Izuzeci, validacija, tabelarni prikaz, njegovo ažuriranje, čuvanje podataka u datoteci

Nakon priče o osnovama **WPF**-a, ovde će ona biti nadograđena dodatnim konceptima.

U odnosu na prvu nedelju vežbi, ovde je zadatak od prošlog puta smešten u prozor ***AddWindow***, dok će glavni prozor služiti za prikaz svih kreiranih objekata klase *Student* u okviru kontrole ***DataGrid.***

Kao što je već poznato, izuzeci (*Exception*) predstavljaju indikaciju grešaka u izvršavanju programa. Posmatrajući domen interakcije čoveka i računara, izuzeci igraju jako važnu ulogu, jer je potrebo korisniku na adekvatan način skrenuti pažnju da je napravio grešku i pomoći mu da je otkloni.

Da bi se proverilo da li je korisnik napravio grešku u popunjavanju polja na formi, implementira se metoda *Validate()* čija je povratna vrednost tipa *bool* i zadatak joj je da nakon korisnikovog pritiska na dugme za dodavanje proveri da li su sva polja adekvatno popunjena.

Da bi se ispratilo da li je korisnik adekvatno popunio polja, potrebno je u metodu dodati jednu promenljivu tipa *bool*, koja će biti indikator da li je bilo koje od polja forme popunjeno na nevalidan način.

Pošto u promenljivoj tipa *string* može da stoji gotovo bilo kakav tekst, za polja tipa ***TextBox*** će biti dovoljno da nisu ostala prazna. Time se sprečava ishod da korisnik nije popunio dato polje. Kao što je predstavljeno u prvoj nedelji vežbi, sadržaju unesenom u ***TextBox*** se pristupa preko njegovog svojstva **Text**. Provera tekstualnog polja je prikazana u **Listingu 1**.

```
if (textBoxIme.Text.Trim().Equals(""))
{
      result = false;
      textBoxIme.BorderBrush = Brushes.Red;
      textBoxIme.BorderThickness = new Thickness(1);
      labelImeGreska.Content = "Ne moze biti prazno!";
}
else
{
      textBoxIme.BorderBrush = Brushes.Green;
      labelImeGreska.Content = string.Empty;
}
```

Listing 1. Provera da li je polje za unos imena prazno

Ako metoda *Validate()* kao povratnu vrednost vrati *false*, znači da je korisnik napravio grešku u unosu i neće se dozvoliti dodavanje novog studenta u tabelu. Da bi se prikazala poruka o grešci, na prozor sa formom je dodata po jedna ***Label*** kontrola ispod svakog polja i u njoj će se ispisivati poruka o grešci za to konkretno polje. Da bi se ispisala poruka, potrebno je labeli zadati vrednost svojstva **Content**. Definicija labele kada je u pitanju izgled aplikacije, data je u **Listingu 2**.

```
<Label x:Name="labelImeGreska" Content="" HorizontalAlignment="Left" Height="24"
Margin="103,111,0,0" VerticalAlignment="Top" Width="229" Foreground="Red"/>
```

Listing 2. Definicija Label kontrole za ispis greške u XAML delu aplikacije.

Još jedan efikasan način da se korisniku skrene pažnja da je napravio grešku je tako što se promeni boja okvira polja gde je napravljena greška, tako da okvir bude crven. To je realizovano promenom vrednosti svojstava **BorderBrush** (boja konturne linije) i **BorderThickness** (debljina konturne linije). Debiljnu konturne linije je potrebno promeniti jer ona inicijalno nema vrednost, odnosno sama linija se ne vidi.

Ostatak forme će biti identično realizovan, sa tim da se za ***RadioButton*** može izbeći promena tako što se neki od njih u startu postavi kao da je označen (svojstvo **IsChecked** = *true*), tako da nije moguće da, pošto su međusobno isključivi, neki ostane neoznačen. U suprotnom, potrebno je proveriti samo slučaj u kome oba nisu označena, kako je prikazano u **Listingu 3**.

```
if (radioButtonM.IsChecked == false && radioButtonZ.IsChecked == false)
```

Listing 3. Provera da li su oba RadioButton-a neoznačena.

