POGLAVLJE 2

.NET brojni i vrednosni tipovi

U prethodnom poglavlju ste naučili kako da koristite Visual C# Express integrisano razvojno okruženje, odnosno .NET pomoću CIL i CLR. U o vom poglavlju ćemo zavrnuti rukave i pristupiti pisanju C# koda, konkretno, koda kojim se kreira kalkulator.

Kalkulator je idealan primer za početak, pošto vam omogućava da se fokusirate na aplikaciju, a da ne morate da posvećujete značajnu pažnju na detalje koji su obično povezani sa programima. U programskom jeziku kakav je C#, sabiranje dva broja je veoma jednostavna operacija. Ono što je malo složenije je postupak dodavanja operacije sabiranja dva broja u program.

U ovom poglavlju glavna pažnja je posvećena postupku pisanja programa korišćenjem programskog jezika C#, odnosno postupku prevođenja odgovarajuće ideje u C# program koji može da izvršava ono što ste zamislili. Naučićete na koji način možete da organizujete postupak razvoja programa i implementiranja C# biblioteke klasa, odnosno kako .NET CLR upravlja brojnim tipovima.

Podešavanje i organizovanje okruženja

Prilikom razvoja softvera, svoj rad ćete podeliti u dva glavna zadatka: organizovanje i implementiranje. Organizovanje projekta podrazumeva identifikovanje elemenata i biblioteka koje je neophodno da definišete, broja ljudi koji će učestvovati u razvoju, i slično.

Organizovanje razvoja je jedan od najznačajnijih zadataka prilikom pisanja koda, a to je obično nešto što najviše zbunjuje nove programere. Profesionalni programeri se ponašaju tako da izgleda da svoj posao instinktivno organizuju, ali to samo deluje tako, jer su iste aktivnosti obavljali veliki broj puta, tako da one postaju automatske.

Kada su programeri vezani za kreiranje programa, od njih se zahteva da kreiraju softver koji implementira određeni skup funkcionalnosti. Funkcionalnosti podrazumevaju izračunavanje dnevnih interesnih uplata, automatsko generisanje pisama koja ukazuju na prihvatanje ili odbijanja zahteva za pozajmicom, i tako dalje. Funkcionalnost je uvek povezana sa izvršavanjem nekog zadatka koji je definisan određenim procesom. Možemo da kažemo da je implementacija funkcionalnosti direktna implementacija određenog zadatka.

Prilikom definisanja funkcionalnosti neophodno je da uradite sledeće:

- Potpuno razumevanje funkcionalnosti. Ne možete da implementirate nešto što ne razumete u potpunosti. Da biste mogli da pišete izvorni kod programa koji omogućava implementiranje određene funkcionalnosti, neophodno je da u potpunosti razumete sve ono što je vezano za funkcionalnost.
- Opisivanje funkcionalnosti korišćenjem metoda strukturnog projektovanja. Jednostavno organizovanje misli može da bude dovoljno ukoliko ste jedina osoba koja radi na razvoju programa; međutim, veoma često, bićete deo tima. Neophodno je da primenjujete metod strukturnog projektovanja, kako biste vi i ostali članovi vašeg tima mogli da međusobno komunicirate u toku razvoja programa.

Jedan od standardnih metoda strukturnog projektovanja je Unified Modeling Language (UML). UML se primenjuje za organizovanje elemenata u jedinice koje odgovaraju konstrukcijama programskog jezika, kao što su klase. UML možete da smatrate specifičnim žargonom programera, koji se koristi za opisivanje različitih aspekata programskog okruženja na visokom nivou apstrakcije. UML vam omogućava da steknete predstavu o arhitekturi bez potrebe za pristupanjem izvornom kodu same aplikacije. UML možete smatrati strukturnim slojem, koji se nalazi iznad programiranja softvera.

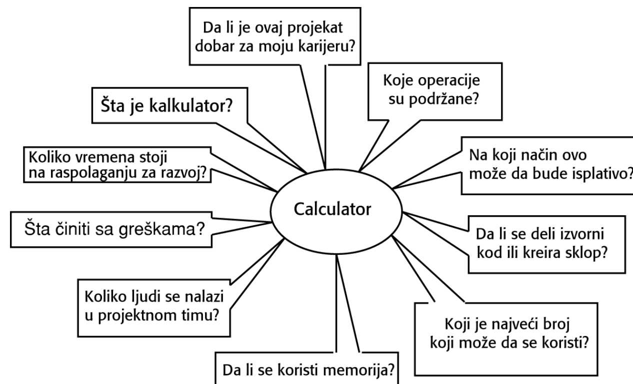
Pored UML-a, u svom projektnom timu možete da primenjujete agilno softversko projektovanje, ili neki drugi metod za strukturno projektovanje softvera. Ideja vezana za agilno projektovanje softvera je korišćenje table i razvoj sopstvenog mehanizma za komuniciranje.

