

3. Ievads programmēšanā ar Visual C#

Šajā nodaļā apskatīsim vizuālās programmēšanas pamatkonceptijas, definēsim programmas struktūru un izpildes secību, kā arī programmas koda noformēšanas ieteikumus („labas programmēšanas stils”).

3.1. Vizuālās programmēšanas pamatkonceptijas

Windows vai tīmekļa aplikācijas izveidošanas laikā tiek izveidota uz notikumiem bāzētā lietotne. Šāda tipa lietotnes izpilda kodu reaģējot uz dažāda veida notikumiem, kurus inicializē lietotājs vai izpildāmais kods. Katrai vadīklai ir sava specifiskā notikumu kopa. Jebkura programmas koda rinda (priekšraksts) var tikt izpildīts tikai reaģējot uz notikumu (peles klikšķis, ievads no tastatūras vai operētājsistēmas darbības).

Notikumu bāzētai programmēšanai piemīt šādas īpatnības:

- programmas izpildes gaita galvenokārt tiek regulēta ar ārējiem notikumiem;
- šīs programmēšanas veids ir elastīgs un var reaģēt uz dažādiem notikumiem un ievadāmām vērtībām;
- atbalsta neatkarīgu uzvedību un ļauj iztikt ar pēc iespējas mazāku ierobežojumu skaitu.

Programmēšanas valoda C# ir objektorientētā, tādēļ tās pamata vienības ir klases un objekti. **Klase** ir fundamentālais lietotāja definētais tips, kurš organizē informāciju noteiktā veidā. Šī organizācija ir līdzīga klasifikācijai; sākumā jādefinē klase, tikai pēc tam var izveidot šīs klases objektus (turpmākās nodaļas tas tiks apskatīts sīkāk).

Klase definē objekta pamata raksturojumus. Klases īpašības definē datu tipus, kurus var uzglabāt objekts, bet klases metodes nosaka objekta iespējamās darbības (kāādā veidā objekts piekļūst un apstrādā datus).

Objekts ir klases eksemplārs. Klasi var nosaukt par šablonu, bet objektu par šī šablona konkrēto realizāciju. Piemēram, klase ir tipveida māja, bet objekts ir māja, kur dzīvojat jūs.

Klasēm ir šādas īpatnības:

- klase definē objekta raksturojumu;
- šīs raksturojums nosaka objektu mijiedarbību savā starpā;
- .NET Framework vidē klase ir lietotāja definētais tips, kurš var saturēt skaitliskus un referenču tipus, metodes, atribūtus un īpašības.

Savukārt, objekta īpatnības ir:

- objekts ir klases eksemplārs;
- objekts tiek izveidots programmas izpildes laikā;
- izveidojot objektu tiek izveidots jaunais referenču tips;
- Visual C# programmēšanas valodā viss darbojas līdzīgi objektam;
- katrs objekts izmanto savu atmiņas apgabalu un darbojas neatkarīgi no citiem šīs klases eksemplāriem.

Vēl viena Visual Studio īpatnība ir **vārdvietas** (*namespace*), kuras organizē klases loģiskajā hierarhijā. Vārdvietas ir gan sistēmas iekšējas organizācijas rīks, gan arī ārējais līdzeklis vienādu nosaukumu iespējamo nesaskaņu novēršanai dažādu projektu līmeņos.

.NET Framework klašu bibliotēka sastāv no vārdvietām, katra no kurām satur dažādus izmantošanai pieejamus tipus: klases, struktūras, sakārtotās secības (*enumerations*), delegātus (*delegates*) un saskarnes (*interfaces*). Viena no svarīgākām .NET Framework vārdvietām ir System, kura satur biežāk izmantojamās klases, piemēram:

- **System.Windows.Forms** – satur klases Windows aplikāciju veidošanai;
- **System.IO** – satur klases datu nolasīšanai un ierakstīšanai datnēs;
- **System.Data** – klases, kuras nodrošina pieeju datu bāzes datiem;
- **System.Web** – klases tīmekļa aplikāciju veidošanai.

Vārdvietas iedalās divās grupās: lietotāja definētas un sistēmas definētas. Tiek rekomendēts visas lietotāja definētas klases izvietot vārdvietās. Pēc noklusējuma Visual Studio izveido katram projektam jaunu vārdvietu, kuras nosaukums sakrīt ar projekta nosaukumu.

.NET Framework ļauj izmantot strukturēto izņēmumu situāciju apstrādi, kura iekļauj sevī pašas izņēmuma situācijas, aizsargātus koda blokus un filtrus šo situāciju apstrādei.