Za ***ComboBox*** je potrebno proveriti da li je njegova izabrana vrednost (svojstvo **SelectedItem**) jednaka *null*, i ako jeste, implementacija je identična kao i kod kontrole ***TextBox***. Međutim, pošto je za ***ComboBox*** nemoguće menjati boju okvira bez detaljnijih izmena stilova izgleda same kontrole, to se može zaobići „smeštanjem“ ***ComboBox***-a u ***Border*** kontrolu, kojoj se onda navodi ime i njoj se iz **C#** koda može menjati boja kako bi se dobio identičan efekat kao da ima okvir koji menja boju. Primer postavljanja ***ComboBox*** kontrole u ***Border*** je dat u **Listingu 4**.

```
<Border x:Name="comboBoxBorder" HorizontalAlignment="Left" Margin="103,256,0,0"
VerticalAlignment="Top" Width="229" Height="28">
      <ComboBox x:Name="comboBoxSmer" Height="28" Width="229"/>
</Border>
```

Listing 4. Dodatak na postojeće rešenje za dodavanje studenata

U slučaju da su uslovi zadovoljeni, potrebno je da se labele ne ispisuju i da se ne boje okviri kontrola. Validacija će se pozivati nakon svakog klika na dugme dodaj, pa će se svaki put dodeljivati promene izgleda ovih kontrola i treba voditi računa kako oni trebaju da se ponašaju kada je sve adekvatno uneseno, odnosno da u tom slučaju ne budu uokvirene crvenom bojom.

Pozivanje metode za validaciju se dešava u reakciji na klik na dugme „Dodaj“, i to je prikazano u **Listingu 5.**

```
if (validate())
{
    //logika za dodavanje je ista kao na primeru sa prve nedelje vežbi
}
else
{
    MessageBox.Show("Podaci nisu dobro popunjeni", "Greska!", MessageBoxButton.OK,
                    MessageBoxImage.Error);
}
```

Listing 5. Dodatak na postojeće rešenje za dodavanje studenata

Kada korisnik klikne na dugme „Dodaj“ i metoda za validaciju kao povratnu vrednost vrati *true*, izvršiće se identičan kod onom sa prve nedelje vežbi. U slučaju da validacija nije dala željeni rezultat, potrebno je napraviti neku vrstu prekida u aplikaciji, kako bi korisnik bio svestan da je napravio grešku. Jedan od načina da se ovo realizuje je pozivanjem „samostojeće“ klase, ***MessageBox***.

Pozivanjem njene metode *Show()*, biće prikazan dijaloški prozor kakav se obično u Windows aplikacijama poziva kada je potrebno preneti korisniku neku informaciju (na primer: Are you sure you want to exit?). Metoda *Show()* se poziva sa najmanje jednim, a može se pozvati i sa 4 parametara.

Ovi parametri određuju tekst koji će pisati u dijaloškom prozoru (**messageBoxText**), tekst koji piše u naslovnoj liniji prozora (**caption**), izbor koja će dugmad biti prikazana na prozoru (**MessageBoxButton**, ako se ovaj parametar ne definiše, biće samo dugme „OK“) i koja će ikonica biti prikazana na prozoru (**MessageBoxImage**, ako se ovaj parametar ne definiše, nikakva ikonica neće biti prikazana). Jedan primer poziva ***MessageBox***-a dat je u **Listingu 5**.

DataGrid kontrola predstavlja tabelarni prikaz objekata određenog tipa. Da bi se odredilo koji objekti će biti prikazivani u njemu, potrebno je da postoji lista objekata datog tipa i da se ona dodeli kao vrednost svojstvu **ItemsSource** ***DataGrid***-a.

Data Binding predstavlja proces koji formira vezu između interfejsa aplikacije i podataka koji se prikazuju. Kada podaci promene vrednost, elementi interfejsa koji su povezani sa tim podacima će se automatski ažurirati.