Da li ćete koristiti UML, agilni softverski razvoj ili neki drugi metod strukturnog projektovanja zavisi od vas i članova vašeg tima. Ali, biće neophodno da organizujete način razmišljanja i da primenite određenu strukturnu tehniku komunikacije. Ukoliko to ne uradite, razvoj vašeg softvera će kasniti, softver će imati dosta bagova, biće preskup ili nekompletan. Ne kaže se bez razloga da je dobrom organizacijom bitka skoro dobijena.

U ovom poglavlju primenjuje se pojednostavljena tehnika strukturnog projektovanja, tako da ćete bar imati predstavu o tome kako u praksi funkcioniše strukturno projektovanje.

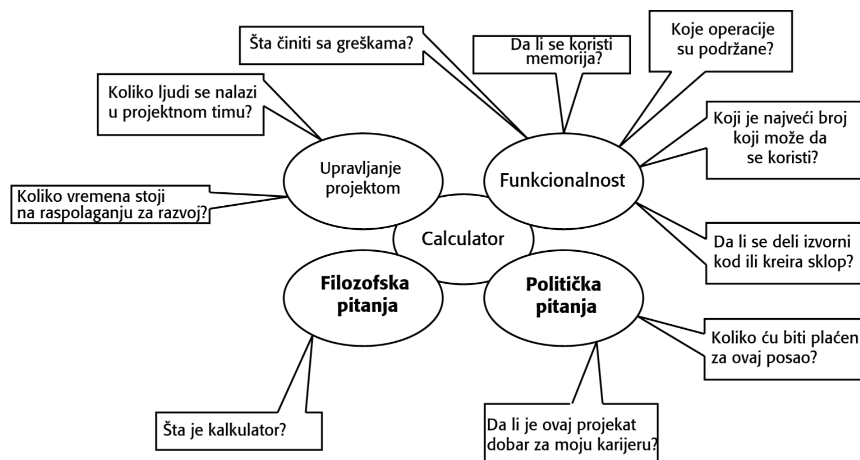
Organizovanje Calculator aplikacije

Pre nego što počnete sa primerom u ovom poglavlju, neophodno je da nabavite papir i olovku, ili možete da koristite tablet PC za pisanje, ukoliko ga imate. Dalje, u centru papira (ili virtuelnog papira), nacrtajte krug i u njemu napišite reč Calculator. Nakon toga se zaustavite i razmislite o tome šta kalkulator znači u odnosu na softver koji želite da kreirate. Napišite osnovne ideje na papiru, u prostoru oko kruga. Naše ideje su prikazane na slici 2.1.



SLIKA 2.1 *Ideje o tome šta treba da predstavlja Calculator aplikacija*

Vaše ideje mogu se neznatno razlikovati od onih koje su prikazane na prethodnoj slici, ali ono što je zajedničko je što sve liči na papazjaniju. Na slici 2.1 prikazano je ono što predstavlja najveći problem sa kojim se susreću ljudi koji učestvuju u razvoju softvera, a to je nedostatak fokusiranja i organizacije. To ne znači da programeri ne mogu da se fokusiraju ili organizuju, već da su bombardovani različitim informacijama, te da je herkulovski zadatak praćenje, organizovanje i lociranje svih neophodnih informacija. Ali, da bi softverski projekat bio uspešan neophodno je da se odlikuje fokusiranošću i organizovanošću. Zbog toga, naredni korak je da fokusirate i organizujete svoje ideje, što će kao rezultat dati nešto slično onom što je prikazano na slici 2.2.



SLIKA 2.2 *Fokusirane i organizovane ideje*

Na slici 2.2 ideje su organizovane tako što su klasifikovane u odgovarajuće grupe. Pošto je ovo knjiga o programskoj jeziku, jedine relevantne ideje vezane su za funkcionalnosti izvornog koda. Uopšteno rečeno, u kategoriji izvornog koda, svaka ideja odgovara određenoj funkcionalnosti koju je neophodno implementirati.