Izņēmuma situācijas notiek kādās kļūdas vai negaidītās programmas uzvedības gadījumu dēļ. Piemēram, dalīšana ar nulli vai neeksistējošas datnes atvēršana ir izņēmuma situācijas.

Jebkurā lietotnē jāparedz visas iespējamās izņēmuma situācijas, nodrošinot tās apstrādi programmas kodā. Šai apstrādei paredzēti **try ... catch** un **finally** koda bloki, kuri ļauj noteikt izņēmuma situāciju un attiecīgi reaģēt uz to (ieskaitot jēdzīgu informācijas paziņojumu nodošanu lietotājam).

3.2. Programmas struktūra un izpildes secība

Visual Studio risinājumā var būt viens vai vairāki projekti, katrā projektā var būt viena vai vairākas asamblejas (*assembly*). Katra asambleja tiek kompilēta no vienas vai vairākām izejas datnēm (*source file*). Izejas datne satur

klašu, struktūru, moduļu un saskarņu definīcijas un implementācijas. Kopumā .NET Framework aplikācijas sastāv no šādiem hierarhijas līmeņiem:

- datņu līmenis;
- vārdvietu līmenis;
- moduļu līmenis;
- procedūras līmenis.

Datņu līmenim izveidojot projektu tiek ģenerēts sākotnējais izejas kods. Jebkurš lietotāja definētais priekšraksts drīkst sākties tikai ar:

using - priekšraksts, kuru izmanto klašu un citu tipu importēšanai no citiem projektiem;

namespace – paredzēts koda organizēšanai vārdvietās.

Vārdvietas elementi var būt klases, struktūras vai saskarnes definīcijas. Šī līmeņa datu elementi ir sakārtotās secības un delegāti.

Moduļa līmenī izmanto procedūru, operatoru, īpašību un notikumu izpildāmā kodā saglabāšanai. Datu elementi šeit ir mainīgie, konstantes, sakārtotās secības un delegāti.

Procedūras līmeņa elementi ir izpildāmie priekšraksti. Datu elementi šeit aprobežojas ar mainīgiem un konstantēm. Programmas ieejas punkts ir metode **Main**, kuras satur objektu definīcijas un metožu izsaukumus. Main metode ir **statiskā** (*static*), tas pārvērš šo metodi par klases līmeņa locekli, kas savukārt neprasa objekta definēšanu Main metodes izmantošanai. Šī metode var būt tikai ar **int** datu tipa vērtības atgriešanu vai vispār bez atriežamās vērtības. Par šīs metodes parametriem var būt **string** datu tipa masīvs.

Visual C# gadījumā galvenās metodes definīcija izskatās šādi:

```
static int Main(object[] args)
{
    //metodes kods
    return 0;
}
```

Metodes C# tiek realizētas **funkciju** (kad tās atgriež kādas vērtības) vai **procedūru** (neatgriež vērtības) veidā.

Apskatīsim divu metožu paraugus (ar un bez atriežamām vērtībām):

```
private int SkKvadrata(int sk)
{
    return sk*sk;
}

private void PasakiSveiks()
{
    MessageBox.Show("Sveiks!");
}
```

Pirmā metode par ieejas parametru paņem veselo skaitli un atgriež šī skaitļa kvadrātu, bet otrā ir bez parametriem un izvada paziņojumu „Sveiks!”.

Papildus īpašību logam, **objekta īpašības** ir iespējams mainīt arī programmas koda tekstā. Vispārīgā veidā to piešķir šādi:

```
objektaNosaukums.objektaĪpašība = vērtība;
```

Piemēram, teksta laukuma īpašības izmaiņu kodā pieraksta šādi:

```
textBox1.Text = "Sveiks!";
```

Metodes izsaukšanai ļoti svarīgi ir precīzi definēt tās ievada parametrus, ieskaitot atbilstošās vērtības datu tipu, parametru skaitu, tās secību un tipu. Visual C# programmēšanas valoda ir reģistru atkarīga, tāpēc metodes nosaukumu (tāpat kā jebkuru citu) jāraksta ievērojot burtu reģistrus. Metodes nosaukumu un parametrus sauc par metodes parakstu (*method signature*).

Piemēram, objekta **MessageBox** metodes **Show** izsaukšana ir šāda:

```
MessageBox.Show("Hello");
```


Noklusēto **notikumu apstrādātāju** var izveidot formas dizaina skatā divreiz noklikšķinot ar peles kreiso taustiņu uz attiecīgas vadīklas.

