tehtud valik toimib.

```
using System;
public class Valik1{
   public static void Main(string[] arg){
      Console.WriteLine("Palun nimi:");
      string eesnimi=Console.ReadLine();
      if(eesnimi=="Mari"){
         Console.WriteLine("Tule homme minu juurde!");
      } else {
         Console.WriteLine("Mind pole homme kodus.");
      }
   }
}
```

Väljund:

```
D:\kodu\0606\opikc#>Valik1
Palun nimi:
Juku
Mind pole homme kodus.
```

Nagu näha - Jukut külla ei kutsutud. C# juures, nii nagu selle aluseks oleva C-keele puhul kasutatakse võrdlemise juures kahte võrdusmärki. Üks võrdusmärk on omistamine ehk kopeerimine. Arvude puhul saab kasutada ka võrdlusi < ja > ehk suurem kui ja väiksem kui. Näiteks

```
if(vanus>14){
   Console.WriteLine("Tuleb osta täispilet");
}
```

Samuti kehtivad võrdlused >= ja <= ehk suurem või võrdne ning väiksem või võrdne. Kui uurida, kas arv jääb soovitud vahemikku, tuleb järjest panna kaks võrdlust. Näiteks:

```
if(vanus>6 && vanus <=14){
   Console.WriteLine("Sinu jaoks on lapsepilet");
}
```

Kaks &-märki tähendab, et kogu tingimus on tõene ainult siis, kui mõlemad võrdlused on tõesed. Teistpidi saab ka.

```
if(vanus<7 || vanus>14){
   Console.WriteLine("Sulle lapsepilet ei sobi");
}
```

&&märke saab lugeda sõnaga "ja", || märke sõnaga "või". Ehk siis: kui vanus on alla seitsme või suurem neljateistkümnest, sel juhul lapsepilet ei sobi.

Kommentaarid

Vahel on kasulik enesele ja teistele selgituseks programmi sisse kommentaare lisada. Teksti sisse kirjutades pannakse kommentaari ette kaks kaldkriipsu. Selliselt kirjutatud teksti pole kompilaatori jaoks olemas. Enesel on aga vahel päris hea meenutada, mida mõne lausega kirja panna taheti.

```
      double summa=kogus*pirnihind; //korrutatakse ühe pirni hinnaga
```

Kommentaar võib olla ka üle mitme rea. Sellisel juhul pannakse kommentaari algusesse `/*` ning lõppu `*/` Siin on näiteks programmi väljund sarnaselt kaldkriips-tärni ning tärn-kaldkriipsu vahele paigutatud - et ka ilma käivitamata võib juba näha, mis konkreetsel juhul tehti.

Lisaks märkmete jätmisele on kommenteerimine ka heaks võimaluseks programmi testimisel ja vigade otsimisel. Kui kuidagi viga üles ei leia, siis saab algul kahtlase lause või lausete ploki vahel tervikuna välja kommenteerida. Seejärel veenduda, et programm ilma selle osata ilusti töötab. Ning edasi võib asuda juba kindlama tundega sellest väljakommenteeritud plokist viga otsima teades, et ta kindlasti seal peab olema. Omakorda saab lauseid ükshaaval võtta kommenteeritud osast välja ja lõpuks nõnda kõik ilusti tööle saada. Aga nüüd näide ise.

```
using System;
public class Valik1{
   public static void Main(string[] arg){
      double pirnihind=1.70;
      Console.WriteLine("Mitu pirni ostad?");
      double kogus=double.Parse(Console.ReadLine());
      double summa=kogus*pirnihind; //korrutatakse ühe pirni hinnaga
      Console.WriteLine("Kas kilekotti ka soovid? (jah/ei)");
      string vastus=Console.ReadLine();
      if(vastus=="jah"){
         summa=summa+0.70;
      }
      Console.WriteLine("Kogusumma: "+summa);
   }
}
/*
D:\kodu\0606\opikc#>Valik2
Mitu pirni ostad?
3
Kas kilekotti ka soovid? (jah/ei)
jah
```

```
Kogusumma: 5,8
*/
```

Kui soovida üksikutest osadest summat kokku arvutada nagu ülal näites, siis on küllalt tavaline, et iga küsimuse peal otsustatakse, kas ja mida selle juures teha. Siin nagu näha - juhul kui kilekott ostetakse lisaks, siis pannakse hinnale 70 senti juurde.

```
summa=summa+0.70
```

tähendab just seda.

Ülesandeid

* Küsi temperatuur ning teata, kas see on üle kaheksateistkümne kraadi (soovitav toasoojus talvel).

* Küsi inimese pikkus ning teata, kas ta on lühike, keskmine või pikk (piirid pane ise paika)

* Küsi inimeselt pikkus ja sugu ning teata, kas ta on lühike, keskmine või pikk (mitu tingimusplokki võib olla üksteise sees).

* Küsi inimeselt poes eraldi kas ta soovib osta piima, saia, leiba. Löö hinnad kokku ning teata, mis kõik ostetud kraam maksma läheb.

Kordused

Arvutist on enamjaolt kasu siis, kui ta meie eest mõned sellised tööd ära teeb, kus enesel tarvis ükshaaval ja üksluiselt sama toimetust ette võtta. Ühest hinnast paarikümne protsendi maha arvutamisega saab ka käsitsi mõne ajaga hakkama. Kui aga hindu on mitukümmend, siis on päris hea meel, kui arvuti selle töö meie eest ära teeb. Järgnevalt näide, kuidas arvuti vastu tulevale viiele matkajale tere ütleb. Täisarvuline muutuja nimega nr näitab, mitmenda matkaja juures parajasti ollakse. Käskluse while juurde kuuluvat plokki saab korrata. Plokk läbitakse üha uuesti juhul, kui ümarsulgudes olev tingimus on tõene. Et kordusi soovitud arv saaks, on juba programmeerija hoolitseda. Selleks on siin igal korral pärast tervitust käsklus `nr=nr+1`, ehk siis suurendatakse matkaja järjekorranumbrit. Kui see suurendamine kogemata tegemata jääks, siis jääkski arvuti igavesti esimest matkajat teretama (proovi järele). Nüüd aga töötav korduste näide:

```
using System;
public class Kordus1{
   public static void Main(string[] arg){
      int nr=1;
      while(nr<=5){
         Console.WriteLine("Tere, {0}. matkaja!", nr);
         nr=nr+1;
      }
   }
}
```

Ning tema väljund:

```
D:\kodu\0606\opikc#>Kordus1
Tere, 1. matkaja!
Tere, 2. matkaja!
Tere, 3. matkaja!
Tere, 4. matkaja!
Tere, 5. matkaja!
```

Heal lapsel mitu nime. Ehk ka korduste kirja panekuks on mitu moodust välja mõeldud. Algul näidatud while-tsükkel on kõige universaalsem, selle abil on võimalik iga liiki kordusi kokku panna. Sama tulemuse aga saab mõnevõrra lühemalt kirja panna `for`-i abil. Levinud kordusskeemi jaoks on välja mõeldud omaette käsklus, kus algul luuakse muutuja ja antakse talle algväärtus; seejärel kontrollitakse, kas järgnevat plokki on vaja täita; lõpuks võetakse ette toiming andmete ettevalmistamiseks uue ploki jaoks. `nr++` on sama, mis `nr=nr+1` - ehk siis suurendatakse muutuja väärtust ühe võrra. Näha programminäide:

```
using System;
public class Kordus2{
   public static void Main(string[] arg){
      for(int nr=1; nr<=5; nr++){
         Console.WriteLine("Tere, {0}. matkaja!", nr);
      }
   }
}
```

ning tema väljund:

```
D:\kodu\0606\opikc#>Kordus2
Tere, 1. matkaja!
Tere, 2. matkaja!
Tere, 3. matkaja!
Tere, 4. matkaja!
Tere, 5. matkaja!
```

Järelkontroll

Nii `for`-i kui `while` puhul kontrollitakse alati ploki algul, kas seda on vaja täita. Ning kui matkajate arv poleks mitte 5, vaid hoopis 0, siis sellisel juhul ei täideta plokki ainsatki korda, ehk kõik tered jäävad ütlemata. Mõnikord on aga teada, et plokk tuleb kindlasti läbida. Lihtsalt pole teada, kas sama teed tuleb ka teist või kolmandat korda käia. Tüüpiline näide selle juures on sisestuskontroll. Kui esimene kord toiming õnnestus, pole vaja kasutajalt andmeid uuesti pärida. Juhtus aga äpardus, siis tuleb senikaua jätkata, kuni kõik korras on. Siin küsitakse jälle tunninäitu. Sattus see arusaadavase vahemikku, minnakse rahus uue väärtusega edasi. Kui aga midagi ebaõnnestus, siis on arvuti väga järjekindel uuesti ja uuesti küsima lootuses, et kunagi ka midagi temale sobilikku jagatakse.

```
using System;
public class Kordus3{
```

```
    public static void Main(string[] arg){
       int tund;
       do{
          Console.WriteLine("Sisesta tund vahemikus 0-23");
          tund=int.Parse(Console.ReadLine());
       } while(tund<0 || tund>23);
       Console.WriteLine("Tubli, sisestasid {0}.", tund);
    }
}
/*
D:\kodu\0606\opikc#>Kordus3
Sisesta tund vahemikus 0-23
32
Sisesta tund vahemikus 0-23
11
Tubli, sisestasid 11.
*/
```

Ülesandeid

* Trüki arvude ruudud ühest kahekümneni

* Küsi kasutajalt viis arvu ning väljasta nende summa

* Ütle kasutajale "Osta elevant ära!". Senikaua korda küsimust, kuni kasutaja lõpuks ise kirjutab "elevant".

Korrutustabel

... ehk näide, kuidas eelnevalt vaadatud tarkused ühe programmi sisse kokku panna ning mis selle peale ka midagi tarvilikku teeb.

Algul on näha, kuidas otse programmi käivitamise juures ka mõned andmed sinna kätte anda. Et kui kirjutan

```
Korrutustabel 4 5
```

siis saadakse sellest aru, et soovin korrutustabelit nelja rea ja viie veeruga. Nende käsurea parameetrite püüdmiseks on alamprogramm `Main`-i ümarsulgudes koht `string[] argumendid`. Kõik käsureale kirjutatud sõnad (ka üksik number on arvuti jaoks sõna) pannakse sinna argumentide massiivi ehk jadasse, kust neid järjekorranumbri järgi kätte saab. Andmetüüp `string[]` tähendabki, et tegemist on stringide ehk sõnede ehk tekstide massiiviga. Kirjutades massiivi järgi `.Length`, saab teada, mitu elementi selles massiivis on - mis praegusel juhul on võrdne lisatud sõnade arvuga käsureal. Kõik sõnad saab ka ükshaaval järjekorranumbri järgi kätte. Arvestama peab ainult, et sõnu hakatakse lugema numbrist 0. Nii et kui eeldatakse, et tegemist on kahe parameetriga, siis nende kättesaamiseks peame ette andma numbrid null ja üks.

Nagu tingimusest on näha: juhul kui argumente pole täpselt kaks, siis kasutatakse vaikimisi ridade ja veergude arvu ning joonistatakse korrutustabel suurusega 10 korda 10.

Tabeli trükkimiseks on kaks `for`-tsüklit paigutatud üksteise sisse. Selles pole midagi imelikku - iga rea juures trükitakse kõik veerud esimesest kuni viimaseni. Ning selleks, et erinevate numbrite arvuga arvud meie tabelit sassi ei lööks, on väljatrüki juurde vorminguks kirjutatud `{0, 5}`. Ainsat `Console.Write` argumenti (järjekorranumbriga 0) trükitakse nõnda, et ta võtaks alati viis kohta.

```
using System;
class Korrutustabel{
   public static void Main(string[] argumendid){
      int ridadearv=10, veergudearv=10;
      if(argumendid.Length==2){
         ridadearv=int.Parse(argumendid[0]);
         veergudearv=int.Parse(argumendid[1]);
      }
      for(int rida=1; rida<=ridadearv; rida++){
         for(int veerg=1; veerg<=veergudearv; veerg++){
            Console.Write("{0, 5}", rida*veerg); //5 kohta
         }
         Console.WriteLine();
      }
   }
}
/*
C:\Projects\oma\naited>Korrutustabel
    1    2    3    4    5    6    7    8    9   10
    2    4    6    8   10   12   14   16   18   20
    3    6    9   12   15   18   21   24   27   30
    4    8   12   16   20   24   28   32   36   40
    5   10   15   20   25   30   35   40   45   50
    6   12   18   24   30   36   42   48   54   60
    7   14   21   28   35   42   49   56   63   70
    8   16   24   32   40   48   56   64   72   80
    9   18   27   36   45   54   63   72   81   90
   10   20   30   40   50   60   70   80   90  100
C:\Projects\oma\naited>Korrutustabel 4 5
    1    2    3    4    5
    2    4    6    8   10
    3    6    9   12   15
    4    8   12   16   20
*/
```

Alamprogramm

Nii nagu algul kirjas, nii ka siin tasub meeles pidada, et programmid armastavad paisuda ja paisuda. Seepärast tuleb leida mooduseid, kuidas üha suuremaks kasvavas koodihulgas orienteeruda. Alamprogramm on esmane ja hea vahend koodi sisse uppumise vältimiseks. Lisaks võimaldab ta terviklikke tegevusi eraldi ning mitu korda välja kutsuda. Samuti on ühe alamprogrammi tööd küllalt hea testida. Järgnevalt võimalikult lihtne näide, kuidas omaette tegevuse saab alamprogrammiks välja tuua. Siin on selliseks tegevuseks korrutamine. Luuakse käsklus nimega Korruta, talle antakse ette kaks täisarvu nimedega arv1 ja arv2 ning välja oodatakse sealt ka tulema täisarv.

```
using System;
class Alamprogramm{
   static int Korruta(int arv1, int arv2){
```

```
        return arv1*arv2;
    }
    public static void Main(string[] arg){
        int a=4;
        int b=6;
        Console.WriteLine("{0} korda {1} on {2}", a, b, Korruta(a, b));
        Console.WriteLine(Korruta(3, 5));
    }
}
/*
C:\Projects\oma\naited>Alamprogramm
4 korda 6 on 24
15
*/
```

Ülesandeid

* Koosta alamprogramm kahe arvu keskmise leidmiseks

* Koosta alamprogramm etteantud arvu tärnide väljatrükiks. Katseta.

* Küsi inimeselt kolm arvu. Iga arvu puhul joonista vastav kogus tärne ekraanile

Massiivid

Kuna arvuti on mõeldud suure hulga andmetega ümber käimiseks, siis on programmeerimiskeelte juurde mõeldud ka vahendid nende andmehulkadega toimetamiseks. Kõige lihtsam ja levinum neist on massiiv. Iga elemendi poole saab tema järjekorranumbri abil pöörduda. Algul tuleb määrata, millisest tüübist andmeid massiivi pannakse ning mitu kohta elementide jaoks massiivis on. Järgnevas näites tehakse massiiv kolme täisarvu hoidmiseks. Kusjuures nagu C-programmeerimiskeele sugulastele kombeks on, hakatakse elemente lugema nullist. Nii et kolme massiivielemendi puhul on nende järjekorranumbrid 0, 1 ja 2. Tahtes väärtusi sisse kirjutada või massiivist lugeda, tuleb selleks kirja panna massiivi nimi (praeguse juhul m) ning selle taha kandiliste sulgude sisse järjekorranumber, millise elemendiga suhelda tahetakse.

```
using System;
class Massiiv1{
    public static void Main(string[] arg){
        int[] m=new int[3];
        m[0]=40;
        m[1]=48;
        m[2]=33;
        Console.WriteLine(m[1]);
    }
}
/*
C:\Projects\oma\naited>Massiiv1
48
*/
```

Tsükkel andmete kasutamiseks

Massiivi kõikide elementidega kiiresti suhtlemisel aitab tsükkel. Siin näide, kuidas arvutatakse massiivi elementidest summa. Algul võetakse üks abimuutuja nulliks ning siis liidetakse kõikide massiivi elementide väärtused sellele muutujale juurde. Avaldis summa+=m[i] on pikalt lahti kirjutatuna summa=summa+m[i] ning tähendab just olemasolevale väärtusele otsa liitmist. for-tsükli juures kõigepealt võetakse loendur (sageli kasutatakse tähte i) algul nulliks, sest nullist hakatakse massiivi elemente lugema. Jätkamistingimuses kontrollitakse, et on veel läbi käimata elemente ehk loendur on väiksem kui massiivi elementide arv (massiivinimi.Length). Pärast iga sammu suurendatakse loendurit (i++). Nõnda ongi summa käes.

```
using System;
class Massiiv2{
   public static void Main(string[] arg){
      int[] m=new int[3];
      m[0]=40;
      m[1]=48;
      m[2]=33;
      int summa=0;
      for(int i=0; i<m.Length; i++){
         summa+=m[i];
      }
      Console.WriteLine(summa);
   }
}
/*
C:\Projects\oma\naited>Massiiv2
121
*/
```

Massiiv ja alamprogramm

Nagu ennist kirjutatud, saab eraldiseisva toimingu kergesti omaette alamprogrammi tuua. Siin on nõnda eraldatud summa leidmine. Massiive saab alamprogrammile samuti ette anda nagu tavalisi muutujaid. Lihtsalt andmetüübi taga on kirjas massiivi tunnusena kandilised sulud.

```
using System;
class Massiiv3{
   static int LeiaSumma(int[] mas){
      int summa=0;
      for(int i=0; i<mas.Length; i++){
         summa+=mas[i];
      }
      return summa;
   }
   public static void Main(string[] arg){
      int[] m=new int[3]{40, 48, 33};
      int vastus=LeiaSumma(m);
      Console.WriteLine(vastus);
   }
}


/*
C:\Projects\oma\naited>Massiiv3
```

```
121
*/
```

Algväärtustamine, järjestamine

Kui massiivi elementide väärtused on kohe massiivi loomise ajal teada, siis saab nad loogelistes sulgudes komadega eraldatult kohe sisse kirjutada. Nii saab andmed lihtsalt lühemalt kirja panna.

Kui elemendid on lihtsa võrdlemise teel järjestatavad nagu näiteks täisarvud, siis piisab nende rittaseadmiseks klassi Array ainsast käsust Sort.

```
using System;
class Massiiv4{
   public static void Main(string[] arg){
      int[] m=new int[3]{40, 48, 33};
      Array.Sort(m);
      for(int i=0; i<m.Length; i++){
         Console.WriteLine(m[i]);
      }
   }
}


/*
C:\Projects\oma\naited>Massiiv4
33
40
48
*/
```

Osutid ja koopiad

Kui ühe hariliku täisarvulise muutuja väärtus omistada teisele, siis mõlemas muutujas on koopia samast väärtusest ning toimingud ühe muutujaga teise väärtust ei mõjuta. Massiividega ning tulevikus ka muude objektidega tuleb tähelepanelikum olla. Kui üks massiiv omistada teisele, siis tegelikult kopeeritakse vaid massiivi osuti, mõlema muutuja kaudu pääsetakse ligi tegelikult samadele andmetele. Nagu järgnevas näites: massiivid `m2` ja `m` näitavad samadele andmetele. Kui ühe muutuja kaudu andmeid muuta, siis muutuvad ka teise muutuja kaudu nähtavad andmed nagu väljatrüki juures paistab. Algselt on massiivi `m` ja `m2` elemendid 40, 48, 33. Pärast massiivi m elemendi number 1 muutmist 32ks, on ka massiivi `m2` elemendid muutunud - väärtusteks 40, 32, 33. Nõnda on suurte andmemassiivide juures teise muutuja tegemine andmete juurde pääsemiseks arvuti jaoks kerge ülesanne. Samas aga peab vaatama, et vajalikke andmeid kogemata ettevaatamatult ei muudaks.

```
      int[] m=new int[3]{40, 48, 33};
      int[] m2=m; //Viide samale massiivile
      Tryki(m2);
      m[1]=32;
      Tryki(m2);
```

Kui soovida, et kaks algsetest andmetest pärit massiivi on üksteisest sõltumatud, siis tuleb teha algsest massiivist koopia (kloon).

```
int[] m3=(int[])m.Clone(); //Andmete koopia
m[1]=20;
Tryki(m3);
```

Pärast kloonimist muutused massiiviga m enam massiivi m3 väärtusi ei mõjuta.

Soovides massiivi tühjendada, aitab klassi Array käsklus Clear, mis täisarvude puhul kirjutab etteantud vahemikus (ehk praegusel juhul kogupikkuses, 0 ja Length-i vahel) täisarvude puhul väärtusteks nullid.

```
Array.Clear(m3, 0, m3.Length); //Tühjendus
```

Massiivist andmete otsimiseks sobib käsklus `IndexOf`. Soovitud elemendi leidumise korral väljastatakse selle elemendi järjekorranumber. Otsitava puudumisel aga -1.

```
      Console.WriteLine(Array.IndexOf(m,33));
      Console.WriteLine(Array.IndexOf(m,17)); //puuduv element
using System;
class Massiiv5{
   static void Tryki(int[] mas){
      for(int i=0; i<mas.Length; i++){
         Console.WriteLine(mas[i]);
      }
      Console.WriteLine();
   }
   public static void Main(string[] arg){
      int[] m=new int[3]{40, 48, 33};
      int[] m2=m; //Viide samale massiivile
      Tryki(m2);
      m[1]=32;
      Tryki(m2);
      int[] m3=(int[])m.Clone(); //Andmete koopia
      m[1]=20;
      Tryki(m3);
      Array.Clear(m3, 0, m3.Length); //Tühjendus
      Tryki(m3);
      Console.WriteLine(Array.IndexOf(m,33));
      Console.WriteLine(Array.IndexOf(m,17)); //puuduv element
   }
}

/*
C:\Projects\oma\naited>Massiiv5
40
48
33
40
32
33
40
```

```
32
33
0
0
0
2
-1
*/
```

Massiiv alamprogrammi parameetrina

Massiivimuutuja omistamisel tekib võimalus kahe muutuja kaudu samadele andmetele ligi pääseda. See võimaldab luua alamprogramme, mis massiivi elementidega midagi peale hakkavad. Eelnevalt vaadeldud käsklus `Sort` tõstab massiivis elemendid kasvavasse järjekorda. Siin on näha omatehtud alamprogramm `KorrutaKahega`, mis massiivi kõikide elementide väärtused kahekordseks suurendab.

```
using System;
class Massiiv6{
   static void KorrutaKahega(int[] mas){
      for(int i=0; i<mas.Length; i++){
         mas[i]=mas[i]*2;
      }
   }
   static void Tryki(int[] mas){
      for(int i=0; i<mas.Length; i++){
         Console.WriteLine(mas[i]);
      }
      Console.WriteLine();
   }
   public static void Main(string[] arg){
      int[] m=new int[3]{40, 48, 33};
      KorrutaKahega(m);
      Tryki(m);
   }
}

/*
C:\Projects\oma\naited>Massiiv6
80
96
66
*/
```

foreach - tsükkel

Kui on vaja kogumi kõik elemendid läbi käia, kuid samas pole tähtis läbikäigu järjekord, siis sellisel puhul aitab all näites paistev `foreach`-tsükkel. Selle abi saab näiteks summa arvutamise juures pruukida.

```
using System;
class Massiiv7{
   public static void Main(string[] arg){
      int[] m=new int[3]{40, 48, 33};
      foreach(int arv in m){
```

```
        Console.WriteLine(arv);
      }
   }
}

/*
C:\Projects\oma\naited>Massiiv7
40
48
33
*/
```

Mitmemõõtmeline massiiv

Massiivis võib mõõtmeid olla märgatavalt rohkem kui üks. Kahemõõtmelist massiivi saab ette kujutada tabelina, milles on read ja veerud. Kolmemõõtmelise massiivi elemendid oleksid nagu tükid kuubis, mille asukoha saab määrata pikkuse, laiuse ja kõrguse kaudu. Harva läheb vaja enam kui kolme mõõdet - siis on sageli juba otstarbekam ja arusaadavam oma andmestruktuur ehitada. Kasutamine toimub aga nii, nagu allpool näites näha. Massiivi elementidega saab ümber käia nagu tavaliste muutujatega. `foreach`-tsükliga saab soovi korral läbi käia ka mitmemõõtmelise massiivi kõik elemendid.

```
using System;
class Massiiv8{
   public static void Main(string[] arg){
      int[,] m=new int[2,3]{
         {40, 48, 33},
         {17, 23, 36}
       };
      Console.WriteLine(m[0, 1]); //48
      Console.WriteLine("M66dete arv: "+m.Rank);
      Console.WriteLine("Ridade arv: "+m.GetLength(0));
      Console.WriteLine("Veergude arv: "+m.GetLength(1));
                               //elemente mõõtmes nr. 1
      int summa=0;
      foreach(int arv in m){
        summa+=arv;
      }
      Console.WriteLine("Summa: "+summa);
   }
}
/*
C:\Projects\oma\naited>Massiiv8
48
M66dete arv: 2
Ridade arv: 2
Veergude arv: 3
Summa: 197
*/
```

Ülesandeid

* Küsi kasutaja käest viis arvu ning väljasta need tagurpidises järjekorras.

* Loo alamprogramm massiivi väärtuste aritmeetilise keskmise leidmiseks. Katseta.

* Loo alamprogramm, mis suurendab kõiki massiivi elemente ühe võrra. Katseta.

* Sorteeri massiiv ning väljasta selle keskmine element.

* Koosta kahemõõtmeline massiiv ning täida korrutustabeli väärtustega. Küsi massiivist kontrollimiseks väärtusi.

Käsud mitmes failis

Suuremate programmide puhul on täiesti loomulik, et kood jagatakse mitme faili vahel. Nii on hea jaotuse puhul kergem orienteeruda. Samuti on mugav terviklike tegevuste plokke muudesse programmidesse ümber tõsta, kui see peaks vajalikuks osutuma. Siin näites on kaks lihtsat arvutustehet omaette abivahendite klassis välja toodud.

```
class Abivahendid{
   public static int korruta(int a, int b){
      return a*b;
   }

   public static int liida(int a, int b){
      return a+b;
   }
}
```

Abivahendeid kasutav alamprogramm asub aga hoopis eraldi klassis ja failis.

```
using System;
class Abivahendiproov{
   public static void Main(string[] arg){
      Console.WriteLine(Abivahendid.korruta(3, 6));
   }
}
```

Kui tahta, et kompilaator vajalikud osad üles leiaks, tuleb kompileerimisel ette anda kõikide vajaminevate failide nimed.

```
C:\Projects\oma\naited>csc Abivahendid.cs Abivahendiproov.cs
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.
Lõpliku, exe-faili nimi määratakse vaikimisi faili järgi, kus oli
kirjas Main-meetod.
C:\Projects\oma\naited>Abivahendiproov
18
```

Ülesandeid

* Lisa käsklus täisarvude astendamiseks tsükli abil. Katseta

* Lisa kolmas fail paari tärnidest kujundeid joonistava funktsiooniga. Katseta peaprogrammis mõlema abifaili funktsioonide väljakutseid.

Tekst

Teksti koostamise, analüüsimise ja muutmisega puutuvad kokku enamik arvutiprogramme. Järgnevalt mõned näited, mille põhjal saab enamiku vajalikke tekstiga seotud toimetusi korda.

Teksti pikkuse saab kätte muutujast `Length`. Loetakse kokku kõik tekstis leiduvad sümbolid, kaasaarvatud tühikud.

Tekstist lõigu eraldamiseks sobib `Substring`. Esimese parameetrina olev arv näitab, mitmendast tähest hakatakse andmeid võtma, teine näitab, mitu tähte võetakse. `String` ehk sõne algab tähega number 0. Nii lugedes ongi sõna "tuli" algustäht järjekorranumbriga 5.

`IndexOf` võimaldab tekstis tähte või sõna leida. Leidmise korral väljastatakse leitud jupi algustähe järjekorranumber. Otsitava puudumise tulemusena väljastatakse -1.

```
using System;
class Tekst1{
  public static void Main(string[] arg){
    string s="Juku tuli kooli";
    Console.WriteLine("Pikkus: "+s.Length);
    Console.WriteLine(s.Substring(5, 4));
    Console.WriteLine("'tuli' kohal "+s.IndexOf("tuli"));
  }
}
/*
C:\Projects\oma\naited>Tekst1
Pikkus: 15
tuli
'tuli' kohal 5
*/
```

Muutmine

Teksti muutmiseks sobivad käsud `Insert` ja `Remove`. Esimene lisab soovitud kohale juurde etteantud teksti. Lausest "Juku tuli kooli" sai üheksandale kohale sõna " vara" lisamisel lause "Juku tuli vara kooli". `Remove` võimaldab sobivast kohast tähti välja võtta. Tehniliselt vaadates käsud `Insert` ja `Remove` ei muuda algses muutujas olevat teksti, vaid luuakse uus tekstiplokk mälus, mille poole on võimalik muutuja kaudu pöörduda. Muutujas `s` on endiselt "Juku tuli kooli". Vaid `s2` ja `s3` on uute väärtustega.