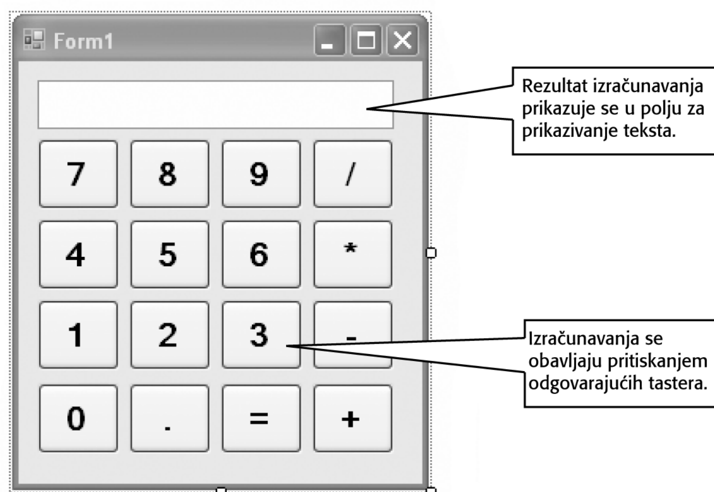
Preciziranje ideja vezanih za Calculator aplikacije

Da biste implementirali odgovarajuću funkcionalnost, neophodno je da napišete izvorni kod programa, što podrazumeva kreiranje datoteke i projekta, odnosno primenu odgovarajućih tehnika programiranja. Pre nego što pristupite implementiranju određenih funkcionalnosti, neophodno je da definišete način organizovanja izvornog koda. Postoje dva nivoa ogranizovanja izvornog koda:

- *Organizovanje na nivo datoteka:* Na ovom nivou, neophodno je da definišete koje ćete tipove projekata i rešenja kreirati.
- *Organizovanje na nivou izvornog koda:* Na nivou izvornog koda organizujete prostore naziva, definišete nazive klasa i ostalih identifikatora koji se primenjuju u okviru izvornog koda.

Implementiranje kalkulatora na nivou datoteka počinje izborom jednog od tri moguća tipa projekata. Kao što je opisano u Poglavlju 1, imate tri mogućnosti: Windows aplikacija, konzolna aplikacija ili biblioteka klasa.

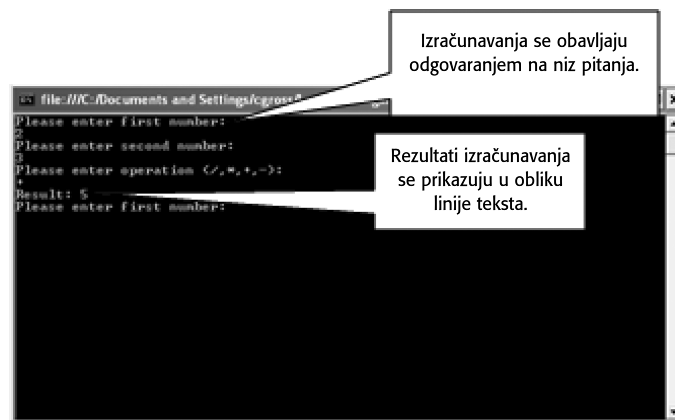
Ukoliko biste kalkulator kreirali kao Windows aplikaciju, ona bi trebalo da izgleda onako kako je prikazano na slici 2.3.



SLIKA 2.3 *Kalkulator implementiran kao Windows aplikacija*

Kalkulator implementiran kao Windows aplikacija omogućava korisnicima da obavljaju izračunavanja pritiskanjem odgovarajućih tastera. Da bi sabrao dva broja, korisnik treba da pritisne odgovarajuće tastere za unos prvog broja, zatim treba da pritisne taster kojim definiše željenu operaciju, nakon toga treba da unese drugi broj i, konačno, da pritisne taster na kome je prikazan znak jednako. Znak jednako je signal aplikaciji da je neophodno da odredi vrednost prethodnog izraza na osnovu unetih podataka, odnosno da generiše rezultat. Nakon toga, neophodno je da se prikaže rezultat u polju za prikazivanje teksta.

Drugi izbor vezan je za implementiranje kalkulatora kao konzolne aplikacije, pri čemu bi se brojevi unosili kao tekst, što je prikazano na slici 2.4.