Piemēram, pogas nospiešanas notikumu apstrādātājs izskatās šādi:

```
private void button1_Click(object sender, EventArgs e)
{
    //apstrādātāja kods
}
```

Notikuma apstrādātāja **parametri** sniedz informāciju šī apstrādātāja kontrolei. Pirmais parametrs **sender** norāda uz objektu, kurš izsauca notikumu (šajā gadījumā poga). Otrais parametrs **e** ir objekts, kurš satur konkrētā notikuma specifisko informāciju. Viens no šī parametra izmantošanas piemēriem var būt notikuma atcelšana (**e.Cancel = true**).

Lietotāja definēta notikuma (kurš nav noteikts pēc noklusējuma) izveidošanai jāizpilda šāda soļu secība:

1. Dizaina skatā atlasa vadīklu, kurai grib definēt notikuma apstrādātāju.
2. Īpašību logā izvēlas **Events**  režīmu.
3. Parādīsies notikumu saraksts, no kura atlasa nepieciešamo un apstiprina ar **Enter** pogas nospiešanu.

Bieži vien ir situācijas, kad nepieciešams apstrādāt vairākus notikumus uzreiz. Šajā gadījumā definē apstrādātāju vienai vadīklai, tad atlasa nākošo

vadīklu un īpašību logā atlasa tikko definēto notikumu. Notikumu apstrādātājs būs spēkā abām vadīklām.

Izveidojot jaunu projektu pamata **references uz asamblejām** jau ir pievienotas. Papildus referenču pievienošana notiek šādi:

1. Risinājumu pārvaldniekā ar labo peles taustiņu nospiež uz projekta nosaukuma un izvēlas komandu **Add Reference**.
2. Izvēlas sadaļu ar nepieciešamas references tipu un divreiz ieklikšķina virs konkrētas references.

Konkrētās klases izmantošanai no citas projektam nepiederošās vārdvietas nosaukumā jāizmanto pilns vārdvietu hierarhijas ceļš, kas padara kodu par grūti lasāmo (piemēram, **System.Data.SqlClient.SqlCommand**). Šo trūkumu var novērst izmantojot atsauces uz šo vārdvietu projekta datnes līmenī (izejas koda sākumā):

```
using System.Data.SqlClient;
```

Iekļaujot šādu atsauci programmas kodā var izmantot īso nosaukumu (**SqlCommand**).

3.3. Programmas koda noformēšana („labās programmēšanas stils”)

Objektu izveidošanas laikā tam jāpiešķir nosaukums. Par nosaukumu var būt praktiski jebkurš identifikators, kurš atbilst šādiem **nosacījumiem**:

1. Nosaukums drīkst sākties tikai ar burtu vai pasvītrojuma simbolu.
2. Nosaukumā var būt tikai burti, cipari un pasvītrojumi.
3. Ja nosaukums sākas ar pasvītrojumu, tam jāsaturs vismaz vienu burtu vai ciparu.
4. Nosaukums nedrīkst sakrist ar programmēšanas valodas definētiem vārdiem.

Pastāv vispārīgās rekomendācijas programmas izejas koda noformēšanai, kurus sauc par „labās programmēšanas stilu”. Viena no tādām rekomendācijām iesaka nosaukumu definēšanā pieturēties pie vienotās stratēģijas, kura iekļauj sevī trīs ieteikumus:

1. **Simbolu reģistrs: PascalCasing** – katru nosaukuma vārdu sākt ar lielo burtu, piemēram, izmantot **OpenButton** nevis **Openbutton** vai **Open_Button**; **camelCasing** – katrs nosaukuma vārds sākas ar lielo burtu izņemot pirmo, piemēram, **openFileDialog**.
2. **Mehānika**: objektu nosaukumos izmantot lietvārdus, metožu – darbības vārdus un tā tālāk.
3. **Vārdu izvēle**: mērķtiecīgi pieturēties jēdzienu (nosaukumu) izvēlei visa programmas koda ietvaros.

Labāku koda lasāmību var panākt ar tās **formatēšanu**, kura šajā gadījumā izpaužas atkāpju izmantošanā, rindu sadalīšanā, apvienošanā (konkatenācija) un `StringBuilder` objekta lietošanā.

Atsevišķus koda segmentus tiek rekomendēts izcelt ar atstarpi un papildus tukšām rindām.