Tekstiga ümber käimiseks on ka mitukümmend muud käsklust, millega lähemat tutvust teha saab veebiaadressilt http://msdn2.microsoft.com/en-us/library/system.string_methods.aspx

Levinumatest on siin näha `StartsWith` alguse kontrolliks, `IndexOf` teksti otsinguks ja `Replace` asendamiseks. Aja jooksul läheb aga vaja tõenäoliselt meetodit `Compare` järjestuse kindlaks tegemisel, `EndsWith` lõpu kontrollimisel, `Trim` tühikute eemaldamisel, `ToUpper` ja `ToLower` suur-

ja väiketähtedeks muutmisel ning `ToCharArray` juhul, kui soovitakse tervikliku teksti asemel tähemassiivi kasutada (näiteks tähtede massiliseks ümbertõstmiseks).

```
using System;
class Tekst2{
  public static void Main(string[] arg){
    string s="Juku tuli kooli";
    Console.WriteLine("Pikkus: "+s.Length);
    Console.WriteLine(s.Substring(5, 4));
    Console.WriteLine("'tuli' kohal "+s.IndexOf("tuli"));
    string s2=s.Insert(9, " vara");
    Console.WriteLine(s2);
    string s3=s.Remove(5, 5); //Kuuendast alates viis tähte
    Console.WriteLine(s3);
    if(s.StartsWith("Juku")){
       Console.WriteLine("Algab Jukuga");
    }
    if(s.IndexOf("kala")==-1){
       Console.WriteLine("Siin ei ole kala");
    }
    Console.WriteLine(s.Replace("tuli", "jooksis"));
  }
}
/*
C:\Projects\oma\naited>Tekst2
Pikkus: 15
tuli
'tuli' kohal 5
Juku tuli vara kooli
Juku kooli
Algab Jukuga
Siin ei ole kala
Juku jooksis kooli
*/
```

Tükeldamine

Pika teksti osadeks jaotamiseks on mitmetes keeltes olemas vastavad käsud ja objektid. Nii ka siin. Käsuga `Split` võib olemasoleva teksti määratud sümbolite koha pealt juppideks lõigata. Kõikidest üksikutest tükkidest moodustatakse massiiv. Siin näiteks on eraldajaks koma, mis tähemassiivi üksiku elemendina ette antakse. Aga nagu aimata võib, saab massiivi lisada ka rohkem eraldajaid.

Kui on põhjust massiiv uuesti üheks pikaks tekstiks kokku panna, siis selle tarbeks leiab käskluse `Join`.

```
using System;
class Tekst3{
  public static void Main(string[] arg){
    string s="Tallinn,Tartu,Narva";
    string[] linnad=s.Split(new char[]{','});
    foreach(string linn in linnad){
       Console.WriteLine(linn);
    }
    Console.WriteLine(String.Join("; ", linnad));
  }
```

```
}
/*
D:\kodu\0606\opikc#>Tekst3
Tallinn
Tartu
Narva
Tallinn; Tartu; Narva
*/
```

Ülesandeid

* Trüki inimese nime eelviimane täht

* Teata, kas sisestatud nimi algab A-ga

* Trüki sisestatud nimi välja suurtähtedega

* Teata, kas lauses leidub sõna "ja"

* Asenda olemasolu korral "ja" sõnaga "ning" ja teata asendusest

* Trüki välja lause kõige pikem sõna

Tekstifailid

Kui samu lähteandmeid on vaja korduvalt kasutada, siis on neid igakordse sissetoksimise asemel mugavam sisse lugeda failist. Samuti on suuremate andmemahtude korral hea, kui sisendi ja väljundi andmed jäävad alles ka masina väljalülitamise järel. Keerukama struktuuriga andmete ning mitme üheaegse kasutaja korral (näiteks veebirakendustes) kasutatakse enamasti andmebaasi. Käsurea- või iseseisva graafilise rakenduse puhul on tekstifail aga lihtne ja mugav moodus andmete hoidmiseks. Samuti on ta tarvilik juhul, kui juba pruugitakse eelnevalt tekstifaili kujul olevaid andmeid.

Kirjutamine

Andmete faili saatmiseks tuleb kõigepealt moodustada voog etteantud nimega faili kirjutamiseks. Edasine trükk toimub juba sarnaselt ekraaniväljundiga `Write` ning `WriteLine` käskude abil. Lõppu tuleb kindlasti lisada käsklus `Close`, et teataks hoolitseda andmete füüsilise kettale jõudmise eest ning antakse fail vabalt kasutatavaks ka ülejäänud programmide jaoks. Siinse näite tulemusena tekib käivituskataloogi lihtne fail nimega "inimesed.txt", kuhu kirjutatakse kaks nime. Tekstiekraanil enesel pole käivitamise järel midagi näha - aga see veel ei tähenda, nagu programm midagi ei teeks. Lihtsalt tema töö tulemus jõuab loodavasse faili.

```
using System;
using System.IO;
class Failikirjutus{
  public static void Main(string[] arg){
            FileStream f = new FileStream("inimesed.txt",
                            FileMode.Create, FileAccess.Write);
            StreamWriter valja = new StreamWriter(f);
            valja.WriteLine("Juku");
            valja.WriteLine("Kati");
            valja.Close();
  }
```

```
}
```

Nagu aga Microsoftil on komme kasutajale kõik võimalikult mugavaks teha, siis on ka failide tervikuna kirjutamine ja lugemine ootamatult mugavaks tehtud – moodus, millest enamike teiste keelte puhul ei oska unistadagi. Nimeruumi System.IO klassis File on hulk käsklusi failidega toimetamiseks. Neist faili sisu kirjutamise jaoks sobib File.WriteAllText(failinimi, sisu); Nõnda lihtsalt jõuabki sisu faili.

```
using System;
using System.IO;
class Failikirjutus2{
 public static void Main(String[] arg){
    File.WriteAllText("tervitus.txt", "tere");
 }
}
```

Kui tegemist pikemate andmetega, siis ei tasu end sellest näilisest lihtsusest eksitada lasta. Siis on jupikaupa andmete lugemine, töötlemine ja kirjutamine ikka omal kohal. Kui aga on vaja mõni ettejäänud tekst lihtsalt kettale salvestada, siis piisab siinsest lihtsast käsust täiesti.

Lisamine

Kui FileMode.Create asendada seadega FileMode.Append, siis jäävad kirjutades vanad andmed alles. Uued read lihtsalt lisatakse olemasoleva teksti lõppu. Sellist lisamist läheb tarvis näiteks sündmuste logi kirjutamise juures.

```
using System;
using System.IO;
class Faililisamine{
  public static void Main(string[] arg){
            FileStream f = new FileStream("inimesed.txt",
                                 FileMode.Append, FileAccess.Write);
            StreamWriter valja = new StreamWriter(f);
            valja.WriteLine("Siim");
            valja.WriteLine("Sass");
            valja.Close();
  }
}
```

Ka lisamiseks on lühimoodus olemas: klassi File käsklus appendAllText

```
using System;
using System.IO;
class Faililisamine2{
 public static void Main(String[] arg){
    File.AppendAllText("tervitus.txt", "\nkuku");
 }
}
```

Lugemine

Faili lugemisel on vood teistpidi. `Create` ja `Write` asemel on `Open` ja `Read`. Ning `StreamWriter`i asemel `StreamReader`. Voost tuleva iga `ReadLine` tulemusena antakse üks tekstirida failist. Kui faili andmed lõppesid, saabub `ReadLine` käsu tulemusena tühiväärtus null. Selle järgi saab programmeerija otsustada, et fail sai läbi.

```
using System;
using System.IO;
class Faililugemine{
   public static void Main(string[] arg){
        FileStream f = new FileStream("inimesed.txt",
                          FileMode.Open, FileAccess.Read);
        StreamReader sisse=new StreamReader(f);
        string rida=sisse.ReadLine();
        while(rida!=null){
           Console.WriteLine(rida);
           rida=sisse.ReadLine();
        }
   }
}
/*
C:\Projects\oma\naited>Faililugemine.exe
Juku
Kati
*/
```

Lugemisekski on võimalik terve faili poole korraga pöörduda. Klassi File käsklus ReadAllText annab kogu sisu korraga ühte muutujasse tagasi. Edasi on juba programmeerija mõelda, mis nende andmetega peale hakata.

```
using System;
using System.IO;
class Faililugemine2{
 public static void Main(String[] arg){
    String sisu=File.ReadAllText("tervitus.txt");
    Console.WriteLine(sisu);
 }
}
```

Kui vaja faili ridadega ükshaaval midagi ette võtta, siis kannatab nad käsuga ReadAllLines lugeda ühte massiivi. Edasi on juba vabad käed edasi toimetamiseks.

```
using System;
using System.IO;
class Faililugemine3{
 public static void Main(String[] arg){
    String[] andmed=File.ReadAllLines("tervitus.txt");
    for(int i=0; i<andmed.Length; i++){
       Console.WriteLine(i+". "+andmed[i]);
    }
 }
}

/*
C:\Documents and Settings\alan>faililugemine3
0. tere
1. kuku
*/
```

Ülesandeid

* Tekita programmi abil fail, milles oleksid arvud ja nende ruudud ühest kahekümneni

* Tekstifaili igal real on müüdud kauba hind. Arvuta programmi abil nende hindade summa.

* Iga hinna kõrval on ka selle hinnaga müüdud kauba kogus. Korruta igal real hind kogusega ning liida lõpuks summad kokku.

* Võrreldes eelmise ülesandega kirjuta teise faili igale reale esimese faili vastaval real oleva hinna ja koguse korrutis.

Juhuarv

Kui soovida, et arvuti samade algandmete abil erinevalt käituks, tulevad appi juhuarvud. Nende abil saab kasutajale ette anda juhusliku tervituse, muuta soovi järgi pildi värvi, või näiteks kontrollida loodud funktsiooni toimimist mitmesuguste väärtuste juures. Kõigi nende erinevate väljundite aluseks on arvuti poolt loodud juhuarvud. Neid aitab saada nimeruumi `System` klassi `Random` eksemplar. Reaalarvu saamiseks on käsklus `NextDouble`. Kui soovida mõnda muud vahemikku kui nullist üheni, tuleb saadud arv lihtsalt soovitud suurusega läbi korrutada. Ühtlase jaotuse asemele normaal- või mõne muu jaotuse saamiseks tuleb mõnevõrra enam vaeva näha - juhul kui see peaks tarvilikuks osutuma.

Täisarv luuakse käsuga `Next`, andes ette ülempiiri, soovi korral ka alampiiri. Ning sobiva nime, anekdoodi või terviku saamiseks tuleb olemasolevad valitavad objektid paigutada massiivi, leida juhuarv massiivi elementide arvu piires ning võibki sobiva väärtuse välja võtta.

```
using System;
```

```
public class Juhuarv1{
   public static void Main(string[] arg){
      Random r=new Random();
      Console.WriteLine(r.NextDouble()); //Nullist üheni
      Console.WriteLine(r.Next(20)); //Täisarv alla 20
      Console.WriteLine(r.Next(50, 100)); //Viiekümnest sajani
      string[] nimed={"Juku", "Kati", "Mati"};
      Console.WriteLine(nimed[r.Next(nimed.Length)]); //Juhuslik nimi
   }
}
/*
D:\kodu\0606\opikc#>Juhuarv1
0,74339002358885
11
95
Kati
*/
```

Ülesandeid

* Trüki juhuslike teguritega korrutamisülesanne

* Kontrolli, kas kasutaja pakutud vastus oli õige

* Sõltuvalt vastuse õigsusest lase arvutil pakkuda olemasolevate hulgast valitud kiitev või julgustav kommentaar.

Omaloodud andmestruktuur

Standardandmetüüpe on .NET raamistikus kätte saada palju. Klasside arvu loetakse tuhandetes. Sellegipoolest juhtub oma rakenduste puhul olukordi, kus tuleb toimetada andmetega, mille hoidmiseks mugavat moodust pole olemas. Või siis on keegi kusagil selle küll loonud, aga lihtsalt ei leia üles. Harilike muutujate ja massiivide abil saab küll kõike arvutis ettekujutatavat hoida. Vahel aga on mugavam, kui pidevalt korduvate sarnaste andmete hoidmiseks luuakse eraldi andmetüüp. Siis on teada, et kokku kuuluvad andmed püsivad kindlalt ühes kohas koos ning pole nii suurt muret, et näiteks kahe firma andmed omavahel segamini võiksid minna.

Järgnevas näites kirjeldatakse selliseks omaette andmestruktuuriks punkt tasandil, kaks täisarvulist muutujat asukohti määramas.

```
struct Punkt{
    public int x;
    public int y;
}
```

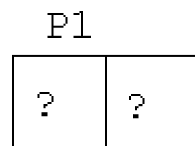
Kui edaspidi programmis kirjutatakse

```
Punkt p1
```

siis on teada, et `p1`-nimelisel eksemplaril on olemas `x` ja `y` väärtused ning neid on võimalik vaadata ja muuta. `struct`-iga kirjeldatud andmestruktuuri põhjal loodud muutuja omistamisel teisele sama tüüpi muutujale kopeeritakse väärtused ilusti ära. Nii nagu ühe täisarvulise muutuja väärtuse omistamisel teisele tekib sellest arvust koopia uude mälupiirkonda, nii ka struktuurina loodud `Punkt`i omistamisel teisele `Punkt`ile on mälus kaks eraldi ja sõltumatut piirkonda, mõlemal x-i ja y-i teise x-i ja y-iga samasugused. Õieti ei peakski sellist kopeerimist eraldi rõhutama - tundub ju nõnda omistamisel kopeerimine täiesti loomulik. Märkus on toodud lihtsalt tähelepanu äratamiseks. Kui tulevikus ei kasutata andmetüübi loomisel mitte sõna struct, vaid sõna class, siis käitutakse omistamisel mõnevõrra teisiti.

Nüüd aga programmi töö kohta seletus.

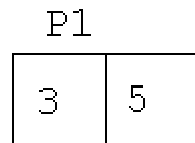
Punkt p1;



Luuakse muutuja nimega p1. Tal on mäluväljad nii x-i kui y-i jaoks.

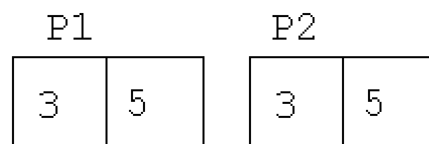
p1.x=3;

p1.y=5;



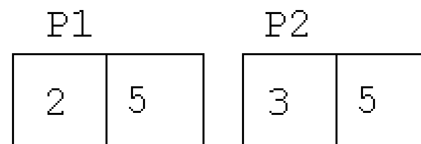
Väljadele antakse väärtused

Punkt p2=p1;



Luuakse muutuja nimega p2. Ka temal on mäluväljad nii x-i kui y-i jaoks. Muutuja p1 x-i väärtus kopeeritakse muutuja p2 x-i väärtuseks ning samuti ka y-iga. Praeguseks on siis mälus kahte kohta kirjutatud kolm ning kahte kohta viis. Edasi muudetakse esimese punkti x-koordinaat kaheks. Teise oma aga jääb endiselt kolmeks.

p1.x=2;



Et kui nüüd teise punkti koordinaadid välja trükkida, siis tulevad sealt rõõmsasti 3 ja 5.

```
Console.WriteLine(p2.x+" "+p2.y);
```

Võrreldes varasemate näidetega on koodi juurde tekkinud sõna `namespace`. Varasemate väiksemate programmide puhul mõtles kompilaator neile juurde nimetu ja nähtamatu vaikimisi nimeruumi. Kui aga klasse või struktuure on rohkem, on viisakas koos kasutatavad klassid teise nimeruumi eraldada. Sel juhul pole nii palju karta, et keegi teine oma klassid meie omadega sarnaselt nimetaks ning nad meie omadega tülli läheksid. Kui ka klassi nimi juhtub sama olema, aga nimeruum on erinev, siis saab kompilaator aru, et need on erinevad ning ei sega üksteist.

Töötav kood tervikuna.

```
using System;
namespace Punktid{
   struct Punkt{
       public int x;
       public int y;
   }
   class Punktiproov{
     public static void Main(string[] arg){
         Punkt p1;
         p1.x=3;
         p1.y=5;
         Punkt p2=p1;  //Väärtused kopeeritakse
         p1.x=2;
         Console.WriteLine(p2.x+" "+p2.y);
     }
   }
}
/*
C:\Projects\oma\naited>Punktid
3 5
*/
```

Punktimassiiv

Oma andmetüüpi on enamasti põhjust luua juhul, kui seda tüüpi andmeid on rohkem kui paar-kolm eksemplari. Suurema hulga samatüübiliste andmete jaoks kasutatakse sageli massiivi. Nii ka omaloodud andmetüübi puhul.

```
Punkt[] pd=new Punkt[10];
```

teeb punktidest kümme eksemplari järjenumbritega nullist üheksani. Ning ilusti järjenumbri järgi elemendi poole pöördudes saab sinna nii väärtusi panna kui küsida. Tsükli abil pannakse iga elemendi y väärtuseks tema järjekorranumbri ruut. Ning väljatrükil

```
Console.WriteLine(pd[4].y);
```

trükitakse rõõmsasti ekraanile 16.

```
using System;
namespace Punktid2{
   struct Punkt{
       public int x;
       public int y;
   }
   class Punktimassiiv{
     public static void Main(string[] arg){
         Punkt[] pd=new Punkt[10]; //mälu kohe olemas
         for(int i=0; i<10; i++){
           pd[i].x=i;
           pd[i].y=i*i;
         }
         Console.WriteLine(pd[4].y);
     }
   }
}
/*
C:\Projects\oma\naited>Punktid2
16
*/
```

Ülesandeid

* Koosta struktuur riidelappide andmete hoidmiseks: pikkus, laius, toon

* Katseta loodud andmetüüpi paari eksemplariga.

* Loo lappidest väike massiiv, algväärtusta juhuarvude abil.

* Trüki välja lappide andmed, mille mõlemad küljepikkused on vähemalt 10 cm.

Edasijõudnute osa: Objektorienteeritud programmeerimine

Tutvustus

struct-lausega loodud kirjed on mõeldud põhiliselt andmete hoidmiseks ning vajadusel üksikute andmete (nt. sünniaasta abil vanuse) välja arvutamiseks. Toimingud andmetega jäävad enamjaolt kirjest väljapool paikneva programmi ülesandeks. Objektide puhul aga püütakse enamik konkreetse objektitüübiga seotud toiminguid ühise kesta sisse koondada. Piir kirjete ja objektide vahel on mõnevõrra hägune ning mõnes keeles (nt. Java) polegi kirjete jaoks eraldi keelekäsklust olemas. Samas on siiski hea eri lähenemised lahus hoida.

Traditsioonilise objektorienteeritud programmeerimise juures pole eksemplari muutujatele sugugi võimalik otse väljastpoolt ligi saada. Samuti on vaid mõned alamprogrammid teistele objektidele vabalt kasutavad, küllalt palju vahendeid võib olla loodud vaid objekti enese toimimise tarbeks. Selline kapseldamine aitab suuremate programmide puhul järge pidada, et vaid ühe teemaga seotud ning teemadevahelised lõigud üksteist segama ei hakkaks. Lisaks väliste käskude vähemast arvust tulevale lihtsusele lubab muutujate ja alamprogrammide varjamine teiste objektide eest hiljem oma objekti sisemist ülesehitust muuta ilma, et muu programmi osa sellest häiritud saaks. Selline muutmisvõimalus on aga hästi tänuväärne olukorras, kus tegijaid on palju, ja samu objekte kasutatakse tulevikus teisteski programmides.

Järgnevas näites on punkt loodud klassina ehk objektitüübina. Koordinaadid `x` ja `y` on privaatse ligipääsuga, neid saab kasutada vaid sama klassi meetodites. Väärtuste küsimiseks on eraldi käsklused `GetX` ja `GetY`. Esimese hooga võib tunduda selline väärtuse küsimise peitmine imelik. Aga kui tulevikus näiteks soovitakse teha statistikat, et mitu korda on küsitud `x`-i ja mitu korda `y`-i väärtust, siis alamprogrammiga andmete küsimise puhul saab selle loenduse kergesti paika panna. Muutuja puhul aga ei ole nõnda tavaprogrammeerimise vahenditega võimalik. Kuna siin on lubatud punkti asukoha määramine vaid eksemplari loomisel, siis kord antud väärtusi enam muuta ei saa, sest `Punkt`i juures pole lihtsalt vastavat käsklust. Klassiga samanimelist väljastustüübita käsklust nimetatakse konstruktoriks. See käivitatakse vaid üks kord - eksemplari loomisel - ning sinna sisse pannakse tavaliselt algväärtustamisega seotud toimingud. Kui näiteks algväärtustamisel seada sisse piirang, et koordinaatide väärtused peavad jääma nulli ja saja vahele, siis pole võimalik muus piirkonnas punkti luua. Sedasi õnnestub objektina luua küllalt keerukaid süsteeme, mis oskavad "iseenese eest hoolitseda" ning millel saab lihtsalt käsklusi ehk meetodeid välja kutsuda.

```
using System;
namespace Punktid3{
   class Punkt{
        private int x;
        private int y;
        public Punkt(int ux, int uy){
          x=ux; y=uy;
```

```
        }
        public int GetX(){
          return x;
        }
        public int GetY(){
          return y;
        }
        public double KaugusNullist(){
          return Math.Sqrt(x*x+y*y);
        }
    }
    class Punktiproov{
      public static void Main(string[] arg){
          Punkt p1=new Punkt(3, 4);
          Console.WriteLine(p1.GetX());
          Console.WriteLine(p1.KaugusNullist());
      }
    }
}
/*
D:\kodu\0606\opikc#>Punktid3
3
5
*/
```

Ülesandeid

* Koosta klass riidelapi andmete hoidmiseks: pikkus, laius, toon

* Lisa käsklus lapi andmete väljatrükiks

* Lisa käsklus lapi pindala arvutamiseks

* Lisa meetod (alamprogramm) lapi poolitamiseks: pikem külg tehakse poole lühemaks.

* Poolitamise meetod lisaks algse lapi poolitamisele väljastab ka uue samasuguse algsest poole väiksema eksemplari.

* Lisa teine poolitusmeetod, kus saab määrata, mitme protsendi peale lõigatakse pikem külg

Klassimuutuja

Klassi juurde korjatakse võimalusel kokku kõik vastavat tüüpi objektidega tehtavad toimingud ja andmed. Enamasti kuuluvad andmed isendite ehk eksemplaride juurde, kuid mitte alati. Näiteks tüüpiliselt - loodud punktide arvu loendur on küll punktidega sisuliselt seotud, kuid pole sugugi ühe konkreetse punkti omadus. Punktide arv on olemas ka juhul, kui ühtegi reaalset punkti eksemplari pole veel loodud. Sellisel juhul on punktide arv lihtsalt null. Samuti on punktide arv olemas juhul, kui punkte on loodud palju. Lihtsalt sel juhul on loenduri väärtus suurem. Sellise klassi ja mitte isendiga seotud muutuja ette pannakse sõna `static`.

```
        static int loendur=0;
```

Teine asi on punkti järjekorranumber - see on iga punkti juures kindel arv, mis antakse punktile tema loomise hetkel ning mida siinse programmi puhul enam hiljem muuta ei saa.

```
private int nr;
nr=++loendur;
```

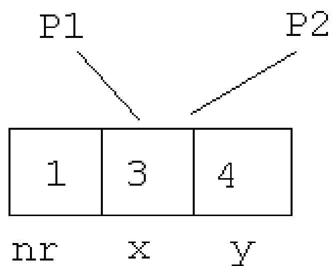
tähendab, et kõigepealt suurendatakse loendurit (olemasolevate punktide arvu) ühe võrra ning saadud arv pannakse uue loodud punkti järjekorranumbriks.

Osuti, omistamine

Klassi eksemplaride puhul toimub omistamine teisiti kui struktuuri eksemplaride puhul. Struktuuri juures tehti lihtsalt väljade väärtustest koopia ja edasi elas kumbki eksemplar oma elu teisest sõltumata. Kui nüüd aga teha omistamised

```
Punkt p1=new Punkt(3, 4);
Punkt p2=p1; //Kasutab sama mälupiirkonda
```

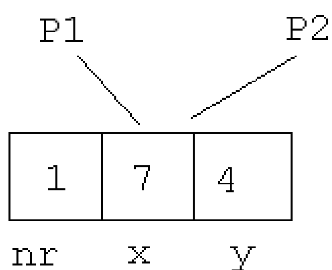
siis on mälus tegelikult koht vaid ühe punkti andmete jaoks, muutujate p1 ning p2 kaudu pääseb lihtsalt ligi samadele andmetele.



Nii et kui ühe muutuja kaudu andmeid muuta:

```
p1.SetX(7);
```

siis muutuvad andmed ka teise märksõna alt paistvas vaates.



Ehk siis ükskõik, kas andmete poole pöörduda `p1` või `p2` kaudu – ikka saan tegelikult samad väärtused.

```
Console.WriteLine(p2.GetNr()+" "+p2.GetX()+" "+p2.GetY());
```

trükib välja

```
1 7 4
```

ehkki esialgu olid koordinaatideks 3 ja 4 ning `p2` kohe deklareerimisel polnud sugugi algne koht esimese punkti andmetele ligi pääsemiseks.

Punkt järjekorranumbriga 2 on alles `p3`, sest tema loomisel käivitati uuesti Punkti konstruktor – loodi uus eksemplar, mille käigus suurendati loendurit ning anti uued kohad andmete mälus hoidmiseks.

Nüüd siis eelnevate seletuste kood tervikuna.

```
using System;
namespace Punktid4{
   class Punkt{
       static int loendur=0;
       private int nr;
       private int x;
       private int y;
       public Punkt(int ux, int uy){
         x=ux; y=uy;
         nr=++loendur;
       }
       public int GetX(){
         return x;
       }
       public int GetY(){
         return y;
       }
       public int GetNr(){
         return nr;
       }
       public void SetX(int ux){
         x=ux;
       }
       public void SetY(int uy){
         y=uy;
       }
       public double KaugusNullist(){
         return Math.Sqrt(x*x+y*y);
       }
   }
   class Punktiproov{
     public static void Main(string[] arg){
         Punkt p1=new Punkt(3, 4);
         Punkt p2=p1; //Kasutab sama mälupiirkonda
         p1.SetX(7);
         Console.WriteLine(p2.GetNr()+" "+p2.GetX()+" "+p2.GetY());
         Punkt p3=new Punkt(77, 32); //Punkt järgmise järjekorranumbriga
         Console.WriteLine(p3.GetNr());
     }
   }
}
```

```
/*
C:\Projects\oma\naited>Punktid4
1 7 4
2
*/
```

Punktimassiiv

Nii nagu üksiku muutuja juures, nii ka massiivi puhul tuleb (erinevalt structist) igale uuele klassi eksemplarile new-käsuga mälu anda.

```
         Punkt[] pd=new Punkt[10];
```

loob vaid kümme mälupesa, mis on võimelised näitama Punkt-tüüpi objektile. Samas punktiobjekte pole selle käsu peale veel ühtegi. Ehk kui keegi sooviks näiteks pd[3] olematu eksemplariga midagi teha, siis antaks veateade.

Alles siis, kui tsüklis luuakse iga ringi juures uus Punkti eksemplar ning pannakse massiivi element sellele näitama, on võimalik iga massiivi elemendiga kui Punkti eksemplariga ringi käia.

```
         for(int i=0; i<pd.Length; i++){
            pd[i]=new Punkt(i, i*i);
         }
```

Näiteks küsida konkreetse elemendi väärtust.

```
         Console.WriteLine(pd[4].GetY());
using System;
namespace Punktid5{
   class Punkt{
       static int loendur=0;
       private int nr;
       private int x;
       private int y;
       public Punkt(int ux, int uy){
         x=ux; y=uy;
         nr=++loendur;
       }
       public int GetX(){
         return x;
       }
       public int GetY(){
         return y;
       }
       public int GetNr(){
         return nr;
       }
   }
   class Punktiproov{
     public static void Main(string[] arg){
         Punkt[] pd=new Punkt[10];
```

```
         for(int i=0; i<pd.Length; i++){
            pd[i]=new Punkt(i, i*i);
         }
         Console.WriteLine(pd[4].GetY());
      }
   }
}
/*
C:\Projects\oma\naited>Punktid5
16
*/
```

Ülesandeid

* Koosta klass riidelapi andmete hoidmiseks: pikkus, laius, toon

* Koosta riidelappidest massiiv.