SLIKA 2.4 *Kalkulator implementiran u obliku konzolne aplikacije*

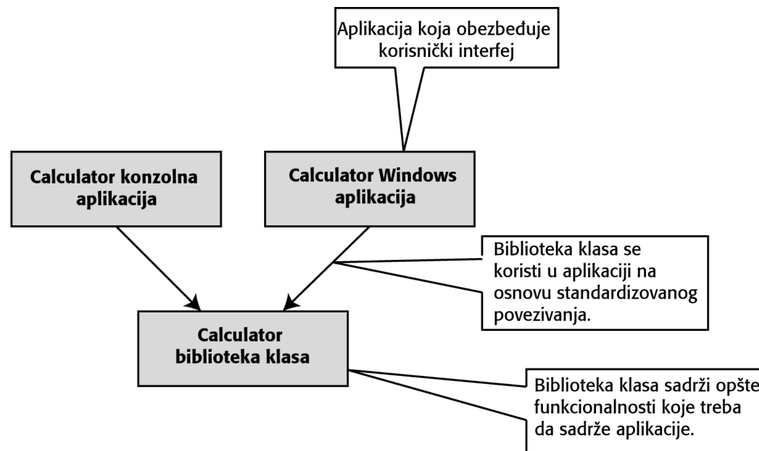
Ukoliko kalkulator implementirate u obliku konzolne aplikacije, korisnici ne moraju da pritiskaju tastere kako bi uneli odgovarajuće operande; umesto toga, oni treba da pritiskaju odgovarajuće tastere na tastaturi kako bi uneli odgovarajuću vrednost u određenom trenutku, odnosno izvršili odgovarajuću operaciju. Tipično, Enter taster se koristi kao taster jednako, odnosno za izračunavanje odgovarajuće operacije, čiji se rezultat prikazuje u komandnom prozoru. Nakon što se izvrši određeno izračunavanje, ponavlja se ciklus izračunavanja.

Interakcija korisnika sa prethodno opisanim tipovima aplikacija je drastično različita, i podrazumeva kreiranje dva različita programa, iako se implementiraju identične funkcionalnosti. Fokus nije na kreiranju konkretnog tipa programa, već na kreiranju celokupne programske strukture.

Ukoliko biste birali između Windows i konzolne aplikacije za kalkulator, verovatno biste izabrali Windows aplikaciju, pošto izgleda bolje, a i jednostavnije je korišćenje takve aplikacije. Na osnovu onoga što je prikazano na slici 2.2, jednostavnost nije navedena kao željena karakteristika aplikacije. Može li tip korisničkog interfejsa da bude jedna od karakteristika aplikacije? Naravno da može, ali u kontekstu ovog poglavlja, to nije od interesa.

Vratimo se jedan korak unazad, i razmotrimo apstraktno ono što je navedeno u prethodnom odeljku. Vi ste programer, i plaćeni ste za implementiranje kalkulatora za oba tipa korisničkog interfejsa. Apstraktno gledano, da li je neophodno da jedne iste funkcionalnosti implementirate dva puta, ili biste mogli da razmislite o tome da neke delove kalkulatora koristite u obe aplikacije, koje se razlikuju u korisničkom interfejsu? Najverovatnije će vaš odgovor biti takav da želite da koristite više puta delove koda kalkulatora, kako biste smanjili neophodan rad. Ali, takođe, veoma je važno da više puta koristite delove tako da izbegnete probleme vezane za dodatno održavanje i proširivanje programa.

Dakle, kada se radi u razvoju softvera, neophodno je da razmišljate o pojedinim elementima koji čine vaš program. Neke elemente možete da koristite više puta, a neke druge elemente ne možete. Zbog toga, neophodno je da Calculator aplikaciju razmatrate iz dva dela: korisnički interfejs aplikacije i deo koji omogućava izvršavanje izračunavanja zasnovanih na podacima koje korisnik unosi pomoću korisničkog interfejsa. Sa stanovišta organizacije, ili programerskim jezikom rečeno, sa stanovišta arhitekture aplikacije, delovi Calculator aplikacije biće organizovani onako kako je prikazano na slici 2.5.



SLIKA 2.5 *Uređenje delova Calculator aplikacije*

Pojedinačni delovi aplikacije, prikazani na slici 2.5, nazivaju se komponente. (Neki pojedinci ove delove nazivaju modulima, ali mi preporučujemo da koristite pojam komponente.) Komponente su uređene tako da su funkcionalnosti nižeg nivoa prikazane u donjem delu slike, a funkcionalnosti višeg nivoa su prikazane u gornjem delu slike.

Svaka komponenta se koristi za izvršavanje odgovarajućeg zadatka, a komponente višeg nivoa primenjuju one zadatke koji su implementirani na nižem nivou. Ideja je da svaki nivo bude odgovoran za određenu funkcionalnost, a da ostali nivoi ne dupliraju funkcionalnosti ponavljanjem onoga što je već prethodno implementirano. Funkcionalnost višeg nivoa zavisi od funkcionalnosti nižeg nivoa, ali funkcionalnost nižeg nivoa ne zavisi od funkcionalnost višeg nivoa.