Kada se instancira, kontrola ***DataGrid*** ne sprečava promenu vrednosti svojstava objekata koji se prikazuju izmenom teksta upisanog u polja same tabele, pa je potrebno definisati svojstvo ***IsReadOnly*** na vrednost *true* (pošto bi tabela trebala da služi za prikazivanje podataka). Definicija ***DataGrid***-a je prikazana u **Listingu 6**.

```
<DataGrid x:Name="dataGridStudenti" ItemsSource="{Binding Studenti}" IsReadOnly="True"
AutoGenerateColumns="False" HorizontalAlignment="Left" Height="158" Margin="10,47,0,0"
VerticalAlignment="Top" Width="322">
      <DataGrid.Columns>
            <DataGridTextColumn Header="Ime" Binding="{Binding Ime}"/>
            <DataGridTextColumn Header="Prezime" Binding="{Binding Prezime}"/>
            <DataGridTextColumn Header="Pol" Binding="{Binding Pol}"/>
            <DataGridTextColumn Header="Smer" Binding="{Binding Smer}" Width="*"/>
      </DataGrid.Columns>
</DataGrid>
```

Listing 6. Definicija DataGrid-a u XAML delu aplikacije

Da bi se uveo **Data Binding**, vrednost svojstva *ItemsSource* se dodeljuje tako da se ona definiše kroz ključnu reč ***Binding***, i uz nju ime liste koja je izvor podataka za ***DataGrid*** kontrolu. Zbog načina realizacije **Data Binding**-a, kolone u tabeli se neće automatski generisati (svojstvo *AutoGenerateColumns* dobija vrednost *false*), već je potrebno definisati same kolone i podatke u njima.

Kolone ***DataGrid***-a su svojstvo kao i svako drugo, tako da se i njima može pristupiti tako što se formira tag *<DataGrid.Columns>* koji, na osnovu priče iz prve nedelje vežbi, slično kao kontejnerska kontrola ***Grid***, zahteva da se unutar nje definišu entiteti koje sadrži. Ti entiteti su tipa *DataGridTextColumn* i predstavljaju kolone u čijim će se poljima prikazivati podaci tekstualnog tipa. *Header* kolone predstavlja koje će biti njeno ime, odnosno šta će pisati u njenom zaglavlju.

Vrednost se zadaje kroz rezervisanu reč *Binding* i predstavlja koji će se podaci prikazivati u njoj. **DataBinding-om se sa dizajnom mogu povezivati samo promenljive definisane kao Property, nikako polja klase, pa se zbog toga i Binding lista koja se koristi za Data Binding mora definisati kao Property**. Potrebu da se poslednja kolona produži do kraja tabele (kako ne bi ostala prazna kolona koja popunjava ostatak tabele), realizovana je definisanjem širine date kolone preko svojstva *Width*, koji dobija vrednost „*“, koja proširuje kolonu tako da popuni ceo prostor do kraja širine tabele.

Postoji nekoliko tazličitih tipova kolona koje se mogu dodati u tabelu:

- *DataGridHyperlinkColumn* (u poljima kolone se nalazi link do neke datoteke ili web-stranice),
- *DataGridCheckBoxColumn* (u poljima kolone se nalazi ***CheckBox***, koji prikazuje vrednosti *true*, odnosno *false*),
- *DataGridComboBoxColumn* (u poljima kolone se nalazi ***ComboBox***, odnosno padajuća lista vrednosti) i
- *DataGridTemplateColumn* (sadržaj polja kolone je kako korisnik definiše - rezervisana reč *Template* u **WPF**-u predstavlja nešto što korisnik definiše kako će izgledati)

Da bi lista podataka mogla da se poveže sa ***DataGrid*** kontrolom preko **Data Binding**-a, ona mora biti definisana kao *BindingList*, tip koji funkcioniše na isti način kao i obična lista, samo se koristi za primenu **Data Binding**-a.