* Koosta uus lapimassiiv, kuhu pannakse osa eelmisest massiivist pärit lappe ja osa muid lappe (kaltsukott).

* Arvuta mõlema massiivi puhul välja seal leiduvate lappide pindalade summa.

* Koosta riidelapi klassile staatiline käsklus näitamaks loodud riidelappide keskmist pindala. Lappide puudumisel väärtuseks -1. Lisa vajalikud staatilised muutujad (lappide arv, kogupindala)

* Koosta klass isikukoodi andmete hoidmiseks

* Lisa käsklus sünnikuupäeva küsimiseks

* Lisa käsklus sünnikuu küsimiseks sõnana

* Lisa käsklus sünniaasta väljastamiseks ka sajandit arvestades

* Kontrolli isikukoodi objekti tegemisel koodi pikkust

* Kontrolli isikukoodi kontrollisumma õigsust vastavalt järgnevale algoritmile:

Isikukoodi kontrolli algoritm

Isikukoodis peavad kõik sugu ning kuupäeva tähistavad väärtused olema

võimalikud ning viimane kontrollnumber arvutatakse järgneva algoritmi järgi:

liidetakse kokku esimese üheksa numbri korrutised igale arvule vastava

järjekorranumbriga, kümnenda numbri puhul ühega ning leitakse saadud summa

jääk jagamisel 11-ga. Kui jääk on võrdne kümnega, siis tehakse arvutus

uuesti ning võetakse teguriteks vastavalt 3, 4, 5, 6, 7, 8, 9, 1, 2, 3.

Näide: isikukoodi 37605030299 kontroll. Summa =

1*3 + 2*7 + 3*6 + 4*0 + 5*5 + 6*0 + 7*3 + 8*0 + 9*2 + 1*9 = 108

108 jääk jagamisel 11-ga on 9 => isikukoodi viimane number peab olema

üheksa.

Dokumenteerivad kommentaarid

Dokumentatsiooni loomiseks on üks võimalus panna kolme kaldkriipsuga märgistatud kommentaarid koodi sisse, kommenteeritava ploki või muutuja ette. Sellised kommentaarid oskab kompilaator sobiva võtme järgi välja, eraldi kommentaaride faili korjata. Kommentaariks sobib xml-märgenditega piiratud tekst. Tavalisimaks elemendiks on <summary>, kuid eraldi on olemas ka kokkuleppelised käsklused parameetrite, versioonide jm. jaoks.

Readonly

Võtmesõna `readonly` muutuja ees tähendab, et sinna võib väärtuse omistada vaid ühe korra. Olgu siis muutuja kirjeldamisel või konstruktoris. Selliste muutujate puhul hoolitseb juba kompilaator, et poleks võimalik kirjutada käsklusi, mis readonlyga kaitstud mäluväljade väärtusi muudavad.

Näide

```
using System;
namespace Punktid6{
   /// <summary>
   ///   Tasandi punkti andmete hoidmine
   /// </summary>
   class Punkt{
       /// <summary>
       ///    Muutuja ainult lugemiseks.
       ///    Andmed sisestatavad vaid konstruktoris.
       /// </summary>
       private readonly int x;
       private readonly int y;
       /// <summary>
       ///    Algandmed punkti loomisel kindlasti vajalikud
       /// </summary>
       public Punkt(int ux, int uy){
         x=ux; y=uy;
       }
       public int GetX(){
         return x;
       }
       public int GetY(){
         return y;
       }
       /// <summary>
       ///   Kaugus arvutatakse Pythagorase teoreemi järgi.
       /// </summary>
       public double KaugusNullist(){
         return Math.Sqrt(x*x+y*y);
       }
   }
   /// <summary>
   ///    Eraldi klass punkti katsetamiseks.
   /// </summary>
   class Punktiproov{
```

```
        public static void Main(string[] arg){
            Punkt p1=new Punkt(3, 4);
            Console.WriteLine(p1.KaugusNullist());
        }
    }
}
```

Kompileerimine

Kompileerides saab siis /doc võtmega anda ette failinime, kuhu kirjutatakse kommentaarid.

```
/*
D:\kodu\0606\opikc#>csc Punktid6.cs /doc:Punktid6.xml
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.
D:\kodu\0606\opikc#>Punktid6
5
*/
```

Kommentaarifail

Edasi on juba vastavalt kasutaja soovile - kas ta loeb tekkinud XML-faili lihtsa tekstiredaktoriga, mõne eraldi XML-lugejaga (nt. vastavate võimetega veebibrauser) või siis kirjutab juurde XSL-lehe andmete omale sobivale kujule muundamiseks.

Väike ülevaade tekkinud XML-failist. Esimene rida on ühine kõigile XML-vormingus dokumentidele.

```
<?xml version="1.0"?>
```

Järgnevaks juurelemendiks on doc, mille sisse kõik ülejäänud andmed mahuvad.

```
<doc>
```

Assembly tähendab kokkukompileeritud üksust. Siin on ta laiendiks .exe, mõnel juhul võib olla aga ka .dll

```
    <assembly>
        <name>Punktid6</name>
    </assembly>
```

Ning edasi juba tulevadki klassi, muutuja, konstruktori ja meetodi töö kirjeldused.

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>Punktid6</name>
    </assembly>
```

```
    <members>
        <member name="T:Punktid6.Punkt">
            <summary>
              Tasandi punkti andmete hoidmine
            </summary>
        </member>
        <member name="F:Punktid6.Punkt.x">
            <summary>
               Muutuja ainult lugemiseks.
               Andmed sisestatavad vaid konstruktoris.
            </summary>
        </member>
        <member name="M:Punktid6.Punkt.#ctor(System.Int32,System.Int32)">
            <summary>
               Algandmed punkti loomisel kindlasti vajalikud
            </summary>
        </member>
        <member name="M:Punktid6.Punkt.KaugusNullist">
            <summary>
              Kaugus arvutatakse Pythagorase teoreemi jC¤rgi.
            </summary>
        </member>
        <member name="T:Punktid6.Punktiproov">
            <summary>
               Eraldi klass punkti katsetamiseks.
            </summary>
        </member>
    </members>
</doc>
```

Ülesandeid

* Koosta klass riidelapi andmete hoidmiseks, lisa kommentaarid.

* Tutvu kompileerimisel loodud kommentaaride failiga.

Pärilus

Ajapikku on objektorienteeritud programmeerimiskeelte juurde lisatud mitmesuguseid täiustusi ja võimalusi. Pärilus (inheritance) on mingil moel enamike objektorienteeritud keelte küljes olemas. Nii ka end suhteliselt arenenuks keeleks pidava C# juures.

Objektidega toimetamisel ning pärilusel sealhulgas on vähemalt kaks eesmärki. Püütakse lihtsustada koodi kohandamist. Samuti võimaldab pärilus vältida sama tööd tegeva koodi kopeerimist, lubades samu käsklusi kasutada mitmel pool.

Päriluse abil püütakse programmeerimiskeele objektitüüpide omavahelisi suhteid lähendada inimeste igapäevaselt tajutavale. Üheksakümnendate aastate algul ilmunud objektorienteeritud programmeerimist tutvustavas õpikus oli ilus näide: "sebra on nagu hobune, kellel on triibud". Sarnane tuttava kaudu uue tüübi kokkupanek ongi päriluse põhisisu. Midagi lisatakse, midagi muudetakse, vahel harva ka kaotatakse. Ning märkamatult muutubki hobune sebraks või tühi paneel tekstiväljaks.

Nõnda õnnestub olemasolevate (pool)valmis tükkide abil omale sobiv komplekt kokku panna. Teiselt poolt võidakse objektide päriluspuu ehitada ka oludes, kus kõik programmi osad on enese määrata. Sellisel juhul pole küll eesmärk omale üks ja parim tüüp kokku panna, vaid otsitakse ühisosi, mille puhul on võimalik valminud tüübid gruppidesse jaotada ning nende gruppidega vajadusel koos midagi ette võtta. Tüüpilise näitena pole teatripiletit müües vaja teada inimese sugu ja ametit. Küll aga on balletirolli juures mõlemad andmed tähtsad. Tavaelus tundub loomulikuna, et eriomadusi arvestatakse vaid kohtades, kus need on vajalikud ning muul juhul aetakse vaid üldiste tunnustega läbi. Haruharva tuleb eraldi rõhutada, et "meie teatrisse tohivad vaatajaks tulla inimesed sõltumata usutunnistusest ja nahavärvist". Arvuti juures tuleb aga tasemed selgelt välja tuua. Igal keerukuse astmel tuleb määrata, millised oskused ja omadused sinna juurde kuuluvad ning millise rolli jaoks millisel tasemel oskuste komplekti vaja on. Keerukaks kiskunud seletuste kõrvale selgitused näidete abil.

Päriluseta näide

Alustame inimese klassiga, kus igasugune pärilus puudub. Üks klass oma muutuja, konstruktori ja meetodiga ning testprogrammis saab tema võimalusi katsetada.

```
using System;
namespace Parilus1{
 class Inimene{
   protected int vanus;
   public Inimene(int uvanus){
      vanus=uvanus;
   }
   public void YtleVanus(){
      Console.WriteLine("Minu vanus on "+vanus+" aastat");
   }
 }
 class InimTest{
  public static void Main(string[] arg){
     Inimene inim1=new Inimene(13);
     inim1.YtleVanus();
  }
 }
}
/*
C:\Projects\oma\naited>Parilus1
Minu vanus on 13 aastat
*/
```

Alamklass

Edasi juba juures inimese alamklass (subclass) `Modell`, kel on olemas kõik inimese omadused (ehk siis selle programmi puhul võime oma vanust öelda), kuid juures käsklus enese esitlemiseks. Et esitlemine koosneb vanuse ja ümbermõõdu teatamisest, on see juba esitlemiskäskluse sisse kirjutatud.

Klass `Inimene` on `Modelli` suhtes ülemklassiks (superclass, base class, parent class). C# puhul on igal klassil alati täpselt üks ülemklass. Kui muud pole määratud, siis kasutatakse selleks vaikimisi nimeruumi `System` klassi `Object`. Muul juhul aga on ülemklass kirjas klassikirjelduse päriluse osas. Nii tekibki klasside puu, mis algab klassist `Object`.

Kui klassil konstruktor puudub, siis loob kompilaator vaikimisi tühja konstruktori, millel pole parameetreid ja mis ka midagi ei tee. Inimese puhul siis näiteks

```
   public Inimene(){}
```

Kui aga vähemalt üks programmeerija loodud konstruktor on olemas, siis seda nähtamatut konstruktorit ei tehta. Päriluse puhul kutsutakse alamklassi eksemplari loomisel alati välja ülemklassi konstruktor. Vaikimisi võtab kompilaator selleks ülemklassi parameetritega konstruktori. Kui see aga puudub või soovitakse käivitada mõnda muud, siis tuleb sobiva konstruktori väljakutse alamklassi juures ära märkida. Siin märgitakse näiteks `Modelli` loomise juures, et `Modelli` isendi loomise juures tehakse kõigepealt valmis baasklassi (inimese) isend, kellele siis `Modelli` enese konstruktoris vajalikud lisandused juurde pannakse. Ülemklassi konstruktori määramine on kohe `Modelli` konstruktori juures. Pärast koolonit olev `base(vanus)` ütleb, et kasutatagu inimese puhul seda konstruktorit, kus tuleb täisarvuline vanus kohe ette öelda.

```
   public Modell(int vanus, int uymberm66t):base(vanus){
      ymberm66t=uymberm66t;
   }
```

Ehkki praegu tegelikult muud võimalust polnudki, tuleb see ikkagi arvutile ette öelda.

```
using System;
namespace Parilus2{
 class Inimene{
   protected int vanus;
   public Inimene(int uvanus){
      vanus=uvanus;
   }
   public void YtleVanus(){
      Console.WriteLine("Minu vanus on "+vanus+" aastat");
   }
 }
 class Modell:Inimene {
   protected int ymberm66t;
   public Modell(int vanus, int uymberm66t):base(vanus){
      ymberm66t=uymberm66t;
   }
   public void Esitle(){
      YtleVanus();
      Console.WriteLine("Mu ymberm66duks on "+ymberm66t+" sentimeetrit");
   }
 }
 class InimTest{
  public static void Main(string[] arg){
     Modell m=new Modell(20, 90);
     m.Esitle();
  }
 }
}
/*
C:\Projects\oma\naited>Parilus2
```

```
Minu vanus on 20 aastat
Mu ymberm66duks on 90 sentimeetrit
*/
```

Ülesandeid

* Lisa Inimesele pikkuse väli.

* Pikkus tuleb sisestada konstruktoris sarnaselt vanusele.

* Modelli käsklus Esitle teatab eraldi lausena ka pikkuse.

* Omista Modell Inimese tüüpi muutujale. Küsi sealtkaudu vanus.

* Koosta inimeste massiiv. Lisa sinna nii Modelle kui tavalisi inimesi. Küsi kõikide vanused.

* Lisa Inimesele int-tagastusväärtusega käsklus pikkuse väljastamiseks.

* Lisa testprogrammi käsklus static bool KasMahubAllveelaeva(Inimene isik), mis väljastab tõese väärtuse juhul, kui pikkust on alla 165 sentimeetri. Katseta käsku nii Inimese kui Modelli puhul, samuti Modellidest ja Inimestest koosneva massiivi juures.

* Väljasta false-vastus ka null-sisendi korral. Allveelaeva luba ka kuni 170 sentimeetri pikkused modellid (painduvad hästi, kontrolliks isik is Modell).

* Loo mõlemale klassile taas ka ilma pikkuseta konstruktor. Sellisel juhul pannakse pikkuse väärtuseks -1 ning esitluse juures teatatakse, et pikkuse andmed puuduvad.

Ülekate

Mõnikord ei piirduta omaduste lisamisega - tahetakse olemasolevaid võimalusi ka muuta. Tavanäiteks on kujundid või graafikakomponendid, mis igaüks ennast vastavalt oma omadustele joonistavad. Aga sarnaselt üle kaetavad võivad olla voogudesse andmete kirjutamise käsklused või andmestruktuuridesse andmeid lisavad või sealt eemaldavad käsklused nii, et neid käsklusi kasutav programm võib lihtsalt kasutatavat tükki usaldada, et tal on vastava nimega käsklus olemas ning ta selle soovitud ajal sooritab.

Siin näites on `Inimese` alamklassiks tehtud `Daam`, kel kõik muud kirjeldatud omadused hariliku Inimesega sarnased. Kuid vanuse ütlemine käib mõnevõrra teisiti. Et käske saaks rahus üle katta, selleks on `Inimese` juurde käsu ehk meetodi `YtleVanus` ette lisatud sõna `virtual`, `Daami` vastava meetodi ette aga `override`. Selliselt on mõlemat tüüpi objektil vanuse ütlemine selge, need lihtsalt käituvad erinevalt.

```
using System;
namespace Parilus3{
 class Inimene{
   protected int vanus;
   public Inimene(int uvanus){
      vanus=uvanus;
   }
   public virtual void YtleVanus(){
```

```
      Console.WriteLine("Minu vanus on "+vanus+" aastat");
    }
  }
  class Daam:Inimene {
    public Daam(int vanus):base(vanus){}
    public override void YtleVanus(){
      Console.WriteLine("Minu vanus on "+(vanus-5)+" aastat");
    }
  }
  class InimTest{
   public static void Main(string[] arg){
      Inimene inim1=new Inimene(40);
      Daam inim2=new Daam(40);
      inim1.YtleVanus();
      inim2.YtleVanus();
   }
  }
}

/*
C:\Projects\oma\naited>Parilus3
Minu vanus on 40 aastat
Minu vanus on 35 aastat
*/
```

Ülesandeid

* Lisa Inimesele käsklus "KutsuEttekandja", katseta seda eksemplari juures.

* Katseta sama käsklust ka Daami eksemplari juures.

* Kata nimetatud käsklus Daami juures üle, pannes sisse pidulikum ja peenem tekst. Katseta.

* Loo Inimeste massiiv, kus on nii Inimesed kui Daamid. Palu igaühel teatada oma vanus ning kutsuda ettekandja.

Liidesed

Nagu eespool kirjeldatud, on C# puhul üks ja kindel objektitüüpide pärinemise puu. Igal klassil on oma ülemklass ning juureks on klass `Object` nimeruumist `System`. Samas aga mõnegi tüübi puhul on sellest klassist objekte mugav kasutada tunduvalt rohkemates kohtades, kui otsene päriluspuu ette näeb. Nii nagu Ambla kihelkonna Lehtse valla Läpi küla Troska talu perepoeg Karl oli lisaks nendele ametlikele tiitlitele veel küla sepp, puutöömees ning korralik vanapoiss, nii on hea mõnegi klassi puhul programmi ülesehituse lihtsuse huvides pakkuda sinna rohkem rolle kui otsese päriluse järgi neid kokku tuleks. Näiteks Karlal oli leivateenimise mõttes sepatööst kindlasti rohkem kasu kui teatest, et ta Ambla kirikuraamatus kirjas on. Niisamuti on klassidele võimalik juurde panna liideseid, mis annavad õiguse vastava klassi eksemplare kloonida, järjestada või muid kasulikke omadusi lisada. Liideste arv ühe klassi juures ei ole piiratud. Liidesed mõeldi programmeerimiskeelte juurde välja, kuna selgus, et nõnda kirjeldusi määrates ja kokku leppides õnnestub programmeerimisel vigu vähendada.

Järgnevas näites loodi liides `IViisakas`. I on liidesel ees sõnast Interface. Liidese juurde käib enamasti sisuline seletus selle kohta, millised seda liidest realiseerivad objektid on. Ning lisaks võivad

olla mõned käsklused, millele vastavat liidest realiseerivad objektid on võimelised reageerima. Liidese `IViisakas` puhul valiti selliseks käskluseks `Tervita`, mis saab omale tekstilise parameetri. Ehk siis eeldatakse, et iga viisakas tegelane mõistab tervitusele vastata juhul, kui talle öeldakse, kellega on tegemist. Nagu näha - `Lapse` ja `Koera` puhul need tervitused on erinevad. Kuid nii nagu liides nõuab - nad on olemas. Sinnamaani suudab kompilaator kontrollida. Ülejäänu on juba programmeerija hoolitseda, et kindlaksmääratud nimega käskluse juurde ka vastavale klassile sobiv sisu saaks.

```
  static void TuleSynnipaevale(IViisakas v){
     v.Tervita("vanaema");
  }
```

Loodud meetodi juures on näha, et parameetrina võetakse vastu vaid nende klasside eksemplare, kelle puhul liides `IViisakas` on realiseeritud. Ning igaühel neist palutakse tervitada vanaema. Kuidas keegi sellega hakkama saab, on juba klassi enese sees hoolitsetud.

```
using System;
namespace Liides1{
 class Inimene{
   protected int vanus;
   public Inimene(int uvanus){
      vanus=uvanus;
   }
   public virtual void YtleVanus(){
      Console.WriteLine("Minu vanus on "+vanus+" aastat");
   }
 }
 interface IViisakas{
   void Tervita(String tuttav);
 }
 class Laps:Inimene, IViisakas {
   public Laps(int vanus):base(vanus){}
   public void Tervita(String tuttav){
     Console.WriteLine("Tere, "+tuttav);
   }
 }
 class Koer: IViisakas{
   public void Tervita(String tuttav){
      Console.WriteLine("Auh!");
   }
 }
 class InimTest{
  static void TuleSynnipaevale(IViisakas v){
     v.Tervita("vanaema");
  }
  public static void Main(string[] arg){
     Laps juku=new Laps(6);
     juku.YtleVanus();
     Koer muki=new Koer();
     TuleSynnipaevale(juku);
     TuleSynnipaevale(muki);
  }
 }
}
/*
C:\Projects\oma\naited>Liides1
```

```
Minu vanus on 6 aastat
Tere, vanaema
Auh!
*/
```

Kui mingil põhjusel jäänuks Lapsele või Koerale käsklus Tervita lisamata, siis annaks kompilaator veateate. Sama tekiks ka juhul, kui käskluse tekstis trükiviga tehtaks. Selline kompilaatoripoolne kontroll aitab vead üles leida juba enne tegelike andmetega katsetamise alustamist.

Ülesandeid

* Katseta, mis juhtub, kui Lapse tervita kirjutada väikese tähega.

* Lisa liidesesse IViisakas käsklus KoputaUksele.

* Muuda liidest realiseerivaid klasse nii, et kood kompileeruks.

* Testi töö tulemust.

* Koosta liides ISummeerija käsklustega Alusta, Lisa ning KysiSumma. Esimene siis tühjendab andmed, teine lisab ühe väärtuse ning kolmas väljastab olemasolevate summa.

* Koosta liidest realiseeriv klass, kus on muutuja summa hoidmiseks. Alustamise peale pannakse see nulli, lisamise puhul suurendatakse väärtust ning summa küsimisel väljastatakse meeles olev summa. Omista klassi eksemplar liidese tüüpi muutujale. Katseta.

* Koosta sama liidest realiseerima teine klass, kus lisatavate andmete jaoks on olemas massiiv. Kogu aeg peetakse meeles lisatud väärtuste arv ning väärtused ise. Summa küsimisel arvutatakse kokku elementide väärtuste summa ning väljastatakse see.

* Koosta eraldi peaprogramm, mis eeldab, et seal sees on kasutada ISummeerija liidest realiseeriv objekt – tegelikult aga algul on seal tühiväärtus-null. Peaprogramm koostatakse nõnda, et ta küsib kasutajalt arve ja palub ISummeerijal need kokku liita. Kontrolli, et kood kompileeruks.

* Koosta libasummeerija, mille Alusta ja Lisa-käsklused ei tee midagi. KysiSumma aga väljastab alati väärtuse -1. Katseta peaprogrammi loodud libaklassi objektiga.

* Loo eraldi klass SummeerijaVabrik summeerijaobjektide tootmiseks. Lisa sinna sisse staatiline käsklus LooSummeerija(int EeldatavKogus). Kui kogus on alla kümne, väljastatakse andmeid salvestav summeerija, muul juhul kohe andmeid kokku liitev summeerija. Katseta süsteemi toimimist.

Abstraktne klass

Liideseid pidi pärinevad vaid meetodite nimed. Klasside kaudu pärinevad nimed koos sisuga. Aga mõnikord soovitakse vahepealset varianti: et mõnedel meetoditel on olemas sisu, teistel aga veel mitte. Sellisel juhul aitab välja abstraktne klass. Tüüpiline näide on toodud allpool. Sirgete ja püstiste seintega kujundi puhul saab ruumala arvutada juhul, kui on teada põhja pindala ja kõrgus - valemi järgi piisab vaid nende omavahelisest korrutamisest. Kui aga kujundeid on palju, siis on niru seda

valemit igale poole uuesti kirjutada. Rääkimata juhtudest, kus valem tunduvalt keerukam on, või neid iga kujundi kohta mitu.

Siin on kujundi näidetena toodud `Tikutops` ja `Vorstijupp`. Esimesel neist on standardmõõtude puhul suurus kohe teada, vastavad meetodid väljastavad kohe konkreetsed arvud. Teisel juhul antakse mõõtmed objekti loomisel ette, meetodid peavad küsimise peale need salvestatud väärtused välja andma. Ning kui vorst ilusti matemaatiliselt omale silindrina ette kujutada, siis annab pii korda raadiuse ruut ilusti põhja ehk ristlõike pindala välja.

Klassist `Kujund` enesest ei saa eksemplare luua. Kui klass sisaldab abstraktseid ehk defineerimata meetodeid, siis peab klass ka ise tervikuna olema abstraktne ning siis ei lubata temast eksemplare teha. Muidu ju tekiks probleem, kui soovitaks käivitada kujundi olematu koodiga käsklust `KysiKorgus`().

Küll aga tohib muutujale tüübist `Kujund` tegelikke objekte omistada - olgu nad siis `Tikutopsid`, `Vorstijupid` või pärit mõnest muust `Kujundi` alamklassist. Sarnaselt nagu võis `Daami` omistada `Inimese` tüüpi muutujale või `Lapse` muutujale tüübist `IViisakas`. Kui üle kaetud klassis on eelnevalt abstraktsetele meetoditele sisu antud, siis võib sellest klassist julgesti isendeid luua ning neid ka kõikidest ülemklassidest pärit muutujatele omistada.

```
using System;
namespace AbstraktseKlassiUuring{
 abstract class Kujund{
   public abstract double KysiPohjaPindala();
   public abstract double KysiKorgus();
   public double KysiRuumala(){
      return KysiPohjaPindala()*KysiKorgus();
   }
 }
 class Tikutops:Kujund{
   public override double KysiPohjaPindala(){return 8;}
   public override double KysiKorgus(){return 1.5;}
 }


 class Vorstijupp: Kujund{
   int pikkus, raadius;
   public Vorstijupp(int upikkus, int uraadius){
      pikkus=upikkus;
      raadius=uraadius;
   }
   public override double KysiPohjaPindala(){
     return Math.PI*raadius*raadius;
   }
   public override double KysiKorgus(){
     return pikkus;
   }
 }
 class Test{
   public static void Main(string[] arg){
      Tikutops t=new Tikutops();
      Vorstijupp v=new Vorstijupp(10, 3);
      Console.WriteLine("Ruumalad {0} ja {1}",
                t.KysiRuumala(), v.KysiRuumala());
   }
```

```
  }
}
/*
D:\kodu\0606\opikc#>AbstraktseKlassiUuring
Ruumalad 12 ja 282,743338823081
*/
```

Ülesandeid

* Lisa klassile Kujund abstraktne meetod PohjaYmbermoot ning meetod KyljePindala. Katseta - lisades vajalikud meetodid ka alamklassidesse.

* Loo Kujundi alamklass Risttahukas lisades talle vajalikud väljad kolme mõõtme hoidmiseks ja kattes üles Kujundi abstraktsed meetodid. Katseta mitmesuguste Risttahuka eksemplaridega.

* Koosta mitmesuguste Kujundite massiiv. Loo alamprogramm leidmaks massiivis olevate kujundite ruumalade summa. Loo eraldi alamprogramm leidmaks massiivis olevate kujundite pindalade summa.

Meetodite asendus

Harilikult kirjutatakse meetodite üle katmise juures ülemklassi meetodi ette virtual ning alamklassi juurde override. Sellisel juhul alamklassi (siinses näites `Daami`) objekti puhul kasutatakse alati seda meetodit, mis tema juurde käib - sõltumata, millisest tüübist on muutuja, mille kaudu eksemplari poole pöördutakse. C++ võimalusi säilitades aga on jäetud ka teine võimalus. Meetodi võib asendada nõnda, et tema kirjeldamise ette kirjutatakse sõna `new`. Sel juhul peidetakse vana meetod samuti ära. Vana meetodi saab aga kätte juhul, kui objekti poole pöörduda ülemklassi muutujast, kus vastav meetod vanal kujul kasutusel oli. Kui `virtual`/`override` puhul pidid parameetrid ja väljastustüüp samaks jääma, siis `new` loob täiesti uue ja eelmisest sõltumatu meetodi.

Järgnevas näites on ehitatud kunstlik pärilusahel. Ülemklassiks `Inimene`, kes ütleb oma vanuse nõnda nagu see on. `Inimesest` pärinenud `Daam` võtab ilma pikemalt mõtlemata 5 aastat maha. `Daami` alamklassiks olev `Beib` keeldub üldse vanuse teatamisest ning eriti kaugele arenenud `KavalBeib` palub kasutajal ise tema vanust pakkuda. Sõna `sealed` klassi juures näitab, et sellest klassist ei lubata enam edasi pärida. Selline määrang aitab kompilaatoril koodi optimeerida.

Alljärgnevalt katsetatakse, millist tüüpi muutuja kaudu millise tegeliku objekti poole pöördumisel milline tulemus saadakse. Et omistamine on võimalik ainult ülemklassi suunas, siis igaühe neist saab omistada `Inimese` tüüpi muutujale. Mida tase edasi, seda vähem on omistusvõimalusi.

Katsetamise käigus antakse `Beiblastele` vanuseks 17 aastat, teistele 40. Ning jälgitakse, milline meetod millise muutuja kaudu väljakutsel käima läheb. Kõige pikem ja keerukam lugu on Kavala Beibega. Et ta on pärimispuus kõige kaugemal, siis teda on võimalik omistada kõikide selles puus olevate tüüpidega muutujatele. Enese tüübi puhul küsib ta vanuseks 19, nagu käskluses öeldud. Ka lihtsalt Beib-tüüpi muutuja kaudu küsib ta enesele 19, sest klassi `Beib` meetod `YtleVanus` on virtual ning tegelikult käima läheb objekti enese ehk klassis `KavalBeib` loodud meetod.

Edasi muutub lugu keerulisemaks. `Daami`-muutujast välja kutsutav Kavala Beibe vanus tuleb 12, sest ta lahutab aastad maha nagu Daamile kohane. Ning ka hariliku inimese kaudu tuleb 12, sest `virtual`-piiritleja kaudu võetakse käsklus võimalikult objekti enese lähedalt.