Piemēram, konkrētajā piemērā ar atkāpēm ir izdalīts pogas notikuma apstrādātāja kods, kā arī ciklā izpildāmais kods:

```
private void button1_Click(object sender, EventArgs e)
{
    int k = int.Parse(textBox1.Text);
    for (int i = 1; i < 5; i++)
        k = k + i;
    MessageBox.Show("Jūsu skaitlis ir" + k.ToString());
}
```

Pārāk garu koda rindas gadījumos, tos iesaka sadalīt vairākās rindās, lai programmētājam, kas lasīs šo kodu nevajadzētu izmantot horizontālās šķirstīšanas pogas. C# valodā to var veikt praktiski jebkurā vietā, izņemot pēdējās iekļauto tekstu (piešķirot `string` datu tipa vērtības).

```
System.Data.SqlClient.SqlCommand komanda =
    new System.Data.SqlClient.SqlCommand();
```

Garas `string` teksta virknes var pārnest uz vairākām rindām izmantojot konkatenācijas operāciju (+).

```
MessageBox.Show("Jūsu paroles darbības "+
    "laiks beidzās! Lūdzu, ievadiet "+
    "jauno paroli divas reizes");
```

Izveidojot `string` tipa simbolu virknes un saglabājot tās atmiņā, turpmāk simbolu virknei izdalītais atmiņas apjoms nevar mainīties. Ja simbolu virkne tiek mainīta, ir nepieciešama papildus atmiņas telpas izdalīšana citur.

Piemēram, tiek definēts mainīgais `cena` (`int` datu tips) un mainīgais `pazinojums` (`string` datu tips):

```
int cena = 50;
string pazinojums = "Cena ir ";
pazinojums = string.Concat(pazinojums, cena.ToString());
//paziņojumam tiek pievienota mainīgā cena vērtība
```

Šajā piemērā nekādu nepatīksanu nebūs, tomēr, pielietojot tādu pašu komandu ciklā, tam izpildoties katru reizi mūsu simbolu virknei tiks izdalīta atmiņa. Garu ciklu gadījumos tas var izsaukt programmas stirpu ātrdarbības palēninājumu sakarā ar atmiņas trūkumu. Te rekomendē izmantot `StringBuilder` objektu,



kurš ļauj mainīt simbolu virkni automātiski palielinot tās izmēru un bez jauna virknes eksemplāra izveidošanas:

```
int cena = 50;
StringBuilder sb = new StringBuilder("Cena ir ");
sb.Append(cena);
```

„Labās programmēšanas stils” prasa arī īso **komentāru pieraktu** procedūras sākumā, kurš apraksta tās funkcionalitāti. Visual C# programmēšanas valodā pastāv divu veidu komentāri:

```
//tas ir vienas rindas komentārs

/*šīs ir
vairāku rindu
komentārs*/
```

Komentāri tiek izmantoti tikai informatīvā nolūkā. Kompilators pārveidojot izejas kodu rindas ar komentāriem ignorē. Automātiskai komentāru pievienošanai vai noņemšanai izmanto rīkjoslas **Text Editor** pogas pievienot komentārus  un noņemt komentārus .

Pastāv šādi ieteikumi komentāru pielietošanai programmas kodā:

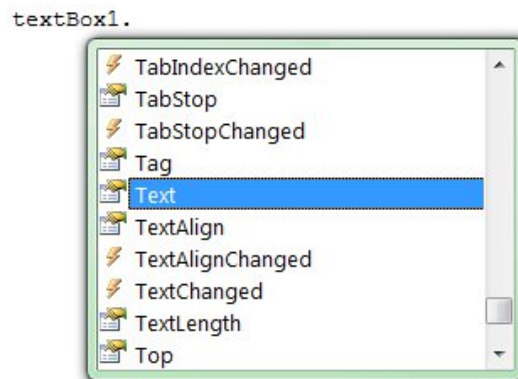
- pirms procedūras (funkcijas utt.) definēšanas, pievienojiet procedūras komentāru, kurš apraksta procedūras nolūku, atgriežamās vērtības un parametrus;
- garu procedūru gadījumos pievienojiet komentārus katra funkcionāli atšķirīgā bloka sākumā;
- pirms mainīgo definēšanas norādiet komentāru ar šī mainīgā izmantošanas mērķi;
- definējot nosacījuma struktūras, aprakstiet katru nosacījuma iznākumu.

Visual Studio 2005 ļauj pievienot komentārus, uz kuru pamata pēc tam tiks ģenerēta **XML formāta** datne. Šajā gadījumā komentārs sākās ar /// simboliem, pēc kuriem pievieno XML tagus.

```
///<summary> Šī klase paredzēta pircēju apstrādei </summary>
class Pirceji
{
    ///<remarks>šeit būs pircēju dati
    ///un tad metodes šo datu apstrādei</remarks>
}
```

XML datnes ģenerēšanai projekta īpašībās šo opciju jāieslēdz. Risinājumu pārvaldniekā uz projekta nosaukuma nospiež ar peles labo taustiņu un izvēlas **Properties**, tad sadaļu **Build** un atzīmē **XML documentation file** režīmu. Komentāru ģenerēšanai var izmantot dažādus XML tagus, tai skaitā arī lietotāja definētus.