Najvažniji deo implementacije **Data Binding**-a u okviru WPF projekta je definicija *DataContext*-a. *DataContext* predstavlja „okidač" **Data Binding**-a i indikaciju u kojoj se klasi nalaze podaci koji će biti spojeni sa interfejsom aplikacije putem **Data Binding**-a. U ovom slučaju, Binding lista „Studenti" se nalazi u klasi „MainWindow.xaml.cs" i iz tog razloga će *DataContext*-u biti dodeljena referenca samu sebe (*this*). Sve ovo se dešava pre metode *InitializeComponent()*, jer ona izgrađuje sam interfejs.

Da bi se omogućio rad sa datotekama, u projektu postoji klasa „DataIO". Njena uloga je da ponudi mogućnost upisivanja podataka u *XML* datoteku i iščitavanje podataka iz iste. Njena implementacija neće biti detaljno objašnjavana, već se predstavlja način kako se koristi. Najpre, u klasi gde je potrebno realizovati funkcije za učitavanje i upisivanje podataka u *XML* datoteku je potrebno napraviti novu instancu objekta klase „DataIO". Nakon toga, moguće je pristupiti njenim metodama za čuvanje i iščitavanje podataka iz datoteke.

Metoda *DeserializeObject<>()*, će iščitati podatke tipa koji je naveden između znakova „<" i „>" iz datoteke čije je ime navedeno kao jedini parametar same metode. Korišćenje ove metode prikazano je u **Listingu 7**.

```
Studenti = serializer.DeSerializeObject<BindingList<Student>>("studenti.xml");
if(Studenti == null)
{
   Studenti = new BindingList<Student>();
}
```

Listing 7. Poziv metode DeserializeObject<>() da bi se iščitali podaci iz XML datoteke

Nakon iščitavanja liste iz datoteke, potrebno je proveriti da li datoteka slučajno nije prazna, jer u tom slučaju metoda *DeserializeObject<>()* kao povratnu vrednost vraća *null* i ako lista u koju se rezultat dodeljuje ostane sa nedefinisanom vrednošću, instancira se kao nova, jer u suprotnom neće postojati kolekcija u koju se dodaju novi studenti i rezultati dodavanja neće biti prikazani u tabeli.

Da bi se neki objekat zapisao u datoteku pomoću klase „DataIO", potrebno je pozvati njenu metodu *SerializeObject<>()*, gde se, opciono, tip podataka koji se zapisuje u datoteku navodi između znakova „<" i „>" i kao parametri se navode koji se to objekat upisuje u datoteku i ime datoteke u koju se upisuje (ako datoteka ne postoji, biće kreirana). **Da bi se formirala XML datoteka, klasa čiji se objekti čuvaju u njoj mora imati dodat atribut *[Serializable]* i prazan konstruktor.**

Pošto se iščitavanje iz datoteke dešava u kontruktoru glavnog prozora (pri pokretanju aplikacije), čuvanje podataka u datoteci će biti realizovano pri izlasku iz aplikacije. U dizajnerskom delu aplikacije je potrebno u okviru taga ***<Window>***, definisati *Closing* event, koji će se pozivati prilikom izlaska iz

aplikacije. Ime dodeljeno događaju predstavlja funkciju kojom će se reagovati na izlazak iz aplikacije. Kada joj se da ime i ona bude generisana u C# kodu, u okviru nje se poziva *SerializeObject<>()*, kako je prikazano u **Listingu 8**.

```
private void save(object sender, CancelEventArgs e)
{
     serializer.SerializeObject<BindingList<Student>>(Studenti, "studenti.xml");
}
```

Listing 8. Serijalizacija Binding liste u XML datoteku

Sve ove funkcionalnosti su iskorišćene da se kreira primer za drugu nedelju vežbi koji prikazuje tabelu svih studenata, gde se može pozvati forma za dodavanje studenata, gde kada se unesu podaci određenog studenta, oni budu prikazani u tabeli na glavnom prozoru. Za sva polja forme za dodavanje realizovana je provera da li su adekvatno popunjena. Tabela je realizovana pomoću Data Binding-a, a podaci u okviru nje se prilikom izlaza iz programa beleže u XML datoteku.