```
using System;
namespace Asendus{
 class Inimene{
   protected int vanus;
   public Inimene(int uvanus){
      vanus=uvanus;
   }
   public virtual void YtleVanus(){
      Console.WriteLine("Minu vanus on "+vanus+" aastat");
   }
 }
 class Daam:Inimene {
   public Daam(int vanus):base(vanus){}
   public override void YtleVanus(){
      Console.WriteLine("Minu vanus on "+(vanus-5)+" aastat");
   }
 }
 class Beib:Daam{
   public Beib(int vanus):base(vanus){}
   new public virtual void YtleVanus(){
       Console.WriteLine("Minu vanus pole sinu asi, vot!");
   }
 }
 sealed class KavalBeib:Beib{ //siit ei saa enam edasi areneda
   public KavalBeib(int vanus):base(vanus){}
   public override void YtleVanus(){
      Console.WriteLine("Arva, kas olen {0}?", vanus+2);
   }
 }
 class InimTest{
  public static void Main(string[] arg){
     KavalBeib kb=new KavalBeib(17);
     Beib b=new Beib(17), bkb=kb;
     Daam d=new Daam(40), db=b, dkb=kb;
     Inimene i=new Inimene(40), id=d, ib=b,ikb=kb;
     kb.YtleVanus();
     b.YtleVanus();
     bkb.YtleVanus();
     d.YtleVanus();
     db.YtleVanus();
     dkb.YtleVanus();
     i.YtleVanus();
     id.YtleVanus();
     ib.YtleVanus();
     ikb.YtleVanus();
  }
 }
}

/*
D:\kodu\0606\opikc#>Asendus
Arva, kas olen 19?
Minu vanus pole sinu asi, vot!
Arva, kas olen 19?
Minu vanus on 35 aastat
```

```
Minu vanus on 12 aastat
Minu vanus on 12 aastat
Minu vanus on 40 aastat
Minu vanus on 35 aastat
Minu vanus on 12 aastat
Minu vanus on 12 aastat
*/
```

Ülesandeid

* Loo klass Punkt väljadega x ja y ning meetoditega KaugusNullist ja TeataAndmed. Esimene väljastab reaalarvuna kauguse koordinaatide alguspunktist. Teine tagastab tekstina koordinaatide väärtused.

* Loo Punktile alamklass RuumiPunkt. Lisa väli z, kata üle KaugusNullist ning asenda TeataAndmed. Esimene väljastab kauguse nullist kolme koordinaadi korral, teine aga kirjutab RuumiPunkti andmed ekraanile, meetodid tagastustüübiks on void.

* Katseta loodud objekte ja nende käsklusi igal võimalikul moel. Punkt Punkti muutujast, RuumiPunkt RuumiPunkti muutujast ning RuumiPunkt Punkti muutujast.

Omadused

Objektide juures tehakse enamasti selget vahet: väljad ehk muutujad on andmete hoidmiseks, käsud ehk funktsioonid ehk meetodid toimingute sooritamiseks, muuhulgas ka andmete poole pöördumiseks. Ning korralikult kapseldatud objektorienteeritud programmis pääseb väljadele otse ligi vaid objekti seest, kõik muud välised toimetused käivad meetodite kaudu.

Kahjuks või õnneks on enamik programmeerijaid kirjutanud ka objektikaugemat koodi. Kes sellepärast, et tema kooliajal polnud veel objektorienteeritus kuigi laialt levinud või mõni teine põhjusel, et piisavalt väikeste programmide puhul võib olla objektindusest rohkem tüli kui tulu. Nõnda pakuvad programmeerimiskeeled mitmesuguseid "vahevorme", kus püütakse kasu lõigata objektide süstemaatilisusest ning samal ajal jätta alles protseduuridel põhineva programmeerimise lihtsuse.

Eelpool oli selliseks heaks mooduseks kirje (`struct`). Kirje puhul eeldatakse, et ta on loodud põhiliselt andmete hoidmiseks, andmete õigsuse ja kokkusobivuse eest hoolitseb väline programm, et tegemist on lihtsalt muutujate komplektiga nagu näiteks punkti koordinaadid. Kirje vaid hoolitseb, et `x` ja `y` alati kokku kuuluksid.

Siiski on erinevalt mõnest muust keelest C# puhul lubatud kirje juurde ka käsklusi lisada. Enamasti kasutatakse neid arvutatavate omaduste - nagu näiteks sünniaja järgi vanuse - leidmiseks. Kirje kasutamise teeb objektist mugavamaks käskude mõningane lühidus. Seal võib rahulikult kirjutada

```
p1.X=17;
int a=p1.X;
```

Objektide puhul peetakse sellist otse muutujate poole pöördumist ebaviisakaks. Paremaks lahenduseks soovitatakse

```
p1.paneX(17);
int a=p1.KysiX();
```

On küll vaid paar tähte juures, aga piisavalt selle jaoks, et laisad programmeerijad vahel pigem muudavad objekti muutujad otse ja avalikult kättesaadavaks, kui et andmeid viisakalt meetodite kaudu vahetavad. Et lühidat kirjastiili säilitada ning samas jätta alles meetoditele iseloomulik kontrollitavus ja muudetavus, selleks ongi loodud võimalus objektile luua omadusi, mis näevad välja nagu muutujad, kuid käituvad nagu meetodid.

Järgnevas näites on klassi `Ilmaandmed` eksemplaridele lisatud omadus `Temperatuur`, millele saab väljapoolt väärtust omistada ning küsida nagu tavaliselt muutujalt ehk väljalt. Küsimise peale pannakse lihtsalt tööle `get`-koodiosa ning omistamise peale `set`-koodiosa. Sõna "`value`" tähistabki omistamisel antud väärtust. Koodi ülesandeks on vastava märksõna all saabunud andmed vajalikku kohta paigutada. Tegelikke andmeid hoitakse privaatses ehk väljapoolt nähtamatus muutujas nimega `temper`. Vajadusel saab nõnda omaduse või meetodi kaudu andmete poole pöördudes peale panna näiteks kontrolli võimalike väärtuste üle omistamisel. Või siis saab programmeerija otsustada hoopis, et temperatuure endid ei hoita mingil hetkel enam muutujas, vaid hoopis andmebaasis. Et kogu andmetega toimetamine on jäetud `get`- ja `set`-meetodite hooleks, siis on selline asendus täiesti võimalik.

```
using System;
namespace Omadused1{
 class Ilmaandmed{
   private int temper;
   public int Temperatuur{
      get{return temper;}
      set{temper=value;} //value on sisendväärtuse nimi
   }
 }
 class Test{
   public static void Main(string[] arg){
      Ilmaandmed jaam1=new Ilmaandmed();
      jaam1.Temperatuur=15;
      Console.WriteLine(jaam1.Temperatuur);
   }
 }
}

/*
D:\kodu\0606\dotnet>Omadused1
15
*/
```

Pöördumisstatistika

Siin ongi toodud võimalikult lihtne näide, kuidas peale omistamise/küsimise veel meelde jätta ja teada saada nende operatsioonide arv. Lihtsalt loendamise jaoks vastavad muutujad ja käsklused juures.

```
using System;
namespace Omadused2{
```

```
 class Ilmaandmed{
   private int temper;
   private int muudetud=0;
   private int loetud=0;
   public int Temperatuur{
      get{
        loetud++;
        return temper;
      }
      set{
        muudetud++;
        temper=value;
      }
   }
   public override string ToString(){
      return "Muudetud: "+muudetud+", loetud:"+ loetud+
                                      ", temperatuur:"+temper;
   }
 }
 class Test{
   public static void Main(string[] arg){
      Ilmaandmed jaam1=new Ilmaandmed();
      jaam1.Temperatuur=15;
      Console.WriteLine(jaam1.Temperatuur);
      Console.WriteLine(jaam1.Temperatuur);
      Console.WriteLine(jaam1);
   }
 }
}
/*
C:\Projects\oma\naited>Omadused2
15
15
Muudetud: 1, loetud:2, temperatuur:15
*/
```

Ülesandeid

* Kui temperatuuriks märgitakse üle 35, trüki hoiatusteade kahtlase väärtuse kohta

* Loo klass Kasutaja. Kasutajanimi määratakse konstruktoris, ning seda on hiljem võimalik ainult omadusena küsida, mitte muuta (`set`-osa puudub). Parooli saab ainult määrata, aga küsida pole võimalik (`get`-osa puudub). Lisa käsklus parooli kontrolliks. Lisa omadusena kasutaja telefoninumber, mida on võimalik küsida ja muuta. Kontrolli, et töötaksid vaid määratud tehted (st. et kord pandud parooli ei saaks küsida).

Indekseering

Massiivide puhul oleme harjunud, et kandiliste sulgude ja järjekorranumbri järgi küsime või seame omale väärtuse. C# süntaks lubab aga ka ise kirjeldatud objektidel omapoolsed käsud sellise pöördumise peale tööle panna. Iseenesest oleks võimalik kõik sellised toimetused mõne hariliku funktsiooni ehk meetodiga ära ajada, aga kandiliste sulgudega kirja panduna tehe näeb lühem välja. Ning mõnigi kord, kui andmed paistavad sarnased nagu massiivides hoitakse, võimaldab selline kantsulgudega tehe koodi ka harjunumalt kirja panna.

Esimeses, võimalikult lihtsas näites väljastatakse indekseerimistehte tulemusena etteantud arvu ruut. Kirjaviis on mõnevõrra sarnane omaduse kirjeldamisele: `get`-osas tuleb soovitud väärtus `return`i abil tagasi anda.

```
using System;
namespace Indekseering1{
 class Ruuduarvutus{
   public int this[int nr]{
      get{return nr*nr;}
   }
 }
 class Test{
  public static void Main(string[] arg){
     Ruuduarvutus r=new Ruuduarvutus();
     Console.WriteLine(r[3]);
  }
 }
}
/*
C:\Projects\oma\naited>Indekseering1
9
*/
```

Vahendus

Mõnelgi korral võib omaloodud indekseerimisel kasutada juba olemasolevat massiivi või muud andmekogumit. Ning indekseerimise kaudu saab lisada selle elementide kasutamisele mõne piirangu või ümberarvutusfunktsiooni. Siinses näites ehitati indekseerimise abil kest ümber paisktabelile - paaride kogumile, kus igas paaris on võti ja väärtus. Võtmeks on kuupäev, väärtuseks asukoht, kus vastaval päeval ansambel esineb. Andmete küsimisel vastatakse veel vaba päeva kohta küsimisel "Vaba", kinnipandud päeva puhul teatatakse, kus vastaval päeval esinemine on. Nimeruumis `System.Collections` asuval `Hashtable` klassist objektil on andmete salvestamise ja küsimise käsklused juba sisse ehitatud. Lihtsalt `Hashtable` annab vastava võtme puudumisel vastuseks tühiväärtuse `null`, meie aga vastame selle peale inimkeelse "Vaba" ning andmete salvestamise juures juhul kui vastav kuupäev kinni on heidetakse erind veateatega, miks vastav päev ei sobi. Kui aga soovitud kuupäev on veel vaba, siis pannakse sinna juurde ilusti sobiv väärtus kirja.

```
using System;
using System.Collections;
namespace Indekseering2{
 class Ringreis{
   Hashtable esinemised=new Hashtable();
   public string this[int kuupaev]{
      get{
         if(esinemised[kuupaev]==null){return "Vaba";}
         return (string)esinemised[kuupaev];
      }
      set{
         if(esinemised[kuupaev]!=null){
            throw new Exception("Juba kinni, esinemine linnas "+
                                      esinemised[kuupaev]);
         }
         esinemised[kuupaev]=value;
```

```
      }
    }
  }
  class Test{
   public static void Main(string[] arg){
      Ringreis r=new Ringreis();
      r[3]="Narva";
      r[4]="Tartu";
      Console.WriteLine(r[3]);
      Console.WriteLine(r[5]);
      r[3]="Viljandi";
   }
  }
}
/*
C:\Projects\oma\naited>Indekseering2
Narva
Vaba
Unhandled Exception: System.Exception: Juba kinni, esinemine linnas Narva
   at Ringreis.set_Item(Int32 kuupaev, String value)
   at Test.Main(String[] arg)
*/
```

Ülesandeid

* Muuda arvu ruutu väljastavat indekseerimise näidet nii, et see väljastaks etteantud arvu kuubi.

* Loo objekt, mis võtaks konstruktoris vastu sõna. Väljasta nii mitmes täht, kui indeksiga näidatakse. Kui arv ületab sõna pikkuse, siis antakse teada, et sellise järjekorranumbriga tähte ei leidu. Kui indeksiks pandud arv on negatiivne, siis väljastatakse niimitmes täht sõna lõpust arvates.

* Loo indekseeringu kaudu kasutatav seitsmeelemendiline massiiv vastaval nädalapäeval tehtud töötundide arvu summeerimiseks. Igal omistamisel liidetakse töötunnid vastava päeva arvule otsa. Igal küsimisel antakse välja sellele nädalapäevale liidetud töötundide summa. Negatiivse arvu sisestamisel tuleb veateade.

Struktuurne andmestik

Mitmedki toimetused saab tehtud ühe klassiga. Vajalikud andmed algul konstruktoris sisse ning käskudega kannatab neid küsida, lisada ja muuta nagu parajasti sobiv ja võimalik on. Kui on tegemist korraga mitme sarnase ülesehituse, kuid erinevate väärtustega andmetega (näiteks mitme isikukoodiga), siis võib loodud klassist luua iga väärtuse jaoks omaette objekti ja sealtkaudu siis mõningast analüüsi vajavate andmetega ümber käia. Kui aga andmeid on palju, mitut tüüpi ja omavahel seotud, siis tasub mõelda nende omavahelise struktuuri peale.

Programmeerimist saab üldjuhul õppida „pastakast välja imetud” lihtsate näidete peal. Kui aga mõnda teemasse sisse minna, siis tuleb erialateadmised ja programmeerimisoskused ühendada. Mõlemad võivad eraldi võttes olla küllaltki lihtsad, kuid nende kokkupanek võib veidi peamurdmist nõuda. Samas aga teadmiste ja tehnika ühendamine võib anda eraldi toimetamisest märksa kasulikumaid tulemusi.

Järgnevas näites mängitakse läbi elektriskeemides loodetavasti põhikooliajast tuttavate takistite ja nende ühendamisega ette tulevad arvutused. Elektroonikule on seostub takistiga tõenäoliselt sageli punast värvi väike silinder, mille mõlemast otsast traat välja tuleb. Iseenesest aga käituvad takistina ka tavalised lambipirnid ja mõned muudki elektroonikaseadmed. Lihtsal takistil on keskkonnaoludest suhteliselt sõltumatu väärtusega takistus. Samuti on tal lubatud suurim eralduv võimsus, mille ületamisel võib takistist välja tulla „hall mull" ehk tossupilv ning komponent pole pärast seda enam kasutatav. Takistuseks nimetatakse juhtmeotstel ehk klemmidel oleva pinge (surve, voltides) ning takistit läbiva voolu (elektronide voog, amprites) suhet. Mida suurem takistus, seda vähem läheb sama pinge juures takistist voolu läbi. Takistil soojusena eralduv võimsus (wattides) leitakse takisti klemmidele pandud pinge ning takistit läbiva voolu korrutisena. Põhivalemid siis:

U/I=R

U*I=N

Kui veidi avaldada, siis leiab sealt, et $I = \sqrt{\frac{N}{R}}$. Ehk siis teadaoleva lubatud maksimumvõimsuse ja takistuse põhjal on võimalik leida takistit läbi suurim lubatud vool.

Takisteid saab omavahel kombineerida. Tüüpilised ühendused on järjestikku (jadamisi) ja rööbiti (paralleelselt). Jadamisi ühendades on arvutuskäik lihtne – takistuste komplekti kogutakistus on võrdne ühendatud takistite takistuste summaga. R=R1+R2 või ka rohkem komponente üksteisele järele liidetult. Rööbiti on arvutuskäik veidi keerulisem, kuid ka mitte lootusetu. Kahe rööbiti takisti kogutakistus on väiksem kui kummalgi eraldi, kogutakistuse pöördväärtus on võrdne üksikute takistite takistuste pöördväärtuste summaga $\frac{1}{R} = \frac{1}{R_1} + \frac{1}{R_2}$ - esialgses programmis aga rööbiti ühendusega lihtsuse mõttes ei tegele.

Alustame kõige lihtsamast – loome klassi takisti andmete hoidmiseks ja sealt vajalike väärtuste küsimiseks või arvutamiseks vastavalt lisaandmetele. Takistil omal muutujateks takistus R ja maksimumvõimsus MaxN. Need antakse sisse ka konstruktoris. Vastavate andmete kättesaamiseks eraldi käsklused LeiaTakistus ja LeiaMaksimumV6imsus – kuna tegemist piiratud ligipääsuga (protected) muutujatega, siis muidu ei pruugi loodud objektist enam väärtusi näha. Otse muutujate poole pöördumine pole viisakas – ei võimalda programmeerijal hiljem enam loodud objektiga toimuvat kontrollida. Siin aga kui andmed antakse sisse konstruktoris ning hiljem on võimalik neid ainult küsida, siis pole muret, et takisti andmed seletamatul põhjusel muutuma hakkaksid. Juurde ka mõned käsklused voolu ja võimsuse arvutamiseks ning kontrollimiseks, et kas soovitud pinge või vool ka konkreetsele takistile lubatud on.

```
using System;
using System.Text;
namespace Takistid {
    class Takisti {
        /// <summary>
        /// Takistus
        /// </summary>
        protected double R;
        /// <summary>
        /// Maksimumvõimsus
```

```
        /// </summary>
        protected double MaxN;
        public Takisti(double Takistus, double Maksimumv6imsus) {
            this.R = Takistus;
            this.MaxN = Maksimumv6imsus;
        }
        public double LeiaVool(double Pinge) {
            return Pinge / R;
        }
        public double LeiaV6imsus(double Pinge, bool SobivusKontroll) {
            double V6imsus=Pinge * LeiaVool(Pinge);
            if (SobivusKontroll) {
                if (V6imsus > MaxN) {
                    throw new Exception("Pingel " + Pinge + " ületab võimsus " +
                       V6imsus + " lubatud maksimumvõimsust " + MaxN);
                }
            }
            return V6imsus;
        }
        public double LeiaV6imsus(double Pinge) {
            return LeiaV6imsus(Pinge, true);
        }
        public bool KasLubatudVõimsusVastavaltPingele(double Pinge) {
            return LeiaV6imsus(Pinge, false) <= MaxN;
        }
        public double LeiaMaksimumVool() {
            return Math.Sqrt(MaxN / R);
        }
        public double LeiaTakistus() {
            return R;
        }
        public double LeiaMaksimumV6imsus() {
            return MaxN;
        }
    }
}
```

Kui takisti klass valmis, siis on hea seda katsetada. Loome konkreetsete omadustega takisti – näiteks 5Ω takisti maksimumvõimsusega 2W – ehk siis ettekujutatuna ühe pisikese lapse näpuotsasuuruse jupikese, millest kaks juhet välja tulevad. Kontrollime, kas sellise takisti kannataks ühendada 1,5-voldise patarei taha. Programm arvutab ja teatab, et kannatab küll, väljundvõimsuseks 0,45 Watti. Ise järgi arvutades võime tulemust kontrollida. 1,5 volti jagatud 5 oomiga annab 0,3 amprit. 0,3 amprit korrutatuna 1,5 voldiga teebki 0,45 watti. Mis siis teeb küll takisti õrnalt soojaks, aga ei lõhu seda veel ära. Kui küsitaks peale tunduvalt suurem pinge, siis meie programm peaks teatama, et sellise pingega tekkiv võimsus pole lubatud.

```
using System;
using System.Text;
namespace Takistid {
    class TakistiProov {
        public static void Main(string[] arg) {
            Takisti t1 = new Takisti(5, 2); //5 oomi, 2 watti
            double PatareiPinge = 1.5; //volti
            if (t1.KasLubatudVõimsusVastavaltPingele(PatareiPinge)) {
                Console.WriteLine("Patareilt " + PatareiPinge + "V  saadakse " +
                    " takistiga " + t1.LeiaTakistus() + " oomi vool " +
                      t1.LeiaVool(PatareiPinge) + "A ja " +
                    " võimsus " + t1.LeiaV6imsus(PatareiPinge) + "W");
            } else {
                Console.WriteLine("Pingel "+PatareiPinge+"V tekkiv võimsus "+
                   t1.LeiaV6imsus(PatareiPinge, false)+"W ületab lubatud "+
```

```
                    "võimsust "+t1.LeiaMaksimumV6imsus()+"W");
                }
            }
        }
}

/*
Patareilt 1,5V  saadakse  takistiga 5 oomi vool 0,3A ja  võimsus 0,45W
*/
```

Ühe või kahe takistiga saab rahumeeli toimetada. Nendega ümber käimiseks pole isegi programmi vaja. Kui aga andmeid rohkem, siis tasub mõelda nende haldamise peale. Elektroonikutel on suuremate takistikogustega ümber käimiseks kasutada takistussalved. Sarnase vahendi kannatab ka programmi poole pealt valmis teha. Nii nagu programmiobjektid ühel või teisel moel jäljendavad reaalse maailma objektid omadusi, nii ka siinpool. Koostame takistite jadamisi ühendamiseks eraldi takistussalve. Nii nagu pärissalves on kindel arv kohti takistite jaoks, nii siin on massiivil ka kindel hulk mälupesi takistiobjektide jaoks. Pesade arv määratakse kindlaks salve konstruktoris. Salvest voolu läbilaskmisel on piirajaks kõige nõrgemat voolu kannatava takisti maksimumvool. Kui aga salv on tühi, ka siis ei või sellest lõpmatult suurt voolu läbi lasta. Siin näites on määratud tühja salve maksimumvooluks 16 amprit – selline korraliku koduse pikendusjuhtme läbilaskevõime. Salvel on käsklus takisti lisamiseks – esialgu aga veel mitte eemaldamiseks või väljavahetamiseks – seda lihtsalt lühiduse ja lihtsuse mõttes. Lisamisel jäetakse meelde lisatud takistite arv – siis teab, millisesse pessa järgmine lisatav takisti panna. Samuti on võimalik hoiatusteade anda juhul, kui salve enam takisteid ei mahu. Kogutakistuse leidmiseks liidetakse salves olevate takistite takistuste summad kokku. Maksimumvoolu leidmiseks leitakse vool, mida kannatab ka kõige nõrgem takisti ahelas. Lubatud pinge kontrollimiseks leitakse salve kogutakistuse järgi arvutatud vool vastavalt pingele ning kontrollitakse, kas see on piisavalt väike, et kõik salves olevad takistid selle välja kannataksid.

```
using System;
using System.Text;
namespace Takistid {
    class JadamisiTakistusSalv {
        Takisti[] Andmed;
        int TakistiteArv = 0;
        const int TyhjaSalveMaksimumvool = 16;
        public JadamisiTakistusSalv(int TakistiteMaksimumArv) {
            Andmed = new Takisti[TakistiteMaksimumArv];
        }
        public void LisaTakisti(Takisti t) {
            if (TakistiteArv < Andmed.Length) {
                Andmed[TakistiteArv] = t;
                TakistiteArv++;
            } else {
                throw new Exception("Takistussalv täis!");
            }
        }
        public double LeiaKoguTakistus() {
            double summa = 0;
            for (int i = 0; i < TakistiteArv; i++) {
                summa = summa + Andmed[i].LeiaTakistus();
            }
```

```
            return summa;
        }
        public double LeiaMaksimumVool() {
            double MaxVool = TyhjaSalveMaksimumvool;
            for (int i = 0; i < TakistiteArv; i++) {
                if (Andmed[i].LeiaMaksimumVool() < MaxVool) {
                    MaxVool = Andmed[i].LeiaMaksimumVool();
                }
            }
            return MaxVool;
        }
        public bool KasLubatudV6imsusVastavaltPingele(double Pinge)
{
            double TekkivVool = Pinge / LeiaKoguTakistus();
            return TekkivVool <= LeiaMaksimumVool();
        }
    }
}
```

Iga loodud klassi on ka hea katsetada. Siis julgem tunne, et tulevikus klassi tööd usaldada võib. Vastutusrikkamatel kohtadel tuleb läbi proovida igasugu kombinatsioonid ja võimalikud erijuhud. Ning seal võib testprogrammide kirjutamine olla paar-kolm korda keerukam kui rakenduse enese loomine. Siin aga lihtsalt proovime, kas salve loomine õnnestub. Paar takistit sisse ja väike arvutus kogutakistuse, maksimumvoolu ja lubatud pinge kohta. Nagu näha, salv loodi viiele takistile, sinna sisse paneme praegu ainult kaks. Üks kümneoomine ühevatine ning teine kümneoomine kahevatine. Siis aimatavalt on kogutakistus kakskümmend oomi ehk kahe takisti takistuste summa.

```
using System;
namespace Takistid {
    class TakistusSalveProov {
        public static void Main(string[] arg) {
            JadamisiTakistusSalv Salv = new JadamisiTakistusSalv(5);
            Salv.LisaTakisti(new Takisti(10, 1));
            Salv.LisaTakisti(new Takisti(10, 2));
            Console.WriteLine("Kogutakistus: "+Salv.LeiaKoguTakistus());
            Console.WriteLine("Maksimumvool: " + Salv.LeiaMaksimumVool());
            Console.WriteLine("Kas 12 Volti salve klemmidel lubatud: " +
                                Salv.KasLubatudV6imsusVastavaltPingele(12));
        }
    }
}
/*
Kogutakistus: 20
Maksimumvool: 0,316227766016838
Kas 12 Volti salve klemmidel lubatud: False
*/
```

Ülesandeid

* Tutvu näidetega, vaheta väärtusi, kontrolli vastuste usaldusväärsust.

* Lisa takistiklassile käsklus, kus leitakse etteantud voolu tekitamiseks vajalik pinge. Katseta

* Lisa sama käsklus takistussalvele

* Kontrolli takisti lisamisel salve, et sama takisti ei oleks seal juba olemas.

Ühine ülemklass

Kui eelmist näidet tähelepanelikult jälgida, siis paistab, et nii takistussalve kui takisti juures on sarnaseid käsklusi. Voolu järgi pinge või pinge järgi voolu leidmise valem on ikka sarnane – tuleb ainult omale selgeks teha, mida see takistus parajasti tähendab. Samuti ei tohi ei üksikule takistile ega salvele panna peale suuremat pinget, kui suurima lubatud voolu jaoks kõlbulik pinge on. Ning mis veel peenem – kui kord on kokku pandud takistussalv sobiva suurusega takistusega, siis võib selle salve juhtmeotsad hea tahtmise korral ühendada teise salve ühe takisti kohale. Nõnda kombineerides saab salvedest ja takistitest kokku ühendada päris paindliku süsteemi. Ja et see päriselus võimalik on, siis on programmeerimiskeelte loojad teinud kõik, et ka programmiklassidega maailma järele tehes sellise paindlikkuse kokku saaks. Nagu alljärgnevalt näha, see ka õnnestub.

Takistile ja takistussalvele (ja võibolla veel mõnele muule elektronide voolamist pidurdavale seadmele, näiteks lambipirnile) loodi ühine ülemklass TakistusKomponent. Käsklused LeiaTakistus ja LeiaMaksimumVool on määratud abstraktseteks. Sest igal seadmel on selle leidmise moodus isesugune, vastavad omadused on aga igal elektriseadmel ehk takistuskomponendil olemas. Ülejäänud abstraktse klassi käsklustes võime takistuse ja maksimumvoolu lugeda juba teatuks ning konstrueerida muid käsklusi neid käske kasutades. Kui takistus teada, siis voolu saab pinge jagades takistusega. Kui maksimumvool teada ja pingele vastav vool arvutatav, siis saab kergesti järele kontrollida, kas komponendi klemmidele tohib olemasolevat pinget rakendada või mitte.