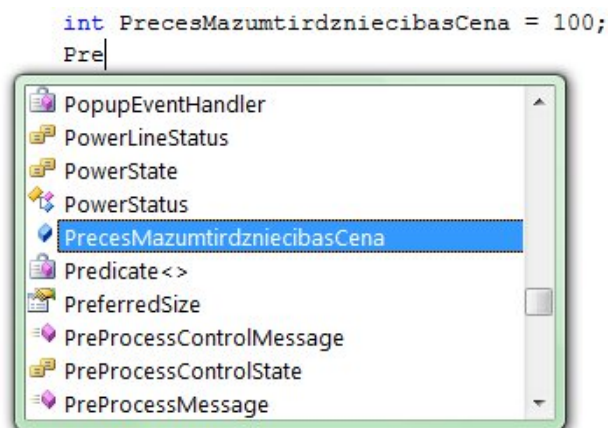
Visual Studio piedāvā dažādas iespējas programmas koda ievades atvieglošanai, kuras kopumā saucās **IntelliSense** īpašības. Viena no tādām īpašībām ir **List Members** vai visu iespējamo pierakstu variantu attēlojums saraksta veidā, no kura var izvēlēties nepieciešamo ar pogas Enter nospiešanu:



Nākošā īpašība ir **Quick Info**, kura ļauj iegūt pilno definēšanas aprakstu katram koda objektam:

```
cena = 30;
(local variable) int cena
```

Complete Word īpašība ļauj automātiski pabeigt kāda identifikatora ievadu pēc tās pirmo burtu ieraksta, kas atvieglo garu objektu nosaukumu izmantošanu:



Vēl viena īpašība ir **Code Snippets** vai gatavas koda sagataves izmantošana, kura ļauj ātri izveidot iepriekš definētās programmēšanas struktūras šablonu, piemēram, ciklu:

```
for (int i = 0; i < length; i++)
{
}
}
```

Nodaļas kopsavilkums

Windows vai tīmekļa aplikācijas izveidošanas laikā tiek izveidota uz notikumiem bāzētā lietotne. Šāda tipa lietotnes izpilda kodu reaģējot uz dažāda veida notikumiem, kurus inicializē lietotājs vai izpildāmais kods. Katrai vadīklai ir sava specifiskā notikumu kopa.

Programmēšanas valoda C# ir objektorientētā, tādēļ tās pamata vienības ir klases un objekti. Klase ir fundamentālais lietotāja definētais tips, kurš organizē informāciju noteiktā veidā. Objekts ir klases eksemplārs.

Vārdvietas organizē klases loģiskajā hierarhijā. .NET Framework klašu bibliotēka sastāv no vārdvietām, katra no kurām satur dažādus izmantošanai pieejamus tipus: klases, struktūras, sakārtotās secības, delegātus un saskarnes.

.NET Framework ļauj izmantot strukturēto izņēmumu situāciju apstrādi, kura iekļauj sevī pašas izņēmuma situācijas, aizsargātus koda blokus un filtrus šo situāciju apstrādei. Izņēmuma situācijas notiek kādās kļūdas vai negaidītās programmas uzvedības gadījumu dēļ.

.NET Framework aplikācijas sastāv no šādiem hierarhijas līmeņiem: datņu līmenis; vārdvietu līmenis; moduļu līmenis un procedūras līmenis.

„Labās programmēšanas stils” iekļauj sevī rekomendācijas programmas izejas koda noformēšanai, kura uzlabo koda lasāmību.

UZDEVUMI PAŠPĀRBAUDEI

1. Raksturojiet notikumu bāzēto programmēšanu.
2. Kādas ir .NET Framework aplikācijas galvenās sastāvdaļas?
3. Nosauciet „labās programmēšanas stila” rekomendācijas.

UZDEVUMI PATSTĀVĪGAJĀM DARBAM

1. Izveidojiet Windows aplikāciju, kur uz galvenās formas ir divas pogas (Button). Pirmās pogas nospiešanas gadījumā mainās formas virsraksts uz „Mana forma”. Otrās pogas nospiešanas gadījumā tiek izvadīts paziņojumu logs ar Jūsu vārdu un uzvārdu un mainās pirmās pogas krāsa uz sarkano.
2. Noformējiet pirmā uzdevumā uzrakstīto kodu pēc „labā programmēšanas stila”. Kādas IntelliSense īpašības un kādā veidā Jūs izmantojat programmēšanas procesā?