Aplikacije su realizovane korišćenjem top-down ili bottom-up arhitekture. Top-down metodologija podrazumeva kreiranje komponenti višeg nivoa, a zatim implementiranje komponenti nižeg nivoa, onda kada je to neophodno. Nasuprot tome, bottom-up metodologija podrazumeva kreiranje prvo komponenti nižeg nivoa.

Bottom-up pristup je koristan onda kada jasno znate koje funkcionalnosti treba da implementirate. Top-down pristup je koristan onda kada imate grubu predstavu o tome koje je funkcionalnosti neophodno implementirati, ali ne želite da se previše udaljite od cilja same aplikacije. Cilj ovog poglavlja je kreiranje Calculator biblioteke klasa, koja je prikazana u donjem delu slike 2.5, pa ćemo u ovom poglavlju primenjivati bottom-up pristup.

Implementiranje biblioteke klasa

Kreiranje biblioteke klasa je jedna forma organizovanja datoteka. U narednom koraku kreiraćete izvorni kod za biblioteku klasa. Izvorni kod se implementira u dva koraka:

- Definisanje klase i metoda
- Implementiranje metoda

Jedan od najvćih problema vezanih za učenje novog programskog jezika je utvrđivanje šta programskim jezikom možete da uradite, a šta ne. Ne možete da pišete izvorni kod koji nema smis-

la u odgovarajućem programskom jeziku. Zbog toga je veoma značajno da dobro poznajete programski jezik, pošto ćete na osnovu toga struktuirati svoje ideje.

Kreiraćete dva tipa izvornog koda: izvorni kod koji organizuje i izvorni kod koji izvršava određene operacije. Izvorni kod koji organizuje predstavlja neku vrstu ormana sa fasciklama. Izvorni kod koji izvršava određene operacije predstavlja fasciklu u kojoj se nalazi odgovarajući sadržaj. Prilikom kreiranja ormana, ne vodite računa o sadržaju pojedinačnih fascikli koje će se nalaziti u tom ormanu. Onda kada popunjavate dasciklu, vi ne razmišljate o ormanu.

Klase, prostori naziva i metodi su koncepti koji se koriste za organizovanje izvonog koda. Metod sadrži izvorni kod, a omogućava izvršavanje akcija kao što su sabiranje brojeva ili kreiranje tekstualnog stringa.

Ono o čemu morate da vodite računa prilikom kreiranja izvornog koda metoda je korišćenje drugih delova organizovanog izvornog koda putem referenciranja. Referenciranje prestavlja isto što i lepljiva napomena na kojoj se nalazi "Molimo vas da pogledate sadržaj fascikle B".

Sledeći deo izvornog koda je 100% organizovan, ali se njime ne izvršava nijedna operacija.

```
namespace MyMainTypes {
   static class AType {
       public static void DoSomething() { }
    }
}
namespace MyOtherMainTypes {
   static class AnotherType {
       public static void DoSomething() { }
    }
}
```

Izvorni kod sadrži tri nivoa organizacije. Prostor naziva (`MyMainTypes` i `MyOtherMainTypes` u prethodnom primeru) enkapsulira tipove kao što su klase (`AType` i `AnotherType` u prethodnom primeru). Klase enkapsuliraju metode (`DoSomething` u prethodnom primeru) ili svojstva. U okviru prostora naziva, svi tipovi moraju biti jedinstveni. Vi možete da imate dva tipa sa istim identifikatorom u različitim prostorima naziva. U okviru jednog tipa, ne možete da imate identične indentifikatore sa identičnim parametrima. (Ovo će biti mnogo jasnije onda kada budete naučili nešto više o programskom jeziku C# u toku narednih poglavlja.)

U sledećem primeru prikazana je identična organizacija koda sa odgovarajućim izvornim kodom (koji je prikazan podebljanim slovima).

```
namespace MyMainTypes {
   static class AType {
       public static void DoSomething() { }
    }
}
namespace MyOtherMainTypes {
   static class AnotherType {
       public static void DoSomething() {
           MyMainTypes.AType.DoSomething();
       }
    }
}
```

U kodu koji je označen podebljanim slovima, postoji referenca na drugi prostor naziva, tip i metod sa parom zagrada. To se naziva izvršavanje poziva metoda statičke klase i statičkog metoda. U ovom primeru je implementiranje metoda zapravo pozivanje nekog drugog metoda.