```
using System;
namespace TakistusKomponendid {
    abstract class TakistusKomponent {
        public abstract double LeiaTakistus();
        public abstract double LeiaMaksimumVool();
        public double LeiaVoolVastavaltPingele(double Pinge) {
            return Pinge / LeiaTakistus();
        }
        public bool KasLubatudPinge(double Pinge) {
            return LeiaVoolVastavaltPingele(Pinge) < LeiaMaksimumVool();
        }
    }
}
```

Takisti ise näeb pärast seda juba suhteliselt lihtne välja. Poest ostetud vidinale omased sisetakistus ja maksimumvõimsus tuleb ikka meelde jätta. TakistusKomponent kohustab üle katma käsklused LeiaTakistus ja LeiaMaksimumVool. Esimese saab kätte otse muutujast. Teise leidmise jaoks tuleb üleval tuletatud valemit rakendada.

```
using System;
namespace TakistusKomponendid {
    class Takisti:TakistusKomponent {
        protected double R, MaxN;
        public Takisti(double Takistus, double MaksimumV6imsus) {
            this.R = Takistus;
            this.MaxN = MaksimumV6imsus;
        }
        public override double LeiaTakistus() {
```

```
            return R;
        }
        public override double LeiaMaksimumVool() {
            return Math.Sqrt(MaxN / R);
        }
    }
}
```

Takistussalves tuleb ikka luua koht sinna sisse pandavate komponentide andmete hoidmiseks. Erinevalt eelnevast näitest aga nüüd on andmepesa tüübiks TakistusKomponent. See tähendab, et programmis lubatakse salve sisse panna ka teisi salvesid. Miski ei takista praegu salvel ka iseenese väljuvaid juhtmeid ühe oma takistikomponendi klemmidele ühendada – selline suhteline mõttetus jääks aga praegu lihtsalt programmeerija südametunnistusele. Kusjuures mõne programmi puhul pole iseenese andmete hoidmine sugugi mõttetu nähtus. Näiteks, kui grupijuht peab hoolitsema inimeste toiduportsude eest, siis peab ta kindlasti hoolitsema, et ta ka enese tarbeks supikausi muretseks. Kogutakistuse leidmisel leitakse üksikute pesade takistuste summad. Kui seal juhtuvad olema tavalised takistid, siis saadakse väärtused ühe käsuga muutujast kätte. Kui aga salve pesas juhtub olema teine salv, siis käsklus LeiaTakistus käivitab selle salve kogutakistuse leidmise käskluse. Samuti maksimumvoolu puhul leitakse eraldi iga komponendi maksimaalne lubatud vool. Ning kui mõneks komponendiks on salv, siis uuritakse läbi eraldi kõik selle salve pesad ja antakse tagasi sealse kõige õrnemat voolu kannatava komponendi vooluandmed.

```
using System;
namespace TakistusKomponendid {
    class JadamisiTakistiteSalv:TakistusKomponent {
        TakistusKomponent[] Andmed;
        int KomponentideArv = 0;
        const double TyhjaSalveMaksimumVool=16;
        public JadamisiTakistiteSalv(int MaxKogus) {
            Andmed = new TakistusKomponent[MaxKogus];
        }
        public void LisaTakistusKomponent(TakistusKomponent t) {
            Andmed[KomponentideArv] = t;
            KomponentideArv++;
        }
        public override double LeiaTakistus() {
            double abi = 0;
            for (int i = 0; i < KomponentideArv; i++) {
                abi = abi + Andmed[i].LeiaTakistus();
            }
            return abi;
        }
        public override double LeiaMaksimumVool() {
            double MaxVool = TyhjaSalveMaksimumVool;
            for (int i = 0; i < KomponentideArv; i++) {
                if (Andmed[i].LeiaMaksimumVool() < MaxVool) {
                    MaxVool = Andmed[i].LeiaMaksimumVool();
                }
            }
            return MaxVool;
        }
```

```
    }
}
```

Edasine on taas katsetus. Luuakse kaks salve. Ühele sisse kaks kümneoomist takistit, kahe- ja ühevatine. Teise salve üks 15-oomine kümnevatine takisti. Ning kolmanda salve sisse sootuks kaks esimest salve. Edasi võib juba rahus päringuid tegema hakata ja vaadata, milliseid andmeid kust tagastatakse.

```
using System;
namespace TakistusKomponendid {
    class TakistusKomponentideProov {
        public static void Main(string[] arg) {
            JadamisiTakistiteSalv ts1 =
                new JadamisiTakistiteSalv(5);
            ts1.LisaTakistusKomponent(new Takisti(10, 2));
            ts1.LisaTakistusKomponent(new Takisti(10, 1));
            JadamisiTakistiteSalv ts2 =
                new JadamisiTakistiteSalv(5);
            ts2.LisaTakistusKomponent(new Takisti(15, 10));
            JadamisiTakistiteSalv ts3 =
                new JadamisiTakistiteSalv(5);
            ts3.LisaTakistusKomponent(ts1);
            ts3.LisaTakistusKomponent(ts2);
            Console.WriteLine("Esimese salve maksimumvool: " +
                  ts1.LeiaMaksimumVool());
            Console.WriteLine("Teise salve maksimumvool: " +
                  ts2.LeiaMaksimumVool());
            Console.WriteLine("Kolmanda salve maksimumvool: " +
                  ts3.LeiaMaksimumVool());
        }
    }
}

/*
Esimese salve maksimumvool: 0,316227766016838
Teise salve maksimumvool: 0,816496580927726
Kolmanda salve maksimumvool: 0,316227766016838
*/
```

Ülesandeid

* Koosta salv nelja viievatise kümneoomise takistiga. Leia kogutakistus, samuti suurim lubatud vool.

* Lisa abstraktsele takistuskomponendile käsklus LeiaPingeVastavaltVoolule. Selle ning LeiaMaksimumVool-u abil koosta käsklus LeiaMaksimumPinge. Katseta seda käsklust nii üksiku takisti kui takistussalve juures. Võrdele leitud pinget eelmises ülesandes leitud voolule vastava pingega.

* Koosta TakistusKomponendi alamklass rööpühenduses oleva kahe takistuskomponendi kogutakistuse leidmiseks. $\frac{1}{R} = \frac{1}{R_1} + \frac{1}{R_2}$. Maksimumvoolu leidmisel tuleb arvestada, et mõlemale

takistile mõjub sama pinge, takisteid läbiv vool aga jaotub pöördvõrdeliselt vastavalt takistusele – mida suurem takistus, seda väiksem vool. Võimalik on ka hakata pingeid katsetama väikese sammuga ja sealtkaudu leida sobiv maksimumvool. Katseta selle rööpühenduse klassi põhjal loodud objekti töö õigsust mitmesuguste takistite ja takistussalvede puhul – kaks ühesugust takistit rööpühenduses, kaks erinevat takistit rööpühenduses. Takisti ja jadamisi salv rööpühenduses. Kaks rööpühendust veel omakorda rööpühenduses.

Operaatorite üledefineerimine

Kui kõik saadud kiidusõnad kokku liita, siis võib end küll inglina tunda ...
Ehk siis liitmine ei pruugi sugugi tähendada arvude aritmeetilist summeerimist, vaid võib vastavalt taustale ja teemale hoopis isesuguse tähenduse saada. Samuti võib omaloodud klassist objektide puhul määrata, mida üks või teine tehtemärk tegelikult nende objektidega teeb. Mõnes keeles (nt. Java) on tehtemärkide programmeerijapoolne määramine veaohu tõttu ära keelatud. Aga siia keelde on võimalus alles jäetud ning aitab loodud objekte tavaelus toimuvaga sarnasemalt käituma panna.

Eelpoolkirjeldatud indekseerimistehte juures määras programmeerija samuti, kuidas omaloodud objekti puhul kantsulgude operaatori juures toimida. Järgnevalt paistab aga, et omapoolse tõlgenduse võib lisada mitmetele kasutatavatele tehtemärkidele. Nii nagu kantsulgude puhul, nii ka igasugu muude tehtemärkide asenduste puhul saab sama töö ära teha tegelikult vaid omaloodud funktsioone kasutades. Ja seetõttu pole koodi kirjutamise juures siinne teema sugugi hädavajalik. Aga võõra koodi mõistmiseks või oma objektidega lihtsaks ja elegantseks toimetamiseks sobivad omapoolselt käituma määratud tehtemärgid hästi.

Näiteks on võetud inimestele hästi tuttav kellaaeg. Tunde ja minuteid saab sarnaselt liita nagu kõiksugu muid suurusi. Kui aga näiteks 14:45le liita kaks tundi ja 30 minutit, siis tulemuseks pole mitte 16:75, vaid 17:15 ja selline ümberarvutus tuleb ka programmile selgeks teha, et välja arvutatud tulemustega ka inimeste keskkonnas midagi peale hakata oleks.

Plussmärgi üledefineerimiseks luuakse funktsioon nimega `operator+`, parameetriteks antakse kaks kellaaega - ehk siis vasakul ning paremal pool tehtemärki olev. Ja funktsiooni tagastustüübiks on samuti `Kellaaeg`. See tähendab, et kui kokku liidetakse kaks `Kellaaega`, siis tulemuseks on ka `Kellaaeg`.

Funktsiooni sisu jääb praegu lühikeseks. Käsu `new` abil luuakse uue `Kellaaja` eksemplar ning mõlema välja jaoks antakse lihtsalt ette plussmärgi mõlemal pool olnud vastava välja summad. Et funktsiooni esimese parameetrina olev `k1` on tehte väljakutsuja, siis siin on võetud tund ja minut ta väljade seest. Objekti `k2` puhul on andmete poole pöördumiseks viisakalt meetodit kasutatud. Aga eks oleks võimalik ka mõlemal juhul meetodiga seda küsimist toimetada.

```
public static Kellaaeg operator+(Kellaaeg k1, Kellaaeg k2){
   return new Kellaaeg(k1.tund+k2.Tund(), k1.minut+k2.Minut());
}
```

Et uue eksemplari loomisel andmed ilusti salvestatud saaksid, selle eest hoolitseb konstruktor. Samuti kutsub viimane välja käskluse `aegKorda`, mille ülesandeks on liigsed minutid tundideks muundada.

```
   public Kellaaeg(int utund, int uminut){
      tund=utund;
      minut=uminut;
      aegKorda();
   }
```

Senikaua, kui minuteid juhtub etteantud olukorras olema üle kuuekümne, minnakse järgmise tunni juurde ning võetakse minutite alt neid tunni jagu vähemaks.

```
   void aegKorda(){
      while(minut>60){
         tund++;
         minut-=60;
      }
   }
```

Kõige tavalisemal juhul, ehk siis, kui minuteid ongi alla kuuekümne, on tsükli tingimus kohe algul väär, tsüklit ei täideta ühtegi korda ning funktsioon lõpetab töö ilma midagi tegemata. Aga vähemasti võime kindlad olla, et ajaga on kõik korras.

```
using System;
class Kellaaeg{
   int tund, minut;
   public Kellaaeg(int utund, int uminut){
      tund=utund;
      minut=uminut;
      aegKorda();
   }
   void aegKorda(){
      while(minut>60){
         tund++;
         minut-=60;
      }
   }
   public int Tund(){return tund;}
   public int Minut(){return minut;}
   public void tryki(){
      Console.WriteLine("{0}:{1}", tund, minut);
   }
   public static Kellaaeg operator+(Kellaaeg k1, Kellaaeg k2){
      return new Kellaaeg(k1.tund+k2.Tund(), k1.minut+k2.Minut());
   }
}
class Test{
   public static void Main(string[] arg){
      Kellaaeg k1=new Kellaaeg(12, 10);
      Kellaaeg k2=new Kellaaeg( 1,  4);
      Kellaaeg k3=k1+k2;
      k3.tryki();
```

```
    }
}
/*
C:\Projects\oma\naited>Operaatorid1
13:14
*/
```

Tüübimuundusoperaatorid

Tahtes reaalarvu täisarvuks muundada nõnda, et komakohad kaduma lähevad, piisab arvu ette sulgudes sõna `(int)` kirjutamisest. Näiteks

```
int a=(int)6.34;
```

Kui uut tüüpi ette ei kirjutaks, siis annaks kompilaator veateate, sest reaalarvu ei pruugi olla võimalik kadudeta muundada täisarvuks. Sama lugu kehtib ka erineva suurusvaruga täisarvude või erineva täpsusega reaalarvude puhul: kui muundusel võib andmeid kaduma minna, siis tuleb muunduskäsk selgelt välja kirjutada. Vastupidi võib lasta ka arvutil tüübimuunduse automaatselt ära teha. Näiteks

```
double b=3;
```

Ehkki kirjutatud kolm on arvuti jaoks algselt tüübist int, lubatakse see rahus reaalarvule omistada.

Tüübimuundusi võib aga ka omaloodud tüüpide juures ette võtta.

Järgnevaga teatatakse, mis tuleb ette võtta juhul, kui kellaaeg omistatakse täisarvule. Sõna `implicit` ütleb, et omistada võib eraldi tüübiteisenduskäsku `(int)` näitamata.

```
   public static implicit operator int(Kellaaeg k){
      return k.Tund()*60+k.Minut();
   }
```

Ehk kui algul kirjutatakse

```
      Kellaaeg k1=new Kellaaeg(12, 10);
```

ja pärast

```
      int minutidPaevaAlgusest=k1;
```

siis kellaaja muutmine arvuks käib ilma, et programmeerija peaks sellele eraldi tähelepanu juhtima.

Kui operaatorite kirjeldajal aga tekib kahtlus, et teisenduste käigus võivad andmed ebatäpsemateks muutuda, või lihtsalt soovitakse, et kogemata ei tehtaks kirjeldatavat teisendust, siis tuleb lisada piiritlejaks sõna `explicit`.

```
   public static explicit operator double(Kellaaeg k){
      //kohustuslik muunduse näitamine
      return k.Tund()+k.Minut()/60.0;
   }
```

Sellisel juhul tuleb omistamisel sobivasse kohta kirjutada `(double)`, et teisendusest asja saaks. Muidu annab kompilaator lihtsalt veateate.

```
      double tunnidPaevaAlgusest=(double)k1;
```

Operaatorid võivad töötada ka teises suunas - ehk siis olemasolevast tüübist uue loodava tüübi poole. Siin näites tehakse minutite hulgast taas `Kellaaeg`. Niipalju, kui jagub täistunde, pannakse tundide alla. Mis aga tundideks jagamisest jäägina üle jääb, see muutub minutiteks.

```
   public static explicit operator Kellaaeg(int minutid){
       return new Kellaaeg(minutid/60, minutid%60);
   }
```

Ning omistamisel saabki minutid taas `Kellaajaks`.

```
      Kellaaeg k4=(Kellaaeg)minutidPaevaAlgusest;
```

Kui koodi töö tulemust vaadata, siis algul oli kellaaeg `k1` 12:10. Vahepeal muundati see muutujasse `minutidPaevaAlgusest` ja saadi täisarvuna 730. Lõpuks minutid taas tundide ja minutitena kellaaja sisse jaotatuna andsid jälle 12 tundi ja 10 minutit.

```
using System;
class Kellaaeg{
   int tund, minut;
   public Kellaaeg(int utund, int uminut){
      tund=utund;
      minut=uminut;
      aegKorda();
   }
   void aegKorda(){
      while(minut>60){
         tund++;
         minut-=60;
      }
   }
   public int Tund(){return tund;}
   public int Minut(){return minut;}
   public void tryki(){
```

```
      Console.WriteLine("{0}:{1}", tund, minut);
   }
   public static  Kellaaeg operator+(Kellaaeg k1, Kellaaeg k2){
      return new Kellaaeg(k1.Tund()+k2.Tund(), k1.Minut()+k2.Minut());
   }
   public static implicit operator int(Kellaaeg k){
      return k.Tund()*60+k.Minut();
   }
   public static explicit operator double(Kellaaeg k){
      //kohustuslik muunduse näitamine
      return k.Tund()+k.Minut()/60.0;
   }
   public static explicit operator Kellaaeg(int minutid){
       return new Kellaaeg(minutid/60, minutid%60);
   }
}
class Test{
   public static void Main(string[] arg){
      Kellaaeg k1=new Kellaaeg(12, 10);
      Kellaaeg k2=new Kellaaeg( 1,  4);
      Kellaaeg k3=k1+k2;
      k3.tryki();
      int minutidPaevaAlgusest=k1;
      double tunnidPaevaAlgusest=(double)k1;
      Console.WriteLine(minutidPaevaAlgusest);
      Console.WriteLine(tunnidPaevaAlgusest);
      Kellaaeg k4=(Kellaaeg)minutidPaevaAlgusest;
      k4.tryki();
   }
}

/*
C:\Projects\oma\naited>Operaatorid2
13:8
730
12,1666666666667
12:10
*/
```

Võrdlusoperaatorid

Ikka tahetakse võrrelda, kas midagi toimus enne või pärast; uus on vanast suurem või väiksem; keegi kellestki targem või rumalam. Mõningate objektide puhul ei pruugi selline võrdlus anda selgepiirilist jah/ei vastust ning sellisel juhul on mõistlik arvuti jaoks võrdlusoperaatori defineerimine ära jätta ning piirduda pigem mõnevõrra hägusema ja paindlikuma skaalaga. Kui aga enamikel juhtudel on objektid omavahel kindlalt võrreldavad, siis on vastav operaator omal kohal.

Võrdluse jaoks on tehteid palju: >, <, <=, <=, ==, !=. Enamasti pole aga otstarbekas kõiki neid kohe ja eraldi defineerima hakata. Pigem teha täisarvu väljastav levinud funktsioon `Compare`. Selles võrreldakse aktiivset objekti funktsiooni parameetrina antud objektiga. Kui programmeerija arvates on aktiivne objekt parameetrina antud objektist järjestusreas eespool, siis peaks funktsioon väljastama negatiivse arvu. Kui tagapool, siis positiivse. Ning kui objektide võrreldavad tunnused peaksid programmeerija arvates olema võrdsed, siis tuleb väljastada 0. Edasised võrdlustehted saab juba sedasama funktsiooni kasutades korda ajada.

Siin näites tehakse mõlema kellaaja sisu kõigepealt minutiteks alates päeva algusest, et oleks kergem võrrelda ning siis juba saab ühe lahutustehtega sobiva vastuse kätte.

```
public int Compare(Kellaaeg k){
   int omaminutid=(int)this;
   int teiseminutid=(int)k;
   return omaminutid-teiseminutid;
}
```

Samuti on viisakas üle katta klassist `Object` kaasa tulev käsklus `Equals`, mille ülesandeks on teatada, kas aktiivne objekt on etteantud objektiga sisu poolest võrdne. Kui oma `Compare` juba defineeritud, siis läheb edasine juba küllalt sarnaselt iga objekti puhul. Tingimuse esimese poolega tasub kontrollida, et kas võrreldav objekt üldse on meie omaga sama tüüpi. Ehk siis küsitakse, kas

```
ob is Kellaaeg
```

Vastus on tõene vaid juhul, kui etteantud objekt tõesti oli `Kellaaeg`. Ainult sel juhul minnakse `&&` abil kirjutatud võrdluse kontrollimisega edasi. Muul juhul on tingimus kohe vale, tingimusblokist hüpatakse üle ning väljastatakse funktsiooni lõpus tagastatav vastus teatamaks, et kindlasti ei ole meie `Kellaaeg` sama väärtusega kui võrdlemiseks etteantud objekt, sest see teine objekt lihtsalt ei ole `Kellaaeg`.

Et saadud objekti saaks `Kellaajana Compare`-meetodisse panna, peab eraldi teatama, et me teda ikka `Kellaajana` kasutame. Ehkki `is`-kontrolli abil tegime juba kindlaks, et tegemist on `Kellaajaga`, tuleb funktsioonile ette andmiseks see uuesti muundada. Üheks võimaluseks oleks

```
(Kellaaeg)ob
```

siin aga näitame teist sarnast võimalust

```
ob as Kellaaeg
```

mis käitub üldjoontes samamoodi. Ainsaks erinevuseks on, et kui peaks siiski tüübiprobleem tekkima, siis esimesel juhul heidetakse erind, teisel juhul väljastatakse aga lihtsalt tühiväärtus. Kuna siin on juba kontroll tehtud, siis veateadet nagunii ei saa tekkida ning kood töötaks mõlemal juhul sarnaselt. Käsule `Compare` antakse tulemus kontrollida. Kui väärtused loeti võrdseteks ning väljastatakse 0, siis sel juhul kannatab `Equals` välja anda `true`, muul juhul `false`.

```
public override bool Equals(Object ob){
   if (ob is Kellaaeg && (this.Compare(ob as Kellaaeg)==0)){
     return true;
   }
   return false;
}
```

Oma võrdlusoperaatorite puhul on viisakas üle katta veel ka käsklus `GetHashCode`. Selle salapärase käsu ülesandeks on anda arvutile märku, kas objektid võiksid olla võrdse väärtusega. Selleks tuleb objekti andmete põhjal kokku panna üks täisarv. Kui kahe objekti andmed kattuvad, peab see arv olema ühesugune, muul juhul võimaluse korral erinev. Kasutatakse näiteks paisktabelites, andmete otsimise jms. korral. Et meil on juba olemas andmete minutiteks tegemise operaator, siis see sobib räsikoodiks imehästi. Piisab vaid andmed täisarvuks teha, kui juba ongi eri kellaaegade puhul erinev arv käes, sest sama arv minuteid päeva algusest saab olla vaid ühesuguse kellaaja korral.

```
   public override int GetHashCode(){
      return (int)this;
   }
```

Kui näiteks hoitaks ajahetke objektis nii kellaaegu kui kuupäevi, siis võiks olukord veidi raskemaks minna, sest neljabaidine int ei pruugi suuta näidata kõiki erinevaid väärtusi, mis aastatuhandete jooksul ette tulevad. Sel juhul tuleks juba keerukamaid arvutusi rakendada, et kõik objekti väljad oleksid räsikoodi arvutamisel rakendatud ja muudaksid tulemust. Samas aga oleks eri väärtusega objektidel kahe sarnase räsikoodi kokku juhtumine võimalikult haruldane. Ehk siis ligikaudu üks paari miljardi kohta - nii nagu neid int-muutuja erinevaid väärtusi on.

Kui funktsioonidega eeltöö tehtud, siis edasine operaatorite defineerimine on juba käkitegu: tuleb lihtsalt õige funktsioon välja kutsuda.

```
   public static bool operator!=(Kellaaeg k1, Kellaaeg k2){
     return !k1.Equals(k2);
   }
```

Ning teiste võrdlustega sarnaselt. Nüüd aga kood tervikuna.

```
using System;
class Kellaaeg{
   int tund, minut;
   public Kellaaeg(int utund, int uminut){
      tund=utund;
      minut=uminut;
      aegKorda();
   }
   void aegKorda(){
      while(minut>60){
         tund++;
         minut-=60;
      }
   }
   public int Tund(){return tund;}
   public int Minut(){return minut;}
   public void tryki(){
      Console.WriteLine("{0}:{1}", tund, minut);
   }
   public static  Kellaaeg operator+(Kellaaeg k1, Kellaaeg k2){
      return new Kellaaeg(k1.Tund()+k2.Tund(), k1.Minut()+k2.Minut());
   }
```

```
   public static implicit operator int(Kellaaeg k){
      return k.Tund()*60+k.Minut();
   }
   public static explicit operator double(Kellaaeg k){
      //kohustuslik muunduse näitamine
      return k.Tund()+k.Minut()/60.0;
   }
   public static explicit operator Kellaaeg(int minutid){
       return new Kellaaeg(minutid/60, minutid%60);
   }
   public int Compare(Kellaaeg k){
            //Võrdlusoperaatorite puhul soovitav defineerida
      int omaminutid=(int)this;
      int teiseminutid=(int)k;
      return omaminutid-teiseminutid;
   }
   public override bool Equals(Object ob){
      if (ob is Kellaaeg && (this.Compare(ob as Kellaaeg)==0)){
        return true;
      }
      return false;
   }
   public override int GetHashCode(){
      return (int)this;
   }
   public static bool operator==(Kellaaeg k1, Kellaaeg k2){
     return k1.Equals(k2);
   }
   public static bool operator!=(Kellaaeg k1, Kellaaeg k2){
     return !k1.Equals(k2);
   }
   public static bool operator<(Kellaaeg k1, Kellaaeg k2){
     return k1.Compare(k2)<0;
   }
   public static bool operator>(Kellaaeg k1, Kellaaeg k2){
     return k1.Compare(k2)>0;
   }
}

class Test{
   public static void Main(string[] arg){
      Kellaaeg k1=new Kellaaeg(12, 10);
      Kellaaeg k2=new Kellaaeg( 1,  4);
      Kellaaeg k3=new Kellaaeg( 1,  4);
      if(k2==k3){
         Console.WriteLine("Samad");
      }
      if(k2<k1){
         Console.WriteLine("Enne");
      }
   }
}

/*
C:\Projects\oma\naited>Operaatorid3
Samad
Enne
*/
```

Ülesandeid

* Alusta esimesest lühikesest näitest ning lisa kellaajale ka sekundid.

* Arvesta sekundeid ka kellaaegade liitmise juures.

* Juhul, kui Kellaaeg teisendatakse tüübiks int, anna väärtus endise minutite asemel sekundites.

* Loo klass Nihe, mille väljadeks on pikkus x- ja y-suunal.

* Defineeri operaator nihete liitmiseks.

* Võimalda nihkeid ka lahutada.

* Võrdlusoperaator loeb nihked võrdseks vaid juhul, kui mõlema koordinaattelje suunalised pikkused on võrdsed.

* Suurema ja väiksema võrdlemisel võrreldakse vaid nihete pikkusi.

* Hoolitse ka omaloodud räsikoodi eest.

Abivahendid

Erindid

Kui arvutil palutakse teha tema jaoks võimatu käsk, siis enamasti lõpeb programmi töö veateatega. Nii nagu järgnevas näites, kus tekst "Tere" püütakse muundada täisarvuks.

```
using System;
class Erind1{
   public static void Main(string[] arg){
      string tekst1="Tere";
      int arv1=int.Parse(tekst1);
      Console.WriteLine(arv1);
   }
}
```

Veateatest võib välja lugeda, et klassi `Erind1` käsklusest `Main` kutsuti välja `System.Number.ParseInt32`, mis omakorda kutsus välja käsu `StringToNumber`. Viimane aga jäi teksti arvuks muutmisega hätta. Anti välja veateade tüübist `System.FormatException` koos selgitusega, et etteantud sõna pole sobival kujul.

```
/*
C:\Projects\oma\naited>Erind1
Unhandled Exception: System.FormatException: Input string was not in a
correct format.
   at System.Number.StringToNumber(String str, NumberStyles options,
   NumberBuffer& number, NumberFormatInfo info, Boolean parseDecimal)
   at System.Number.ParseInt32(String s, NumberStyles style,
NumberFormatInfo info)
   at Erind1.Main(String[] arg)
*/
```

Vähemasti saime teada, mis juhtus, aga kasutaja ei pruugi sellise teatega kuigivõrd rahul olla. Eriti kui tuleb ette süsteemne veateateaken, mis püüab andmeid kuhugi saata või programmi siluma hakata ning keeldub eest ära minemast.

Püüdmine

Arvuti veateate saab asendada enese omaga. Või siis hoopis paluda veateate andmise asemel arv uuesti sisestada või sisestusest loobuda. Siin antakse lihtsalt omapoolne väike seletus toimunu kohta.

Veateate püüdmiseks on C# keeles olemas `try{}catch` plokk. Kõik looksulgude vahel juhtunud probleemid saadetakse lahendamisele `catch`-ossa, kus on juba programmeerija otsustada, mida tekkinud olukorras peale hakata.

```
using System;
class Erind2{
   public static void Main(string[] arg){
     try{
      string tekst1="Tere";
      int arv1=int.Parse(tekst1);
      Console.WriteLine(arv1);
     }catch(FormatException probleem){
        Console.WriteLine("Viga teisendusel: "+probleem.Message);
     }
   }
}
/*
C:\Projects\oma\naited>Erind2
Viga teisendusel: Input string was not in a correct format.
*/
```

Reageering tüübi põhjal

Sama koodilõigu juures võib ette tulla mitmesuguseid probleeme. Kord ei leita sobivat andmefaili, teinekord ei saa teksti arvuks muundada ning mõnikord võib hoopis ette tulla jagamine nulliga. Vanemate programmeerimiskeelte juures oli tavaks iga käsu juures kontrollida, kas see õnnestus, ning siis püüda koheselt reageerida. Kui kohene parandamine on võimalik, on selline lähenemine hea. Kui aga peab parandamiseks palju asju ära muutma, siis kulub palju tööd. Selle lihtsustamiseks erindid ja veahaldus välja mõeldigi.

Ploki lõpus oleva `catch`i sulgudesse kirjutatakse selline erinditüüp, millele ollakse valmis reageerima. Nagu eespool oli - `FormatException` tekkis sisendandmete vormingu vea tõttu ning sellele probleemile ka reageeriti. Võib tekkida aga olukord, kus sisendiks on küll kõik numbrid, aga kokku tuleb int-vormingu jaoks liiga suur arv. Sellisel juhul heidetakse hoopis `OwerflowException`. Eraldi `catch`idega püüdes saab nendele vigadele sobivalt reageerida.

Vigade klassid moodustavad isekeskis hierarhia. Selle puu juureks on klass nimega `Exception`. Tema alamklassideks on hulk .NET runtime käivitamisega seotud probleemiklasse, aga muuhulgas ka `SystemException`, mille alt siis omakorda kättesaadavad enamik meil programmides ettetulevaid System-nimeruumi objektidega seotud erindeid.

`SystemException`i enese alt leiab omakorda näiteks `ArithmeticException`i, mille juurest omakorda `OverflowException`i ja `DivideByZeroException`i. Kui me tahame liialt suurt arvu (ületäitumine) ning nulliga jagamist kontrollida sama `catch`i sees, siis võib piirduda `ArithmeticException`i püüdmisega. Kui aga kummagi olukorra jaoks on soov käivitada eri kood, siis tasub need eraldi kinni püüda.

Viga jääb kinni ainult ühes `catch`-plokis. Seepärast pannakse detailsemad erindid püüdmisel ettepoole ning üldisemad tahapoole. Muidu juhtuks, et teade jääb üldisematele tingimustele vastavasse plokki kinni ja väljavalitud lõiku kunagi ei pruugitagi. Tahtes kõik teated kindlasti kätte saada, võib lõppu panna `catch(Exception)`. Sellele tüübile vastavad kõik veateated - ka need, mis on kõigist muudest püünistest juba mööda tulnud.

Juhul, kui soovitakse saabunud veateate andmetega midagi lähemat ette võtta, saab selle püüda omaette muutujasse ning sealtkaudu erindiobjektiga suhelda. Nagu näiteks

```
catch(FormatException probleem){
   Console.WriteLine("Viga sisendandmetega: "+probleem.Message);
}
```

Kui aga piisab vaid teadmisest, et juhtus vastavat tüüpi olukord ning sellest teadmisest on meile reageerimiseks küllalt, siis võib oma muutuja loomata jätta nagu näiteks

```
catch(OverflowException){
   Console.WriteLine("Liiga suur arv.");
}
```

`finally`-plokki püüniste lõpus kasutatakse käskluste jaoks, mis tulevad igal juhul ära teha. Näiteks faili sulgemine andmete lugemisel: isegi siis, kui lugemine ebaõnnestus, tuleb fail teistele kasutajatele kättesaadavaks teha. Aga mainimist väärib, et `finally`-plokki jõutakse siiski ainult juhul, kui viga polnud või sai sellele reageeritud. Nii et kindlaks lõpuni jõudmiseks on mõistlik panna viimaseks veapüüniseks ikkagi `catch(Exception)`. Kuigi - vahel soovitatakse, et pigem jäta erind püüdmata, kui et püüad midagi, millega sa mõistlikku peale hakata ei oska. Et kui tuleb ametlik veateade, on see vahel parem, kui omalt poolt vea peitmine, mis võib hiljem vigaste andmete näol kusagil kätte maksta.

Nüüd aga näide tervikuna. Käsurea parameetrina oodatakse numbreid, mille programm arvuks teisendab ning välja trükib. Juhtub aga midagi sobimatut, siis teatatakse vastav veateade.

```
using System;
class Erind3{
   public static void Main(string[] arg){
     try{
        if(arg.Length!=1){
          Console.WriteLine("Kasuta kujul: Erind3 sisendarv");
          return;
        }
        string tekst1=arg[0];
        int arv1=int.Parse(tekst1);
        Console.WriteLine("Sisestati edukalt "+arv1);
     } catch(FormatException probleem){
        Console.WriteLine("Viga sisendandmetega: "+probleem.Message);
     } catch(OverflowException){
        Console.WriteLine("Liiga suur arv.");
     } catch(Exception){
        Console.WriteLine("Tundmatu probleem");
     } finally{
        Console.WriteLine("Plokk otsas");
     }
   }
}
/*
C:\Projects\oma\naited>Erind3
Kasuta kujul: Erind3 sisendarv
Plokk otsas
C:\Projects\oma\naited>Erind3 tere
Viga sisendandmetega: Input string was not in a correct format.
Plokk otsas
C:\Projects\oma\naited>Erind3 1234567890123456
```

```
Liiga suur arv.
Plokk otsas
C:\Projects\oma\naited>Erind3 78
Sisestati edukalt 78
Plokk otsas
*/
```

Püüdmine alamprogrammist

Veapüüniste tähtsaim eelis varasema veakoodinduse ees ongi kogu rakenduse alamprogrammide rägastikus tekkinud probleemide transport konkreetsetesse kohtadesse kokku, kus nendega üheskoos on vahel mõnevõrra kergem hakkama saada.

Järgnevas näites tekibki tõenäoline probleem alamprogrammis nimega `LoeArv` juhul, kui sisendiks pole arv. Veateade aga trükitakse alles `Main`-meetodi juures. Nõnda võib näiteks paluda kasutajal arvutamise jaoks anda mitu arvu. Kui aga kasvõi ühel korral sisestusel eksiti, on tulemus ikka sama - tulemust pole võimalik kokku saada. Ning sellest antakse veapüünises ka teada.

```
using System;
class Erind4{
   public static int LoeArv(){
      Console.WriteLine("Palun arv:");
      string s=Console.ReadLine();
      int a=int.Parse(s);
      return a;
   }
   public static void Main(string[] arg){
     try{
      int arv1=LoeArv();
      Console.WriteLine("Kirjutati: "+arv1);
     }catch(FormatException probleem){
        Console.WriteLine("Viga teisendusel: "+probleem.Message);
     }
   }
}
/*
D:\kodu\0606\opikc#>Erind4
Palun arv:
5
Kirjutati: 5
*/
```

Erindi heitmine

Sugugi ei pea leppima vaid arvuti enese antud veateadetega. Kui ikka oma programmis paistab, et midagi läheb väga käest ära, siis on vahel kasulik ise märku anda, et sarnaselt edasi toimida pole enam mõtet. Näiteks, kui arvutuse algandmed on ilmselgelt valed (kolmnurga üks külg pikem kui teised kaks kokku), siis võib julgesti enne arvutamist teada anda, milles asi ning heita selleteemalise erindi. Edasi on juba vastavat koodilõiku väljakutsuva programmeerija ülesandeks silumise käigus kindlaks teha, millest probleem tekkis ning vastavalt edasi toimida.

Siin näites lihtsalt keelati sajast suuremate arvude sisestus. Kui arv juhtub liiga suur olema, heidetakse erind. Lihtsuse mõttes pole oma tüüpi loodud, kasutatakse `SystemException`it. Kuigi -

vähegi pikema programmi selguse huvides oleks oma tüübi loomine kasulik. Et peaprogrammis pole `SystemException`i jaoks veapüünist, siis tuleb ette süsteemne veateade, mille järele programmeerija peab juba ise edasi mõtlema, mida edasi teha.

```
using System;
class Erind5{
   public static int LoeArv(){
      Console.WriteLine("Palun arv:");
      string s=Console.ReadLine();
      int a=int.Parse(s);
      if(a>100){
        throw new SystemException("Liiga suur arv");
      }
      return a;
   }
   public static void Main(string[] arg){
     try{
      int arv1=LoeArv();
      Console.WriteLine("Kirjutati: "+arv1);
     }catch(FormatException probleem){
        Console.WriteLine("Viga teisendusel: "+probleem.Message);
     }
   }
}
/*
D:\kodu\0606\opikc#>Erind5
Palun arv:
789
Unhandled Exception: System.SystemException: Liiga suur arv
   at Erind5.LoeArv()
   at Erind5.Main(String[] arg)
*/
```

Ülesandeid

* Katseta näite "Erind2" juures, kuidas käitub programm juhul, kui ette anda veatu arv.

* Muuda täisarvu käsklused reaalarvu omadeks ja leia, mis kasutamisel muutus.

* Loo tsükkel, mille abil küsitakse arvu senikaua, kuni saadakse sobiv sisend.

* Muuda näidet "Erind5" nõnda, et see annaks peaprogrammis viisaka seletuse ka omaheidetud erindi korral.

* Loo erindeid kasutades programm, mis suurendaks faili arv.txt sisu ühe võrra. Kui fail puudub, või failis pole arv, siis antakse selgitusega veateade.

* Kui failis olev arv ületab 365, siis anna välja omapoolne erind ning püüa sellele reageerida.

Enum

Ikka leidub kohti, kus on võimalik teha piiratud arv valikuid. Asukoht Eestis on ühes maakondadest. Ühissõiduk on üldjuhul rong, tramm, troll või buss jne. Kui programmikoodis tuleb leida käitumisjuhis ühele etteantud valikutest, siis on enum hea abivahend. Maakonna nime saab kirjutada mitmeti.

Olgu siis "Harjumaa", või "Harju maakond", rääkimata suurte ja väikeste tähtede ning tühikute erisusest. Andmebaaside puhul kasutatakse üldjuhul võimalust, et ei kirjutata inimese andmete juurde maakonna nime, vaid pannakse selle maakonna kood. Ning selle järgi on vajadusel võimalik teisest andmetabelist järele vaadata, millise maakonnaga siis päriselt tegu. Midagi sarnast toimub ka enumi puhul. Ehk siis vastavas loetelus kirjeldatakse ära kõik võimalikud väärtused. Ning hiljem programmi sees pole võimalik enam vastavat nimetust valesti kirjutada ilma, et kood kompileerimata jääks. Sedasi on võimalik vältida vigu, mis muidu üllatavatel hetkedel võiksid avalduda. Tüüpiliseks kasutuskohaks on näiteks alamprogrammi parameetrid, kus enumi abil määratakse, kuidas just sel korral vastavate andmetega tuleb käituda. Järgnevas näites siis tuuakse tugevuse kohta kolm konstanti: tumm, ühekordne ja mitmekordne. Ning praegusel juhul alamprogrammis esimese variandi puhul jäetakse etteantud tekst sootuks trükkimata. Teisel juhul trükitakse ühe korra ning viimasel juhul mitu korda. Kui aga ühekordne kirjutatuks nõrga g-ga, siis jäänuks kood kompileerimata. Pealtnäha iseenesestmõistetav. Aga kui enumi asemel olnuks kasutatud stringi, siis just sellised vead on kerged tulema.

```
using System;
namespace Enumeratsioon1{
   enum tugevus{tumm, yhekordne, mitmekordne};
   class Trykkimine{
      static void Tryki(string tekst, tugevus t){
            if(t==tugevus.yhekordne){
               Console.WriteLine(tekst);
            }
            if(t==tugevus.mitmekordne){
               Console.WriteLine(tekst);
               Console.WriteLine(tekst);
            }
      }
      public static void Main(string[] arg){
         Tryki("Tere", tugevus.yhekordne);
      }
   }
}
/*
E:\jaagup\07\12>Enumeratsioon1
Tere
*/
```

Ülesandeid

* Loo klass vooluallika andmete hoidmiseks (pingevahemik, enum näitamaks kas tegemist alalis- või vahelduvvooluga)

* Koosta sellistest vooluallikatest mitmesuguste väärtustega massiiv.

* Koosta alamprogramm, mis saab parameetriks soovitud pinge, voolutüübi ja vooluallikate massiivi ning trükib välja soovitule vastavate vooluallikate andmed.

Andmekollektsioonid

Andmetega ümberkäimisele kulub märgatav osa arvutite ja programmeerija ajast. 2000 aasta paiku arvati selleks osaks olema ligikaudu kolmandik. Nüüd ehk veidi vähem, kuid tähtsus on ikka alles jäänud. Et põhioperatsioonidele ei kuluks liialt palju tähelepanu, selleks on programmeerimiskeeltes välja mõeldud valmis vahendid andmeoperatsioonideks. Nii ka C# puhul.

ArrayList

Hea lihtne koht andmete hoidmiseks ja kätte saamiseks. Võrreldes tavalise massiiviga pole vaja elementide arvu kohe ette määrata. `ArrayListi` objekt hoolitseb ise selle eest, et oleks parajalt ruumi sissepandud andmete hoidmiseks. Iga `Add`-käsklusega lisatakse sissepandud väärtus olemasolevate lõppu. Käsuga `Contains` võib kontrollida otsitava elemendi olemasolu. `Count` näitab elementide arvu. `Insert`-käsklus lisab uue elemendi soovitud järjekorranumbriga kohale, lükates ülejäänud ühe koha võrra edasi. `IndexOf` aitab soovitud väärtust otsida. Viimase puudumisel tagastatakse järjekorranumbrina -1. Ning `foreach`-tsükkel sobib kõigi elementide läbi käimiseks.

```
using System;
using System.Collections;
class Kollektsioon1{
   public static void Main(string[] arg){
      ArrayList nimed=new ArrayList();
      nimed.Add("Kati");
      nimed.Add("Mati");
      nimed.Add("Juku");
      if(nimed.Contains("Mati")){
         Console.WriteLine("Mati olemas");
      }
      Console.WriteLine("Nimesid kokku "+nimed.Count);
      nimed.Insert(1, "Sass");
      Console.WriteLine("Mati asub kohal "+nimed.IndexOf("Mati"));
      Console.WriteLine("Mari asub kohal "+nimed.IndexOf("Mari"));
      foreach(string eesnimi in nimed){
         Console.WriteLine(eesnimi);
      }
   }
}

/*
D:\kodu\0606\dotnet>Kollektsioon1
Mati olemas
Nimesid kokku 3
Mati asub kohal 2
Mari asub kohal -1
Kati
Sass
Mati
Juku
*/
```

Sortimine

Andmete järjestamiseks on välja mõeldud hulk algoritme, millel enamikul mõni eriline koht, kus ta teistest kiiremini töötab. Kui aga meid rahuldab korralik "Harju keskmine" tulemus, siis võib kasutada `ArrayList`ile sisseehitatud käsku `Sort`, mis elemendid kasvavasse järjekorda sätib.

Tahtes andmeid trükkides teada, mitmenda elemendi juures ollakse, tuleb ükshaaval neid järjekorranumbri abil küsida. `ArrayList`i elemendi poole saab pöörduda sarnaselt nagu massiivigi elemendi poole kantsulgude abil.

```
using System;
using System.Collections;
class Kollektsioon1a{
   public static void Main(string[] arg){
      ArrayList nimed=new ArrayList();
      nimed.Add("Kati");
      nimed.Add("Mati");
      nimed.Add("Juku");
      nimed.Sort();
      for(int i=0; i<nimed.Count; i++){
          Console.WriteLine(nimed[i]);
      }
   }
}

/*
D:\kodu\0606\opikc#>Kollektsioon1a
Juku
Kati
Mati
*/
```

Tüübimäärang

Eelkirjeldatud `ArrayList` on lahke - lubab enesesse panna ja sealt võtta igasugu andmetüüpe. Mõnikord on see mugav, kuid vähegi pikemate programmide juures võivad kogemata nimistud sassi minna ja näiteks sünniaasta andmed sattuda näiteks hoopis perekonnanime kohale. Et programmeerimiskeeltes püütakse vea võimalusi vältida, siis on alates .NET 2.0st lisatud uus nimeruum System.Collections.Generic, kus kasutatavate andmestruktuuride juures tuleb kohe algul ära määrata, millist tüüpi andmeid kollektsiooni panna tohib. Ehk siis tekstiliste andmete hoidmiseks sobib

```
      LinkedList<string> nimed=new LinkedList<string>();
```

Kui tegemist oleks arvudega, siis peaks <> märkide vahel olema sõna `int`, mõne muu andmetüübi puhul selle nimi. Lisamiseks ja küsimiseks mõnevõrra teistsugused käsud, aga kõik vajaliku saab tehtud. Kuna `ArrayList`i puhul hoitakse andmeid mälus massiivina, siis on arvuti jaoks lihtne ülesanne anda vastavalt järjekorranumbrile element. Samas aga jada algusesse lisamine võib suurema andmehulga puhul ootamatult palju ressursse nõuda. `LinkedList`iga on vastupidi:

konkreetse elemendi poole pöördumine võib raske olla. Mööda ahelat edasi-tagasi liikumine ning elementide lisamine või eemaldamine käib kiiresti ka pika ahela juures.

```
using System;
using System.Collections.Generic;
class Kollektsioon2{
   public static void Main(string[] arg){
      LinkedList<string> nimed=new LinkedList<string>();
      //lubab ainult stringe
      nimed.AddLast("Kati");
      nimed.AddLast("Mati");
      nimed.AddLast("Juku");
      if(nimed.Contains("Mati")){
         Console.WriteLine("Mati olemas");
      }
      Console.WriteLine("Nimesid kokku "+nimed.Count);
      nimed.AddAfter(nimed.Find("Kati"), "Sass");
      LinkedList<string>.Enumerator enumr=nimed.GetEnumerator();
      while(enumr.MoveNext()){
         string eesnimi=enumr.Current;
         Console.WriteLine(eesnimi);
      }
   }
}

/*
D:\kodu\0606\dotnet>Kollektsioon2
Mati olemas
Nimesid kokku 3
Kati
Sass
Mati
Juku
*/
```

Järjekord

Näiteks graafikaülesannete juures on vajalik andmeid panna järjekorda ootele ning neid siis sealt sissepaneku järjekorras välja küsida. Iseenesest on sarnane toiming ka `LinkedList`i abil tehtav, aga juba .NET versioonis 1.0 oli selle tarvis omaette klass loodud, nimeks `Queue`. Kasutamine lihtne: käsuga `Enqueue` lisatakse andmeid ning `Dequeue` võetakse neid teisest otsast ära.

```
using System;
using System.Collections;
class Kollektsioon3{
   public static void Main(string[] arg){
      Queue jarjekord=new Queue();
      jarjekord.Enqueue("Juku");
      jarjekord.Enqueue("Kati");
      jarjekord.Enqueue("Mati");
      while(jarjekord.Count>0){
         string eesnimi=jarjekord.Dequeue() as string;
         Console.WriteLine(eesnimi);
      }
   }
}
```

```
/*
D:\kodu\0606\dotnet>Kollektsioon3
Juku
Kati
Mati
*/
```

Paisktabel

Vahend andmepaaride hoidmiseks. Kord indekseerimise juures juba tutvusime selle vahendiga, siin nüüd vaatame talle veel korra otsa. Paisktabelis sobib hoida näiteks konfiguratsioonifailist loetud omaduste väärtusi, kasutajanimele vastavaid seadeid või tõlkefaili andmeid. Põhiliseks tingimuseks on, et võti (kasutajanimi või omaduse nimi) ei kordu ning võtme järgi saab küsida väärtuse. Siin näites hoitakse inimeste nimedele vastavaid hindeid.

```
      if(ht.ContainsKey("Kati")){
          Console.WriteLine("{0}", ht["Kati"]);
      }
```

Kontrollitakse, kas Kati on nimede hulgas olemas. Kui jah, siis trükitakse ta hinne.

```
      ht["Sass"]=((int)ht["Sass"])-1;
```

Sassi hinnet alandatakse ühe võrra.

```
      ht.Remove("Mati");
```

Mati eemaldatakse nimekirjast.

Tahtes kõik andmed kätte saada, aitab jälle enumeraator, ainult et igal enumeraatori elemendil on võti ja väärtus. Siin trükitakse nad lihtsalt välja, aga eks igaüks tea ise paremini, mida tal oma programmis nendega kõige mõistlikum teha on.

```
      IDictionaryEnumerator enumr=ht.GetEnumerator();
      while(enumr.MoveNext()){
         string eesnimi=enumr.Key as string;
         int hinne=(int)enumr.Value;
         Console.WriteLine("{0}: {1}", eesnimi, hinne);
      }
```

Ning kogu näide tervikuna.

```
using System;
using System.Collections;
class Kollektsioon4{
```

```
    public static void Main(string[] arg){
      Hashtable ht=new Hashtable();
      ht.Add("Juku", 3);
      ht.Add("Kati", 5);
      ht.Add("Mati", 4);
      ht.Add("Sass", 4);
      if(ht.ContainsKey("Kati")){
          Console.WriteLine("{0}", ht["Kati"]);
      }
      ht["Sass"]=((int)ht["Sass"])-1;
      ht.Remove("Mati");
      IDictionaryEnumerator enumr=ht.GetEnumerator();
      while(enumr.MoveNext()){
         string eesnimi=enumr.Key as string;
         int hinne=(int)enumr.Value;
         Console.WriteLine("{0}: {1}", eesnimi, hinne);
      }
   }
}

/*
D:\kodu\0606\dotnet>Kollektsioon4
5
Kati: 5
Juku: 3
Sass: 3
*/
```

Ülesandeid

* Küsi kasutajalt arve, kuni ta sisestab nulli. Salvesta ArrayListi. Väljasta need arvud tagurpidises järjekorras.

* Proovi eelmine ülesanne lahendada LinkedListi abil. Omadus Last annab loetelu viimase elemendi, RemoveLast() kustutab viimase.

* Loe tekstifailist arvud, väljasta nad sorteerituna teise tekstifaili.

* Loe tekstifailist arvud. Teise tekstifaili väljasta, mitu korda iga arv esines.

Mallid

Objektorienteeritus võimaldab alamklasside eksemplare omistada ülemklassi tüüpi muutujatele. Nõnda saab lahendada enamiku olukordi, kus koodilt nõutakse paindlikkust ning võimet veidi erinevaid objekte ühiselt hoida või käidelda. Kus pole võimalik objekte omistada muidu pärimispuu järgi, seal tuleb appi teadmine, et kõik pärineb ühisest ülemklassist System.Object. Või siis saab eri pärimispuudest tulnud klasside ühiseid käsklusi kasutada liideste abil. Nii et kõik vajalik peaks sellega olemas olema.

Ometigi on C# juurde kaasa võetud C++ist mallid ehk šabloonid ehk geneerilisus. Ehk siis võimalus kasutatavaid andmetüüpe määrata pärast kasutatava klassi koodi enese valmiskirjutamist. Sellega kaasneb vähemasti kaks head omadust:

* Kui andmetüüp on täpselt määratud, siis on karta vähem valest omistamisest tingitud vigu.

* Kompilaatoril on võimalik koodi optimeerida konkreetse andmetüübi omadustest lähtudes ning programmi töö käigus ei pea kulutama aega tegeliku andmetüübi kontrollimisele.

Võimalust kasutatakse tihti geneeriliste andmekollektsioonide juures. Kui muidu oli hoiustatud andmete kohta teada ainult, et need on klassi Object järglased (ehk siis nagu polnudki tüübi kohta suurt midagi teada), siis geneerilise Listi puhul saab määrata näiteks, et loetelus esinevad elemendid on täisarvud. Ning selle põhjal on edaspidises kasutuses teada, et vastavast loetelust välja võetavad andmed on ka sama tüüpi. Nagu järgnevast näitest näha, siis ka loetelu läbimiseks mõeldud enumeraatorile tuleb sama tüüp määrata.

Andmestruktuuri ülesehituse eripärast lähtudes on LinkedListi läbikäimine enumeraatori abil tunduvalt ökonoomsem kui järjestikuste andmete küsimine elemendi järjekorranumbri järgi. Konkreetse järjekorranumbriga kohani jõudmiseks tuleb järjekorranumbri puhul enne kõik elemendid listisiseselt läbi jalutada. Enumeraator aga mõistab andmeid ilusti järjest võtta.

```
using System;
using System.Collections.Generic;
class GeneerilineList{
        static void Main(string[] args)
        {
            LinkedList<int> loetelu = new LinkedList<int>();
            loetelu.AddLast(5);
            loetelu.AddLast(3);
            LinkedList<int>.Enumerator ahel = loetelu.GetEnumerator();
            while (ahel.MoveNext()) {
                Console.WriteLine(ahel.Current);
            }
        }
}
```

Geneerilisi klasse saab ka ise luua. Järgneva näitena pandi kokku lihtne väärtusehoidla. Kui klassi nime taga on kirjeldamise ajal <> märkide vahel täht, siis seda tähte saab klassi sees kasutada andmetüübi kirjeldamiseks. Sarnaselt, nagu võin andmetüübiks märkida int või string, sarnaselt võin andmetüübiks kirjutada T. Ja alles pärast - siis kui klassist luuakse eksemplar. Alles siis määratakse täpsemalt, millist tüüpi seal andmete hoidmiseks tegelikult kasutatakse.

Kui katseprogrammi loomisel kirjutatakse, et

```
        Hoidla<int> h=new Hoidla<int>();
```

siis sellega määratakse selles konkreetses hoidlas hoitavate väärtuste tüübiks int ning midagi muud sinna panna ei saa.

```
namespace Geneeriline1{
 public class Hoidla<T>{
  T sisu;
  public void Pane(T sisu){
     this.sisu=sisu;
  }
```

```
  public T Kysi(){
     return sisu;
  }
 }
 public class Katsetus{
    public static void Main(string[] arg){
       Hoidla<int> h=new Hoidla<int>();
       h.Pane(3);
       System.Console.WriteLine(h.Kysi());
    }
 }
}
```

Kasutatavatele andmetüüpidele saab ka mõningasi piiranguid seada - selleks, et nendega koodis mõnevõrra rohkem midagi hiljem teha oleks. Sest ilma määramata pole isegi teada, kas kasutatav tüüp on struct või class. Neil aga mäluhalduse poolest küllalt erinevad omadused. Kui aga siin teatan, et T on klass, siis on vastavat tüüpi muutujale võimalik anda algväärtuseks null näitamaks, et selle muutuja kaudu ühegi objekti juurde ligi ei pääse. Ning sealtkaudu samuti võimalik küsida, et kas me hoidlas on juba sisu olemas.

```
using System;
namespace Geneeriline2{
 public class Hoidla<T> where T:class{
  T sisu=null;
  public void Pane(T sisu){
     this.sisu=sisu;
  }
  public T Kysi(){
     return sisu;
  }
  public bool KasOlemas(){
    return sisu!=null;
  }
 }
 public class Katsetus{
    public static void Main(string[] arg){
       Hoidla<String> h=new Hoidla<String>();
       h.Pane("Kuku");
       if(h.KasOlemas()){
         System.Console.WriteLine(h.Kysi());
       }
    }
 }
}
```

Enese loodud geneerilistes klassides saab kasutada ka varemvalminud geneeriliste klasside võimalusi. Ehk siis kui on põhjust või tahtmist andmete hoidmist omale sobival moel täiustada või piirata, siis selleks on täiesti võimalus olemas. Omaloodud klassi külge antud geneerilise andmetüübi võib rahumeeles edasi kanda klassi sees loodud andmekollektsiooni eksemplarile ning sinna hiljem vastavat tüüpi andmeid edastada. Siin näites on loodud klass, mis hoiab oma andmeid Listi sees. Tuues muuhulgas aga juurde käskluse loetelust juhusliku elemendi tagastamiseks.

```
using System;
using System.Collections.Generic;
namespace Geneeriline3{
 public class Hoidla<T> where T:class{
  List<T> loetelu=new List<T>();
  Random r=new Random();
  public void Pane(T sisu){
     loetelu.Add(sisu);
  }
  public T Kysi(){
     return loetelu[r.Next(loetelu.Count)];
  }
  public bool KasOlemas(){
    return loetelu.Count>0;
  }
 }
 public class Katsetus{
    public static void Main(string[] arg){
       Hoidla<String> h=new Hoidla<String>();
       h.Pane("Kuku");
       h.Pane("Ahoi");
       h.Pane("Tere");
       if(h.KasOlemas()){
         System.Console.WriteLine(h.Kysi());
       }
    }
 }
}
```

Ülesandeid

* Katseta LinkedListi string-tüüpi andmetega

* Katseta LinkedListi omaloodud klassiga tüübist andmetega

* Loo klass kahe samast tüübist väärtuse hoidmiseks.

* Loo klass, mille puhul saab määrata elementide maksimummäära, mis klassi sees olevasse hoidlasse panna tohib.

Atribuudid

Atribuute kasutakse klassi ja seal sees leiduva omaduste kirjeldamiseks. Teisele programmeerijale võib kommentaaride teel kirja panna, et mida miski käsklus teeb või kuidas oleks seda hea kasutada. Teine rakendus aga üldjuhul inimkeeles kirjutatud kommentaare lugeda ei mõista. Samas on aga hea, kui saab käsule juurde panna seletuse, kuidas seda mõnes graafilises arendusvahendis näitama peaks. Või siis teate, et seda käsku pole vaja praegu käivitada. Atribuudid just sellisteks teadeteks mõeldud on. Mingi hulk on neid olemas .NETi enese poolt. Kui aga omal suurema eripärase raamistiku ehitamine käsil, siis võivad ka omaloodud atribuudid omal kohal olla. Algusnäiteks sobibki tingimuslik käivitamine. Alamprogrammile trüki on lisatud atribuut Conditional. See süsteemne atribuut teatab, et käsklus pannakse tööle vaid juhul, kui konstant nimega TESTSEISUND on defineeritud. Kui sellist konstanti pole, siis jääb käskluse töö lihtsalt tegemata.

Konstanti saab kompileerimisel defineerida näiteks järgnevalt

```
D:\kasutaja\jaagup\proov3>csc /define:TESTSEISUND Logimine1.cs
```

Sellisel juhul kompilaator leiab, et konstant on olemas ning käsk tuleb oma väljakutsel käima panna nagu tavaliselt.

Kui aga kompileerida ilma defineerimata,

```
D:\kasutaja\jaagup\proov3>csc Logimine1.cs
```

siis käsklust ei käivitata. Lihtne.

```
using System;
using System.Diagnostics;
class Logimine1{
   [Conditional("TESTSEISUND")]
   static void tryki(string teade){
      Console.WriteLine(teade);
   }
   public static void Main(string[] arg){
      tryki("algus");
   }
}

/*
D:\kasutaja\jaagup\proov3>csc Logimine1.cs
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.
D:\kasutaja\jaagup\proov3>Logimine1
D:\kasutaja\jaagup\proov3>csc /define:TESTSEISUND Logimine1.cs
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.
D:\kasutaja\jaagup\proov3>Logimine1
algus
*/
```

Defineerida võib ka koodi sees. Et kui parajasti soovitakse abiteateid trükkida, siis kirjutatakse koodi algusesse

```
#define TESTSEISUND
```

ning kompilaatori jaoks ongi vastav konstant olemas ja käsklus käivitatakse. Kui mitte, siis mitte nagu ennegi.

```
#define TESTSEISUND
using System;
using System.Diagnostics;
class Logimine1a{
   [Conditional("TESTSEISUND")]
```

```
   static void tryki(string teade){
      Console.WriteLine(teade);
   }
   public static void Main(string[] arg){
      tryki("algus");
   }
}

/*
D:\kasutaja\jaagup\proov3>csc Logimine1a.cs
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.
D:\kasutaja\jaagup\proov3>Logimine1a
algus
*/
```

Omaloodud atribuut

Atribuute võib ka ise luua. Näiteks selleks, et mõningad käsklused vajadust mööda esile tuua. Selleks tuleb teha klassi System.Attribute alamklass ning sinna vajadust mööda konstruktorisse või omadustesse andmeid lisada. Käskluse lihtsalt ära märkimiseks pole aga muud vaja, kui selle ette atribuut panna. Hiljem ühe klassi koodi sees teist klassi uurides saab uudistada, et millised atribuudid viimase külge pandud on ning mis neist järeldada võib. Siin pannakse käima sellised käsklused, millele on lisatud HuvitavMeetodAttribute. Ja enne seda trükitakse sõna "huvitav", et oleks näha, millise käsklusega tegu on.

```
using System;
using System.Reflection;
namespace Atribuudid2{
   [AttributeUsage(AttributeTargets.Method)]
   class HuvitavMeetodAttribute: System.Attribute{
   }
   public class ValitudMeetodid{
      [HuvitavMeetodAttribute()]
      public void nuputa(){
         Console.WriteLine("Rakendus nuputab");
      }
      public void tervita(){
         Console.WriteLine("Tere");
      }
   }
   public class AtribuudiProov{
      public static void Main(string[] arg){
         ValitudMeetodid v=new ValitudMeetodid();
         MethodInfo[] m= typeof(ValitudMeetodid).GetMethods();
         foreach(MethodInfo mi in m){
            MethodAttributes ma=mi.Attributes;
            foreach(Attribute at in Attribute.GetCustomAttributes(mi)){
               if(at.GetType()==typeof(HuvitavMeetodAttribute)){
                  mi.Invoke(v, null);
                  Console.Write("huvitav ");
               }
            }
            Console.WriteLine(mi.Name);
         }
      }
```

```
    }
}

/*
D:\ctrell>Atribuudid2
Rakendus nuputab
huvitav nuputa
tervita
GetType
ToString
Equals
GetHashCode
*/
```

Atribuutide parameetrid

Atribuutidele kannatab ka andmeid ette anda. Olgu siis kohustuslikud väljad või vaikimisi väärtustega valikulised väljad. Esimesed antakse konstruktori parameetrina, teised omadustena. Ning omadustega atribuudi puhul peab andmete sisestamisel ette ütlema, millise nime alla väärtus läheb.

Järgnevas näites antakse soovitav käivituskordade arv kaasa konstruktorist ning selle väärtuse saab kätte omadusest Kogus. Valikuline atribuut Koostaja antakse sisse omaduse kaudu. Ning nagu näha käskluse Tervita juurest, võib selle ka andmata jätta.

Atribuudispetsiifiliste omaduste jaoks tuleb infokogumise juurest leitud atribuut kõigepealt sobivasse tüüpi muundada.

```
  HuvitavMeetodAttribute ha=at as HuvitavMeetodAttribute;
```

Edaspidi saab sealt andmeid kätte nagu tavalisest objektist. Küsitakse, kelle loodud käsklusega tegu, mitu korda käivitada ning ongi tegutsemisjuhised olemas.

```
using System;
using System.Reflection;
namespace Atribuudid3{
   [AttributeUsage(AttributeTargets.Method)]
   class HuvitavMeetodAttribute: System.Attribute{
      private int _kogus;
      private string _koostaja="tundmatu";
      public HuvitavMeetodAttribute(int Ukogus){
         _kogus=Ukogus;
      }
      public int Kogus{
        get{return _kogus;}
      }
      public string Koostaja{
        get{return _koostaja;}
        set{_koostaja=value;}
      }
   }
   public class ValitudMeetodid{
      [HuvitavMeetodAttribute(3, Koostaja="Jaagup")]
      public void nuputa(){
         Console.WriteLine("Rakendus nuputab");
```

```
      }
      [HuvitavMeetodAttribute(1)]
      public void tervita(){
         Console.WriteLine("Tere");
      }
   }
   public class AtribuudiProov{
      public static void Main(string[] arg){
         ValitudMeetodid v=new ValitudMeetodid();
         MethodInfo[] m= typeof(ValitudMeetodid).GetMethods();
         foreach(MethodInfo mi in m){
            MethodAttributes ma=mi.Attributes;
            foreach(Attribute at in Attribute.GetCustomAttributes(mi)){
               if(at.GetType()==typeof(HuvitavMeetodAttribute)){
                 HuvitavMeetodAttribute ha=at as HuvitavMeetodAttribute;
                 for(int i=0; i<ha.Kogus; i++){
                  mi.Invoke(v, null);
                 }
                 Console.WriteLine("Koostas "+ha.Koostaja);
                 Console.Write("huvitav ");
               }
            }
            Console.WriteLine(mi.Name);
         }
      }
   }
}

/*
E:\jaagup\07\12>Atribuudid3
Rakendus nuputab
Rakendus nuputab
Rakendus nuputab
Koostas Jaagup
huvitav nuputa
Tere
Koostas tundmatu
huvitav tervita
GetType
ToString
Equals
GetHashCode
*/
```

Ülesandeid

* Katseta omaloodud klassis tingimusliku käivitamise atribuuti

* Otsi abiinfost üles atribuut System.Obsolete ning märgi selle abil mõni omaloodud klassi meetod vananenuks. Tutvu teadetega kompileerimisel

* Loo omaloodud atribuut meetodi loomise kuupäeva ning meetodi kohta selgitusi sisaldava URLi hoidmiseks. Kuupäev on kohustuslik, selgituste jaoks pane vaikimisi aadressiks oma koduleht. Koosta koodi uuriv programm, mis atribuudi olemasolul näitaks välja käskluse loomise kuupäeva. Abiinfo URLilt saabuvad andmed salvesta võimaluse korral omaette faili.

Andmebaasiliides

Ühenduse loomine, päring

Kui juhtub, et arvutis või kättesaadavas võrgus on kasutada mõni andmebaas, siis suure tõenäosusega saab sealsete andmete poole pöörduda ja omakoostatud C#-programmi kaudu. Ühe-kaks väärtust võib olla lihtsam baasi enese juurde käiva halduskeskkonna abil paika sättida. Kui aga andmeid on pidevalt ja kümneid. Või siis pole andmetega tegelejaks mitte teie ise, vaid keegi muu. Või soovite, et teated toimingutest jõuaksid automaatselt andmebaasi – sellisel juhul on kasulik koostatud programmilõigu peale mõelda.

Esimene näide eeldab, et masinas nimega RINDE on üles seatud SQL-serveri eksemplar nimega SQLEXPRESS. Ning sinna sisse on loodud andmebaas nimega proovibaas. Ja selles baasis on tabel nimega "inimesed". Ning kõige esimeseks tulbaks inimeste tabelis on eesnimi. Sellisel juhul, kui kõik etapid õnnestuvad, võib loeteluna näha tabelis olevate inimeste eesnimesid. Eks igaüks saab masina, tabeli ja muud andmed enese omade vastu vahetada, ning ongi lihtne üldkasutatav näide, kuidas vajalikke andmeid baasist oma programmi sisse välja meelitada. Mõningad lähemad seletused toimuva kohta.

Muutujasse constr (connection string) salvestatakse kõigepealt seletus programmi jaoks, kust andmebaas üles leida.

```
string constr="Data Source=RINDE\\SQLEXPRESS;"+
  "Initial Catalog=proovibaas; "+
  "Integrated Security=SSPI; Persist Security Info=False";
```

Eraldi muutujasse pannakse kirja SQL-lause, mille abil loodame andmeid küsida.

```
string lause="SELECT eesnimi FROM inimesed";
```

Järgnevalt luuakse ja avatakse ühendus andmebaasiga. See toiming võib mõnikord märgatava hulga sekundeid aega võtta.

```
SqlConnection cn=new SqlConnection(constr);
cn.Open();
```

Et arvuti oskaks andmeid küsida, selleks luuakse SQL-käsklus, kus baasiühendusega seotakse SQL-lause. Siinse vaheetapi juures on keerulisematel juhtudel näiteks võimalus käsklusele parameetreid lisada. Siin aga piirduma lihtsama variandiga.

```
SqlCommand cm = new SqlCommand(lause, cn);
```

Saabuvate andmete püüdmiseks on SQL-serveri puhu `SqlDataReader`. `SqlCommand`i käsklus `ExecuteReader` väljastab vastavat tüüpi objekti, mille kaudu programm omakorda saab andmeid küsima hakata. Selline vaheetapp on vajalik, et programm saaks vajadusel hakkama ka väga suure andmehulgaga. Kui andmete vahendamise jaoks on omaette objekt, kelle kaudu vaikselt andmeid küsima hakatakse, siis ei pea programm saabuvaid andmeid kõiki korraga enesele mällu laadima, vaid jätab selle töö `SqlDataReader`i hooleks.

```
SqlDataReader reader=cm.ExecuteReader();
```

Alles edaspidises tsüklis võetakse inimeste andmed ükshaaval ja toimetatakse nendega. Käsklus `Read` viib lugemiskursori ühe rea võrra edasi – esimesel korral siis esimese inimese juurde. Ning käsklus `GetString` annab etteantud järjekorranumbriga veerust andmed kätte. Nagu näha, hakkavad veerud lugema nullist.

```
while(reader.Read()){
    Console.WriteLine(reader.GetString(0));
}
```

Iga ühenduse kasutamise järel on viisakas see kinni panna. Mis juhtub, kui ühendus lahti jäetakse sõltub juba otsestest oludest. Aga kinni pandult on ressursside raiskamise mure programmeerija käest ära.

```
cn.Close();
```

Ning kood tervikuna.

```
using System;
using System.Data.SqlClient;
class Baasiproov1{
   public static void Main(string[] arg){
     string constr="Data Source=RINDE\\SQLEXPRESS;"+
       "Initial Catalog=proovibaas; "+
       "Integrated Security=SSPI; Persist Security Info=False";
     string lause="SELECT eesnimi FROM inimesed";
     SqlConnection cn=new SqlConnection(constr);
     cn.Open();
     SqlCommand cm = new SqlCommand(lause, cn);
     SqlDataReader reader=cm.ExecuteReader();
     while(reader.Read()){
         Console.WriteLine(reader.GetString(0));
     }
     cn.Close();
   }
}

/*
D:\kodu\0606\dotnet>Baasiproov1
Juku
```

```
Mati
Sass
*/
```

Kui soovitakse baasiga rohkem inimkeeli suhelda ja küsida andmeid veeru pealkirja ja mitte järjekorranumbri järgi, siis sobib arvuti jaoks suupäraseks teisendamiseks `SqlDataReader`i käsklus `GetOrdinal`.

```
using System;
using System.Data.SqlClient;
class Baasiproov1a{
   public static void Main(string[] arg){
     string constr="Data Source=RINDE\\SQLEXPRESS;"+
       "Initial Catalog=proovibaas; "+
       "Integrated Security=SSPI; Persist Security Info=False";
     string lause="SELECT eesnimi FROM inimesed";
     SqlConnection cn=new SqlConnection(constr);
     cn.Open();
     SqlCommand cm = new SqlCommand(lause, cn);
     SqlDataReader reader=cm.ExecuteReader();
     while(reader.Read()){

Console.WriteLine(reader.GetString(reader.GetOrdinal("eesnimi")));
     }
     cn.Close();
   }
}

/*
D:\kodu\0606\dotnet>Baasiproov1
Juku
Mati
Sass
*/
```

Sugugi alati ei pruugi kasutatavad andmed olla Microsofti SQL-serveris. Levinud veidi väiksemate andmetega ümber käimise programmiks on näiteks Access. Kui saab talle sobiva draiveriga sobiva versiooni faili külge minna, siis on täiesti lootust ka nõnda otse sidet pidada. Kusjuures nõnda programmi kaudu andmeid lugedes/kirjutades ei pea Accessi ennast üldse olema masinasse installeeritud. Piisab vaid sobivast draiverist, mis on üldjuhul juba operatsioonisüsteemiga kaasas.

```
using System;
using System.Data.Odbc;
class Baasiproov2a{
   public static void Main(string[] arg){
     string constr="Driver={Microsoft Access Driver (*.mdb)}; "+
       "DBQ=d:\\kodu\\0606\\dotnet\\proovibaas2.mdb; "+
       "Trusted_Connection=yes";
     string lause="SELECT mark FROM autod";
     OdbcConnection cn=new OdbcConnection(constr);
     cn.Open();
```

```
        OdbcCommand cm = new OdbcCommand(lause, cn);
        OdbcDataReader reader=cm.ExecuteReader();
        while(reader.Read()){
            Console.WriteLine(reader.GetString(0));
        }
        cn.Close();
    }
}
```

Kui aga tegemist pole Accessiga, vaid mõne muu andmebaasikeskkonnaga, mis aga sellegipoolest on Control Paneli kaudu ODBC alt kättesaadav, siis ka sealtkaudu saab oma andmetele ligi. Olgu näiteks olemas juba toimiv veebibaas PHP ja MySQLi abil – kuhu aga tahetakse ka kohalikus arvutis toimiva programmi kaudu pilku peale visata. Tehes see algne baas ODBC kaudu nähtavaks nime all näiteks proovibaas2, näeks sidepidamisprogramm välja nagu järgnevalt. Tasub tähele panna, et võrreldes SQL-serveriga on kasutatavateks objektiklassideks `OdbcConnection` ja `OdbcCommand`. Aga andmetega ümber käiakse ikka samamoodi.

```
using System;
using System.Data.Odbc;
class Baasiproov2{
    public static void Main(string[] arg){
        string constr="DSN=proovibaas2";
        string lause="SELECT mark FROM autod";
        OdbcConnection cn=new OdbcConnection(constr);
        cn.Open();
        OdbcCommand cm = new OdbcCommand(lause, cn);
        OdbcDataReader reader=cm.ExecuteReader();
        while(reader.Read()){
            Console.WriteLine(reader.GetString(0));
        }
        cn.Close();
    }
}
```

Kui baasist küsitakse vaid ühte väärtust ja mitte tervet tabelit, siis selle tarvis on .NET andmebaasi programmeerimise vahendite juurde loodud lihtne ja asjalik käsklus: `ExecuteScalar`. Küsitakse vaid inimeste arv, siis ühe käsuga saab selle kätte ilma, et peaks vahepeal andmepuhvrit looma.

```
using System;
using System.Data.SqlClient;
class Baasiproov1b{
    public static void Main(string[] arg){
        string constr="Data Source=RINDE\\SQLEXPRESS;"+
          "Initial Catalog=proovibaas; "+
          "Integrated Security=SSPI; Persist Security Info=False";
        string lause="SELECT COUNT(*) FROM inimesed";
        SqlConnection cn=new SqlConnection(constr);
        cn.Open();
        SqlCommand cm = new SqlCommand(lause, cn);
        Console.WriteLine("Inimeste arv: "+cm.ExecuteScalar());
```

```
      cn.Close();
    }
}

/*
D:\kodu\0606\dotnet>Baasiproov1b
Inimeste arv: 3
*/
```

Andmete lisamine

Eraldi moodus on käskluste jaoks, mis pole päringud: lisamine, muutmine, kustutamine. SQL-lause tuleb valmis teha nagu ikka, käivitamise jaoks aga käsklus ExecuteNonQuery.

```
using System;
using System.Data.SqlClient;
class Baasiproov1c{
   public static void Main(string[] arg){
     string constr="Data Source=RINDE\\SQLEXPRESS;"+
       "Initial Catalog=proovibaas; "+
       "Integrated Security=SSPI; Persist Security Info=False";
     string lause="INSERT INTO inimesed (eesnimi) VALUES ('Siiri')";
     SqlConnection cn=new SqlConnection(constr);
     cn.Open();
     SqlCommand cm = new SqlCommand(lause, cn);
     cm.ExecuteNonQuery();
     Console.WriteLine("Andmed lisatud");
     cn.Close();
   }
}

/*
D:\kodu\0606\dotnet>Baasiproov1c
Andmed lisatud
*/
```

SQL-parameeter

Tahtes kasutaja andmeid SQL-lausesse lisada – olgu siis andmete küsimisel piirangu seadmiseks või andmete lisamisel salvestamiseks – on seda hea teha parameetri kaudu. Sellisel juhul ei pea SQL-lauset koostades muret tundma, et kasutaja sisestus kuidagi jutumärke või ülakomasid omavahel sõlme võiks ajada. Samuti püsib nõnda ka lause ülesehitus selgemana – lihtsalt @-märgiga tähistatud sõna asendatakse pärast sobiva väärtusega. Siin tähistatakse `@algaasta`ga arv, millisest aastast alates sündinud laste nimesid tahetakse näha.

```
     string lause="SELECT eesnimi, synniaasta FROM lapsed "+
        "WHERE synniaasta >= @algaasta";
```

Enne käskluse käivitamist tuleb siis parameetri kohta sobiv väärtus paigutada. Siin näites on kaks toimingut ühel real. Kõigepealt lisatakse käsklusele cm parameeter nimega @algaasta. Seejärel

omistatakse loodud objekti omadusele Value väärtus 1997. Viimatine aastaarv on praegu küll kirjutatud lühiduse mõttes arvuna koodi, aga sinna saab paigutada kasutajalt tulnud väärtuse.

```
cm.Parameters.Add("@algaasta", SqlDbType.Int).Value=1997;
```

Vastuseks saame andmete loetelu nii nagu ikka.

```
using System;
using System.Data;
using System.Data.SqlClient;
class Baasiparameeter1{
   public static void Main(string[] arg){
     string constr="Data Source=RINDE\\SQLEXPRESS;"+
       "Initial Catalog=baas1; "+
       "Integrated Security=SSPI; Persist Security Info=False";
     string lause="SELECT eesnimi, synniaasta FROM lapsed "+
        "WHERE synniaasta >= @algaasta";
     SqlConnection cn=new SqlConnection(constr);
     cn.Open();
     SqlCommand cm = new SqlCommand(lause, cn);
     cm.CommandType=CommandType.Text;
     cm.Parameters.Add("@algaasta", SqlDbType.Int).Value=1997;
     SqlDataReader reader=cm.ExecuteReader();
     while(reader.Read()){
         Console.WriteLine(reader.GetString(0)+": "+
             reader.GetInt32(1));
     }
     cn.Close();
   }
}

/*
D:\kodu\0606\dotnet>Baasiparameeter1
Juku: 1997
Kati: 1997
Siim: 1997
*/
```

Salvestatud protseduur

Tahtes programmi kaudu salvestatud protseduurile andmeid jagada on kasulik jällegi parameetrit pruukida. Parameetrile antakse nime järgi väärtus ning protseduur saabki selle kätte. Siinne protseduur loodi käsuga

```
CREATE PROCEDURE kysiLapsed(@algaasta decimal)
AS
SELECT eesnimi, synniaasta FROM lapsed
WHERE synniaasta>=@algaasta
```

Andmete lugemine tabeli väljastavast salvestatud protseduurist on sarnane hariliku päringu vastuste lugemisele. Ning tulemus on samuti eelmise näitega sarnane.

```
using System;
using System.Data;
using System.Data.SqlClient;
class Baasiparameeter1{
   public static void Main(string[] arg){
      string constr="Data Source=RINDE\\SQLEXPRESS;"+
        "Initial Catalog=baas1; "+
        "Integrated Security=SSPI; Persist Security Info=False";
      string lause="kysiLapsed";
      SqlConnection cn=new SqlConnection(constr);
      cn.Open();
      SqlCommand cm = new SqlCommand(lause, cn);
      cm.CommandType=CommandType.StoredProcedure;
      cm.Parameters.Add("@algaasta", SqlDbType.Int).Value=1997;
      SqlDataReader reader=cm.ExecuteReader();
      while(reader.Read()){
          Console.WriteLine(reader.GetString(0)+": "+
              reader.GetInt32(1));
      }
      cn.Close();
   }
}
/*
D:\kodu\0606\dotnet>Baasiparameeter2
Juku: 1997
Kati: 1997
Siim: 1997
*/
```

Ülesandeid

* Loo andmebaasitabelid: maakonnad (id, maakonnanimi), autod (id, mark, aasta, maakonna_id). Lisa mõned andmed.

* Küsi programmi abil ekraanile kõikide maakondade nimed

* Küsi ekraanile kõikide autode andmed koos maakondade nimedega, kus nad registreeritud.

* Loo käsklus uue auto lisamiseks. Sisendiks mark, aasta, maakonna_id.

* Olematu id-ga maakonna puhul näidatakse olemasolevaid maakondade nimesid ja id-sid ning palutakse viimane uuesti sisestada.

* Loo võimalus etteantud id-ga auto maakonna muutmiseks.

* Loo salvestatud protseduur näitamaks enne parameetrina antud aastat tehtud autosid. Käivita protseduur ja vaata tulemusi C# kaudu.

* Loo klassid auto ning maakonna andmete hoidmiseks. Katseta paari väärtusega.

* Loo klass AutoRegister toimetamaks andmebaasis paiknevate autodega. Järjekorranumbri abil peetakse meeles jooksvat autot, samuti on klassi eksemplaril meeles baasis leiduvate autode id-d (ArrayListina). Lisa klassile käsud jooksva auto küsimiseks, andmete muutmiseks, id järgi

järgmise/eelmise auto juurde liikumiseks, auto lisamiseks lõppu. Andmete küsimisel väljastatakse auto klassi eksemplar, mis viitab maakonnaeksemplarile.

* Ehita AutoRegistri klassi ümber kasutajaliides, mille kaudu on võimalik käsurealt autode andmetega toimetada: loetelu vaadata, aktiivset autot üles/alla määrata, aktiivse auto andmeid muuta, autosid lisada/kustutada. Autosid soovitud tunnuse alusel sorteerida/filtreerida.

Funktsiooni delegaadid

Kui on programmi käitumise juures vaja valida mitme võimaluse vahel, siis tavalisimaks vahendiks on `if`/`else` plokk, millega tööjärge suunata.

Objektorienteeritud programmeerimise juures kui soovitakse objektid sarnastes kohtades erinevalt käituma panna, siis luuakse algsele klassile iga märgatavalt erineva käitumismooduse jaoks alamklass ning selles kaetakse vastav meetod üle nõnda nagu see parasjagu vajalik on. Hiljem saab soovitud käitumise väljakutsumise jaoks anda sobivasse kohta ette õige alamklassi sobivalt seadistatud eksemplari. Nõnda tuleb näiteks `ArrayList`i omapoolselt määratud järjestusse sorteerides ette anda liidest `IComparer` realiseeriva klassi eksemplar, mille juures tuleb üle katta funktsioon nimega `Compare`. Nimetatud meetodi ülesandeks on kahe etteantud objekti põhjal otsustada, kumb neist järjestuses ettepoole saab. Sedasi võib tekstid järjestada näiteks pikkuse ja mitte tähestiku järgi.

Kes on esimese keelena kirjutanud Cs või C++is, sel tekib nõnda alamklasse luues kohe küsimus, et miks nii keeruliselt peab tegema. Et eelnimetatud keeltes kasutatakse selliseks sortimisjärjekorra määramiseks funktsiooniviitu. Kuna C# püüab eneses ühendada rangemate keelte struktueeritust ning masinalähedasemate keelte võimalusi, siis on siia loodud funktsiooni delegaadid. Lahtiseletatult on tegemist eripärase muutujaga, mille kaudu saab selle külge omistatud funktsiooni välja kutsuda. Seletus järgmise näite põhjal.

Tervituseks on loodud kaks funktsiooni: `RahulikTervitus` ning `TragiTervitus`. Programmi mingis osas saab valida, millist funktsiooni neist kasutatakse. Et funktsiooni delegaadina meelde jätta, tuleb see programmi algul deklareerida.

```
public delegate void Tervitusfunktsioon();
```

Edasi võib sobival ajal omistada loodud delegaatmuutujale tegeliku funktsiooni eksemplari.

```
Tervitusfunktsioon tervitaja=new Tervitusfunktsioon(RahulikTervitus);
```

Delegaatmuutujale sulgude taha kirjutamisega pannaksegi soovitud funktsioon tööle.

```
tervitaja();
```

Ja tulemusena näeme sel korral lihtsat "Tere".

```
using System;
public delegate void Tervitusfunktsioon();
class Delegaat1{
   static void RahulikTervitus(){
      Console.WriteLine("Tere");
   }
```

```
    static void TragiTervitus(){
       Console.WriteLine("Ahoi!");
    }
    public static void Main(string[] arg){
       Tervitusfunktsioon tervitaja=new Tervitusfunktsioon(RahulikTervitus);
       tervitaja();
    }
}
/*
D:\kodu\0606\dotnet>Delegaat1
Tere
*/
```

Funktsioonide komplekt

Mõnikord tuleb talletada terve tegevuste jada. Et delegaatmuutujad käituvad küllalt tavaliste objektide sarnaselt, saab neid ka kogumis (siin `ArrayList`) talletada ja pärast välja kutsuda. Sedasi on võimalik ühes koodiosas teine käsujada kokku panna ja seda hiljem pruukida.

```
using System;
using System.Collections;
public delegate void Tervitusfunktsioon();
class Delegaat2{
    static ArrayList tervitused=new ArrayList();
    static void RahulikTervitus(){
       Console.WriteLine("Tere");
    }
    static void TragiTervitus(){
       Console.WriteLine("Ahoi!");
    }
    public static void Main(string[] arg){
       tervitused.Add(new Tervitusfunktsioon(RahulikTervitus));
       tervitused.Add(new Tervitusfunktsioon(TragiTervitus));
       tervitused.Add(new Tervitusfunktsioon(RahulikTervitus));
       foreach(Tervitusfunktsioon tervitaja in tervitused){
          tervitaja();
       }
    }
}

/*
D:\kodu\0606\dotnet>Delegaat2
Tere
Ahoi!
Tere
*/
```

Sündmused

Sarnane käskude jada salvestamine on mõnes olukorras nõnda tavaline, et selle jaoks on eraldi välja mõeldud tüüp event. Nii nagu võis `Add` käsuga lisada andmeid `ArrayList`i, nii saab += operaatoriga panna juurde funktsioonidelegaate sündmuste loendisse. Ja tulemus sarnane nagu eelmisel korral. Ette rutates märkides kasutatakse sellist delegaatide sündmusteahelasse lisamist näiteks graafilistes

programmides nupuvajutuse korral, kus iga nupu alla võib panna käivituma mitu funktsiooni. Siin aga lihtsalt taas tervitused.

```
using System;
public delegate void Sisenemine();
class Syndmused1{
   static event Sisenemine InimeneSisenes;
   static void RahulikTervitus(){
      Console.WriteLine("Tere");
   }
   static void TragiTervitus(){
      Console.WriteLine("Ahoi!");
   }
   public static void Main(string[] arg){
      InimeneSisenes+=new Sisenemine(RahulikTervitus);
      InimeneSisenes+=new Sisenemine(TragiTervitus);
      InimeneSisenes+=new Sisenemine(RahulikTervitus);
      InimeneSisenes();
   }
}

/*
D:\kodu\0606\dotnet>Syndmused1
Tere
Ahoi!
Tere
*/
```

Ilmajaamad

Järgnevalt eelneva põhjal kokku pandud näide, kus piirkonda hajusalt paigutatud 10 vaatlusjaama omi mõõteandmeid keskusesse saadavad ning kuidas saabunud andmetele reageeritakse.

Peaprogramm loob kõigepealt ilmajaamade objektid, siis määrab nende juures, milline funktsioon panna käima tavateate juures ning mida teha juhul, kui tegemist on mõne hoiatusega. += operaator lubab vajadusel panna vastava sündmuse külge ka mitu teadet või jätta sootuks ilma teateta.

Edasi palutakse kõigil jaamadel välja mõelda (mõõta) andmed ning keskusesse saata teated nende kohta. Ja siis küsitakse kasutajalt, kas ta soovib veel uuesti andmeid küsida. Et teabe trükkimise funktsioonid olid juba ilmajaamade teatesündmuste külge lisatud, siis lähevad nad sündmuste käivitamisel ise käima.

Jaamal on meeles enese number. Teate saatmisel küsitakse andurilt temperatuur ning pannakse kokku keskusesse minev vajalik andmeplokk. Temperatuuri küsimisel arvestab programm kuu järjekorranumbrit ning selle suhtes arvutab välja sellele aastaajale tõenäolise temperatuuri - lihtsalt üks viis kuigivõrd usutavaid vastuseid simuleerida.

```
   int kysiTemperatuur(){
      return 15-Math.Abs(6-System.DateTime.Now.Month)*5+
                                arvugeneraator.Next(20);
   }
```

Et sündmuse käivitamisel saaks sinna andmeid kaasa saata, selleks peavad andmed olema koos ühes terviklikus objektis ning selle objekti klass peab olema klassi `EventArgs` alamklass - mis siis lubab neil andmetel teatega kaasa minna.

```
public class IlmajaamaParameetrid: EventArgs{
  ...
}
```

Kui jaam teateid saadab, siis kõigepealt pannakse kokku eelnäidatud tüübist andmeplokk. Edasi käivitatakse igal juhul tavateate saatmine (sündmus) ning kui põhjust, siis selle järel ka hoiatusteate saatmine. Mis teatesaatmissündmuse puhul tegelikult tehakse, on eespool ilmajaama loomise juures juba kirjeldatud.

```
public void saadaTeated(){
   IlmajaamaParameetrid andmed=new IlmajaamaParameetrid(
                            jaamaNr, ++teateNr, kysiTemperatuur());
   tavaTeade(andmed);
   if(andmed.temperatuur<0 ||
                       andmed.temperatuur>20){hoiatusTeade(andmed);}
}
```

Ja ongi kogu näide, mida võib edaspidi sarnaste rakenduste loomisel aluseks võtta.

```
using System;
namespace Ilmaandmed{
  public delegate void IlmajaamaTeade(IlmajaamaParameetrid p);
  class Syndmused2{
   static void HarilikTeave(IlmajaamaParameetrid p){
      Console.WriteLine(p);
   }
   static void HoiatusTeave(IlmajaamaParameetrid p){
      Console.WriteLine("     Hoiatus: "+p);
   }
   public static void Main(string[] arg){
      Ilmajaam[] jaamad=new Ilmajaam[10];
      for(int i=0; i<jaamad.Length; i++){
         jaamad[i]=new Ilmajaam();
         jaamad[i].tavaTeade+=new IlmajaamaTeade(HarilikTeave);
         jaamad[i].hoiatusTeade+=new IlmajaamaTeade(HoiatusTeave);
      }

      string vastus;
      do{
       for(int i=0; i<jaamad.Length; i++){
        jaamad[i].saadaTeated();
       }
       Console.WriteLine("Kas veel?");
       vastus=Console.ReadLine();
      } while(vastus.ToLower().StartsWith("j"));
   }
  }

  class Ilmajaam{
```

```
    static int jaamadeArv=0;
    int jaamaNr;
    int teateNr=0;
    public event IlmajaamaTeade tavaTeade;
    public event IlmajaamaTeade hoiatusTeade;
    private Random arvugeneraator;
    public Ilmajaam(){
       jaamaNr=++jaamadeArv;
       arvugeneraator=new Random(jaamaNr);
    }
    int kysiTemperatuur(){
       return 15-Math.Abs(6-System.DateTime.Now.Month)*5+
                                  arvugeneraator.Next(20);
    }
    public void saadaTeated(){
       IlmajaamaParameetrid andmed=new IlmajaamaParameetrid(
                             jaamaNr, ++teateNr, kysiTemperatuur());
       tavaTeade(andmed);
       if(andmed.temperatuur<0 || andmed.temperatuur>20){
                               hoiatusTeade(andmed);}
    }
   }
   public class IlmajaamaParameetrid: EventArgs{
      public readonly int temperatuur;
      public readonly int jaamaNr;
      public readonly int teateNr;
      public IlmajaamaParameetrid(int uusJaamaNr, int uusTeateNr,
                                             int uusTemperatuur){
         temperatuur=uusTemperatuur;
         jaamaNr=uusJaamaNr;
         teateNr=uusTeateNr;
      }
      public override string ToString(){
         return "Jaam "+jaamaNr+", teade "+teateNr+" temperatuur "+
                                                  temperatuur;
      }
   }
}
/*
D:\kodu\0606\dotnet>Syndmused2
Jaam 1, teade 1 temperatuur 14
Jaam 2, teade 1 temperatuur 25
     Hoiatus: Jaam 2, teade 1 temperatuur 25
Jaam 3, teade 1 temperatuur 15
Jaam 4, teade 1 temperatuur 26
     Hoiatus: Jaam 4, teade 1 temperatuur 26
Jaam 5, teade 1 temperatuur 16
Jaam 6, teade 1 temperatuur 27
     Hoiatus: Jaam 6, teade 1 temperatuur 27
Jaam 7, teade 1 temperatuur 17
Jaam 8, teade 1 temperatuur 28
     Hoiatus: Jaam 8, teade 1 temperatuur 28
Jaam 9, teade 1 temperatuur 18
Jaam 10, teade 1 temperatuur 29
     Hoiatus: Jaam 10, teade 1 temperatuur 29
Kas veel?
j
Jaam 1, teade 2 temperatuur 12
Jaam 2, teade 2 temperatuur 18
Jaam 3, teade 2 temperatuur 23
     Hoiatus: Jaam 3, teade 2 temperatuur 23
```

```
Jaam 4, teade 2 temperatuur 29
     Hoiatus: Jaam 4, teade 2 temperatuur 29
Jaam 5, teade 2 temperatuur 15
Jaam 6, teade 2 temperatuur 21
     Hoiatus: Jaam 6, teade 2 temperatuur 21
Jaam 7, teade 2 temperatuur 27
     Hoiatus: Jaam 7, teade 2 temperatuur 27
Jaam 8, teade 2 temperatuur 13
Jaam 9, teade 2 temperatuur 19
Jaam 10, teade 2 temperatuur 25
     Hoiatus: Jaam 10, teade 2 temperatuur 25
Kas veel?
e
*/
```

Graafiline liides

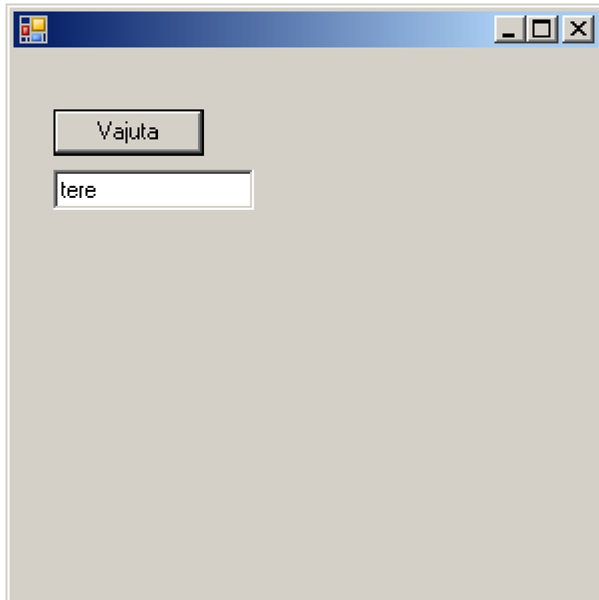
Lõppu veel näide, kuidas sündmusi graafilise keskkonna juures pruugitakse.

Windowsi aknarakenduse puhul on lihtsaks võimaluseks luua oma rakendus klassi Form alamklassina. Graafilised elemendid saab klassi algul ära kirjeldada ning konstruktoris ekraanile paigutada. Ekraanikoordinaatide järgi paigutus akna ülanurga suhtes on üks võimalusi. Kõigepealt määratakse graafikakomponendid asukohapunkti järgi paika ning `Controls.Add` käsu abil jõuavad nad sinna nähtavana ekraanile. Tahtes nupuvajutussündmuse peale miskit käivitada, saab eelpoolvaadatud += operaatori abil lisada omaloodud funktsiooni `Click`-nimelisele sündmusele.

`nupp1_Click` funktsioonile tulevad kaks parameetrit. Esimese kaudu saab eristada millist nuppu vajutati - juhul kui on oht, et sündmusi võib mitmelt poolt tulla. ning teisest võiks teavet saada näiteks hiire andmete kohta vajutushetkel. Aga kuna praegu meil on tegemist ainukese nupuga ning soovime käima panna lihtsa teretuse, siis pole meil neid parameetreid endid pruukida vaja.

```
using System;
using System.Windows.Forms;
using System.Drawing;

class Tervitaja2:Form{
  Button nupp1=new Button();
  TextBox tekst1=new TextBox();
  public Tervitaja2(){
     nupp1.Location=new Point(20, 30);
     tekst1.Location=new Point(20, 60);
     nupp1.Text="Vajuta";
     Controls.Add(nupp1);
     Controls.Add(tekst1);
     nupp1.Click+=nupp1_Click;
  }
  void nupp1_Click(object saatja, EventArgs e){
     tekst1.Text="tere";
  }
  public static void Main(string[] arg){
     Application.Run(new Tervitaja2());
  }
}
```



Ülesandeid

* Uuri näidet `Delegaat1`, lisa sinna funktsioon `UinuvTervitus`, mis trükiks "brrr". Katseta.

* Muuda näidet `Syndmused1` nõnda, et lisanduks sündmus `InimeneUinus`, kuhu oleks tsükli abil 10 korda lisatud `UinuvTervitus`. Katseta.

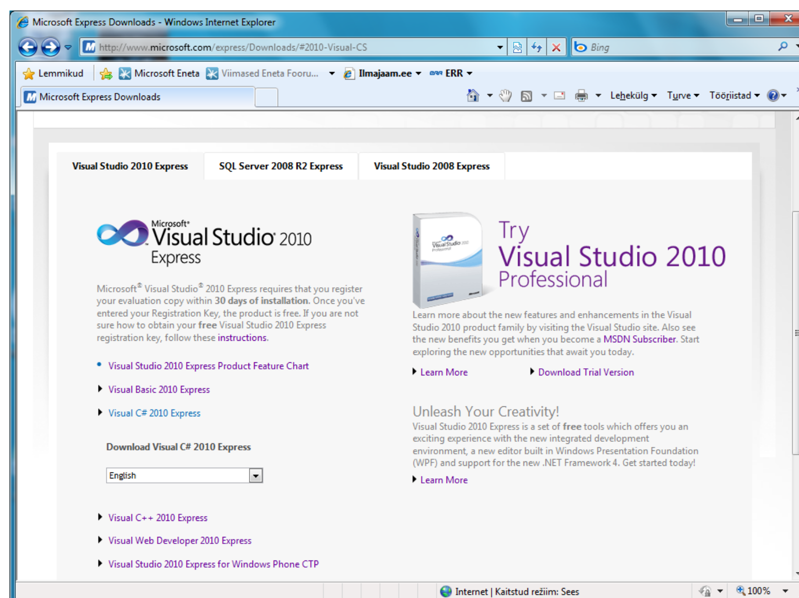
* Lisa ilmajaama näitele tuule kiirus.

* Koosta nupuvajutusnäite abil graafiline kalkulaator, mille abil saab sentimeetreid tollideks ja tagasi arvutada.

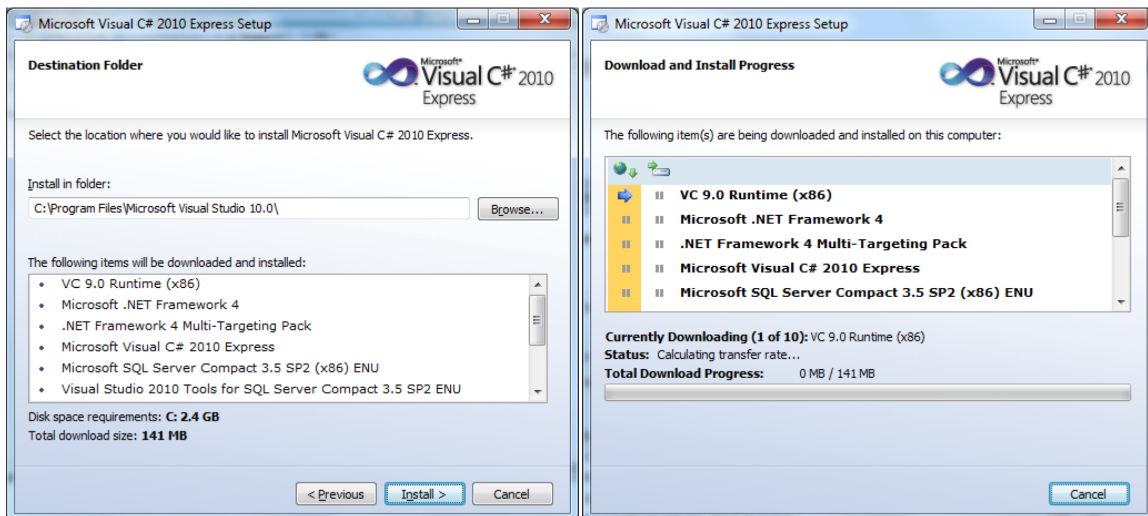
Visual Studio C# Expressi install

Programmikoodi kirjutamise ja käivitamise jaoks on loodud mitmesuguseid abivahendeid. Microsofti ametlikuks graafiliseks arenduskeskkonnaks on Visual Studio. Express-versioonid sellest on kõigile vabalt kasutatavad. Tutvumiseks ja allalaadimiseks sobib veebiaadress www.microsoft.com/express/

Selle juurest Windows-valiku alt leiab Visual Studio C# 2010 Expressi.

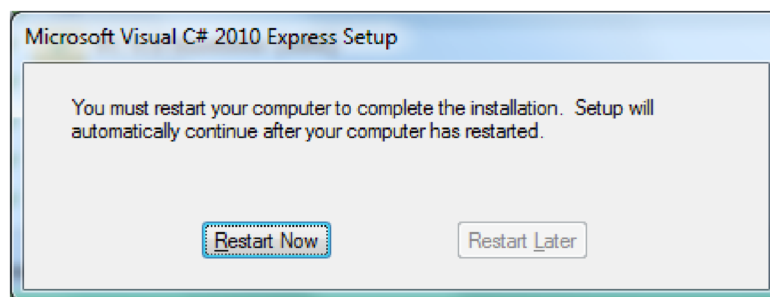


Programmi allalaadimisel salvestatakse sealt kõigepealt mõne megabaidinefail, mis käivitades omakorda ülejäänud vajalikud andmed kohale laeb. Nagu ikka, tasub lihtsama installeerimise huvides programmi pakutavate valikutega nõus olla. Ühe etapina tasub vaadata kataloogi, kuhu rakenduse paigaldatakse nagu allolevalt pildilt näha. Kui masin leitakse rakenduse jaoks sobiv olema, siis hakkab installimistöö peale, mis olenevalt masina jõudlusest ja võrgu ühenduskiirusest võib kesta kümnekonnast minutist paari tunnini.

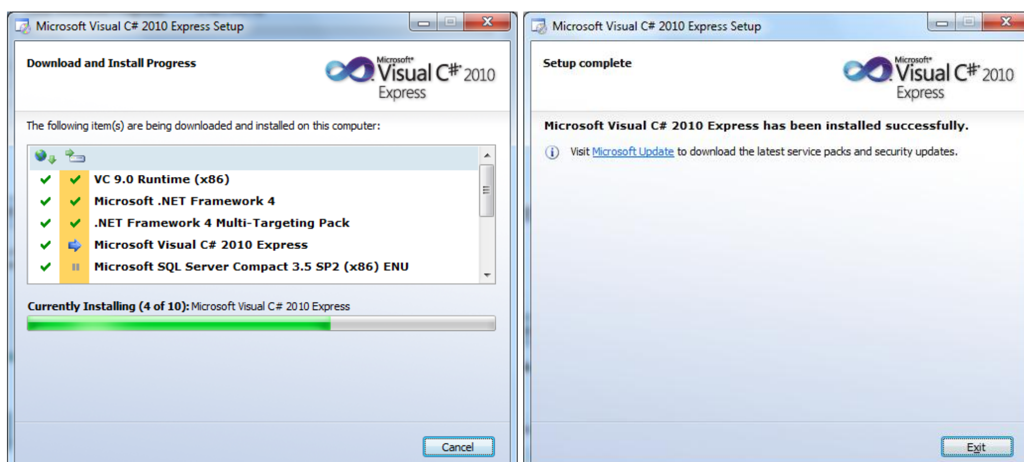


´

Pärast mõningat ketta ragistamist, kui uus raamistiku versioon on paika seatud, tuleb arvutile restart teha, et järgmiste komponentide install saaks jätkuda.

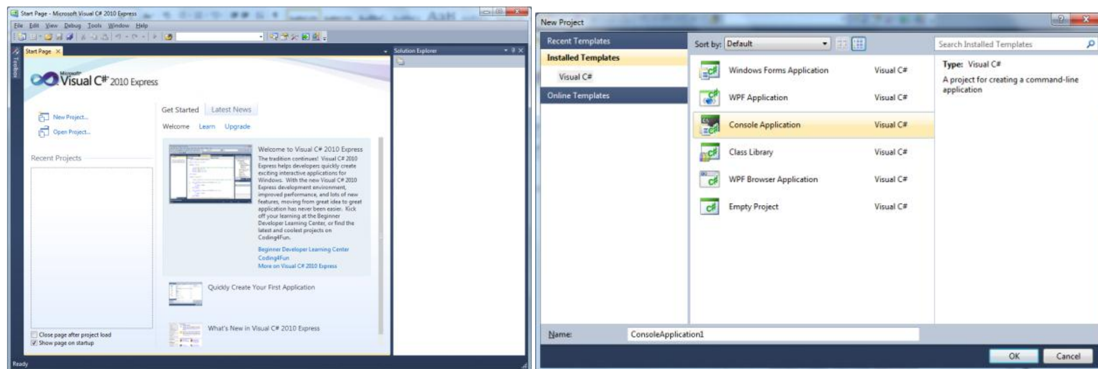


Pärast veel mõnda aega paigaldustöid on lootus näha installi lõppakent teatega, et kõik läks õnneks.

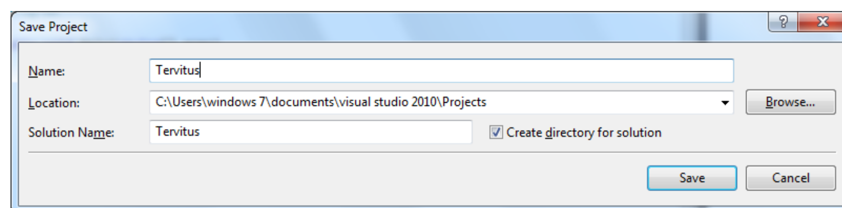


Esmase rakenduse loomine.

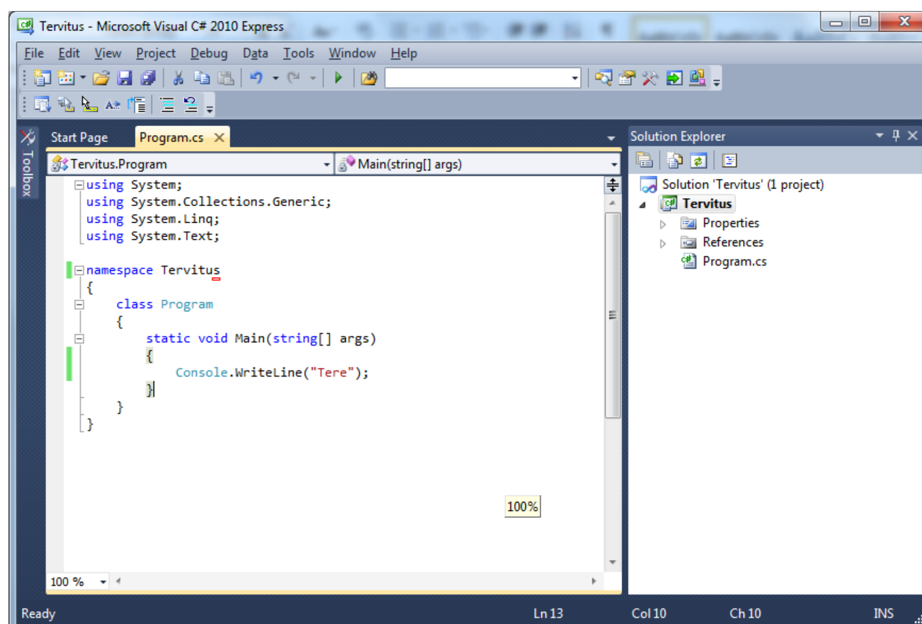
Keskkonna abil kirjutades on lihtsaim tee lasta luua uus projekt. Olgu siis uhkelt avalehelt või hiljem vastavast menüüst. Et siinses õppematerjalis näitatakse keele põhikonstruktsioonid käsurearakenduste abil, siis tasub ka projektivalikust see võtta.



Soovides vaikimisi pandud ConsoleApplication1 asemele midagi enesele äratuntavamat panna, tuleb see nimi projektile salvestamisel anda.



Ning samuti on kasulik siis ka rakenduse nimeruumile sama nimi panna.



Rohelisele noolele või F5le vajutades saab rakenduse käivitada, kuid pilt kaob eest kohe töö lõppedes. Rahumeeli saab toimuvat imetleda käsurealt käivitades. Tuleb minna kausta, kuhu .exe fail on kompileeritud ning see käima panna.

```
C:\Users\windows 7\Documents\Visual Studio
2010\Projects\Tervitus\Tervitus\bin\Debug>Tervitus.exe
Tere
```

Nii võibki esimest töötavat näidet imetleda ning edasi asuda konspektist juba keerukamaid peatükke omale selgeks tegema.

LINQ - .NET Language-Integrated Query

LINQ lisab programmeerimiskeelde (nagu C#, VB.NET jne) päringute kirjutamise käsustiku, mille abil on võimalik andmeid valida, filtreerida ning teha kokkuvõtteid.

Kõik LINQ vahendid paiknevad System.Linq nimeruumis.

All olev näide kasutab LINQ päringut selleks, et töödelda massiivis olevaid andmeid.

```
using System;
using System.Linq;
using System.Collections.Generic;
class app {
  static void Main() {
    string[] names = { "Burke", "Connor", "Frank",
                       "Everett", "Albert", "George",
                       "Harris", "David" };
    IEnumerable<string> query = from s in names
                                where s.Length == 5
                                orderby s
                                select s.ToUpper();
    foreach (string item in query)
      Console.WriteLine(item);
  }
}
```

Tulemuseks on:

```
BURKE
DAVID
FRANK
```

LINQ mõistmiseks tuleks esmalt vaadelda programmi esimest lauset:

```
IEnumerable<string> query = from s in names
                            where s.Length == 5
                            orderby s
                            select s.ToUpper();
```

Kohalik muutuja Query väärtustatakse päringu avalisega. Päringuavaldis võtab andmeid ühest või mitmest andmeallikast ning võimaldab sooritada mitmeid operatsioone. Antud päring kasutab kolme standardset operaatorit: Where, OrderBy ja Select.

Seega võiks seda päringut lugeda järgmiselt: loome loendatava teksti sisaldava massiivi Query, valides andmed tekstimassiivist names kus teksti pikkuseks on 5 märki ning sorteerime tulemuse tähestikuliselt ning tulemusena näitame sobivaid tekstiväärtusi suurte tähtedega.

LINQ päringuid saab teha kõigi loendatavate kollektsioonide ja massiivide peal.

Kõik LINQ päringud koosnevad kolmest tegevusest:

- Andmeallika tekitamine
- Päringu loomine
- Päringu käivitamine

```
using System;
using System.Linq;
using System.Collections.Generic;
class app {
  static void Main() {
        // Linq päringu kolm osa:
        //  1. Andmeallika loomine.
        int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };
        // 2. Päringu loomine.
        // numQuery tekitatakse IEnumerable<int> tüüpi
        var numQuery =
            from num in numbers
            where (num % 2) == 0
            select num;
        // 3. Päringu käivitamine.
        foreach (int num in numQuery)
        {
            Console.Write("{0,1} ", num);
        }
  }
}
```

Põhikonstruktsioonide näited

Lühim tervitav programm

```
using System;
class Tervitus{
   public static void Main(string[] arg){
      Console.WriteLine("Tere");
   }
}
```

Valik

```
      if(eesnimi=="Mari"){
         Console.WriteLine("Tule homme minu juurde!");
      } else {
         Console.WriteLine("Mind pole homme kodus.");
      }
```

Kordus while abil

```
      int nr=1;
      while(nr<=5){
         Console.WriteLine("Tere, {0}. matkaja!", nr);
         nr=nr+1;
      }
```

Sama kordus for-i abil

```
      for(int nr=1; nr<=5; nr++){
         Console.WriteLine("Tere, {0}. matkaja!", nr);
      }
```

Massiivi algväärtustamine ning foreach-kordus

```
      int[] m=new int[3]{40, 48, 33};
      foreach(int arv in m){
        Console.WriteLine(arv);
      }
```

Massiivi loomine ja kasutamine

```
      int[] m=new int[3];
      m[0]=40;
      m[1]=48;
      m[2]=33;
      Console.WriteLine(m[1]);
```

Kokkuvõte

Eelpoolsetel lehekülgedel alustati C# pisikese tervitava programmiga ning käidi läbi enamik tähtsamaid keele süntaksi ja ülesehituse juurde kuuluvaid teemasid. Hulk nüansse ja erijuhte jäi käsitlemata, kuid olemasolevate põhjal peaks õnnestuma enamik enesele tarvilikke programme kokku panna ja ette juhtunud valmisprogrammidest aru saada. Samuti võiks pärast siinse kirjutise läbitöötamist tekkida piisav karkass ja kogemus, mille külge on kergem mujalt leitud C# ja objektorienteeritusega seotud lõike haakida.

Siinsed programmid käivitati käsurealt. Kuid nagu lõpunäitest näha, võib ka käsurearakendustel olla täiesti graafiline liides. Kui edasi uurida System.Windows.Forms nimeruumi käske ja näiteid, siis sealtkaudu võib oma rakendusele üha uusi graafilisi vidinaid juurde lisada. Kui veel käivitada mitte mustast käsureaaknast, vaid seada töölauale sobiv ikoon, siis ongi "harilikule" kasutajale tavaline rakendus valmis.

C# loodi veebiajastul. Selle tõttu on tal kasutada mitmesuguseid mooduseid võrgu kaudu suhtlemiseks. Nii otseühenduses üle TCP-protokolli näiteks jututoa või võrgumängu tarbeks kui ka kogu tehnikate komplekt veebirakenduste loomiseks. Sealne käivitus näeb tunduvalt teistsugune välja, kui siin loodetavasti tuttavaks saanud `public static void Main`. Aga sellegipoolest jääb keele põhisisu samaks. Nii muutujad, tsüklid, valikud, klassid ja objektid on endistviisi vajalikud. Lihtsalt tuleb neile osalt uus ja parasjagu vajalikule toimingule vastav sisu kokku panna.