Uočite način na koji je drugi metod referenciran korišćenjem prostora naziva i identifikatora tipa. To je način na koji se referenciraju svi tipovi i metodi. Identifikator prostora naziva je neophodan samo onda kada tip (na primer, klasa) nije definisan u trenutno korišćenom prostoru naziva.

Ukoliko imate previše duge prostore naziva, prethodno opisano referenciranje može biti veoma naporno. Alternativno rešenje je da dodate using naredbu za referenciranje prostora naziva, kao što je prikazano u sledećem primeru.

```
using MyMainTypes;
namespace MyOtherMainTypes {
   static class AnotherType {
       public static void DoSomething() {
           AType.DoSomething();
       }
    }
}
```

Using naredbom se definiše da se, ukoliko se u kodu referencira bilo koji tip koji nije lokalno definisan, analizira prostor naziva (`MyMainTypes` u prethodnom primeru) kako bi se pronašao neophodan tip. Obratite pažnju na to da korišćenje dva prostora naziva, koji sadrže isto nazvane tipove, prilikom referenciranja tih tipova dovode do greške prilikom prevođenja, pošto kompajler ne može da utvrdi koji tip se zapravo referencira.

Ovo predstavlja suštinu pisanja koda, tako da možemo da pristupimo pisanju koda kojim se implementiraju odgovarajuće funkcionalnosti.

Pisanje Add() metoda

Pristupićemo pisanju koda za sabiranje dva broja. Da biste imali mogućnost da pišete kod, neophodno je da kreirate novi Visual C# projekat na sledeći način:

1. Pristupite Visual C# okruženju (ukoliko ste već prethodno pristupili Visual C# okruženju, izaberite File➥Close Solution stavku menija, kako biste zatvorili prethodno rešenje).
2. Izaberite File➥New Project stavku menija ili Create: Project opciju.
3. Izaberite Class Library tip projekta, dodelite mu naziv Calculator, a zatim pritisnite taster OK.
4. Datoteku Class1.cs preimenujte u Calculator.cs.
5. Snimite kreirano rešenje.

Sada ćemo pristupiti kreiranju `Add()` metoda. U Calculator.cs datoteku dodajte kod prikazan podebljanim slovima.

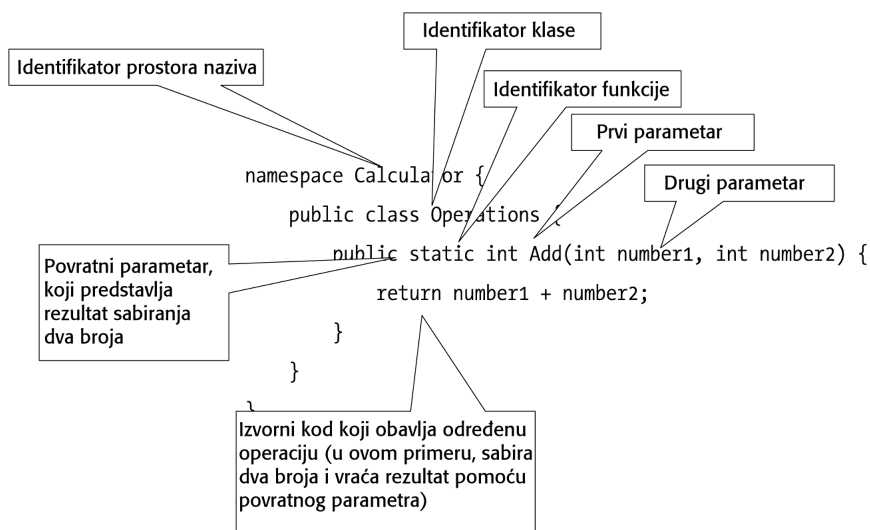
```
using System;
using System.Collections.Generic;
using System.Text;

namespace Calculator
```

```
{
    public class Calculator
    {
    }

    public class Operations
    {
        public static int Add(int number1, int number2)
        {
            return number1 + number2;
        }
    }
}
```

Na slici 2.6 prikazani su različiti delovi `Add()` metoda, tako da možete da vidite koji deo metoda ima koju namenu.



SLIKA 2.6 *Detaljan opis operacije sabiranja*

U izvornom kodu, parametri se koriste za specificiranje ulaznih podataka. Svaki parametar predstavlja jedan od brojeva koji će se koristiti u operaciji sabiranja.

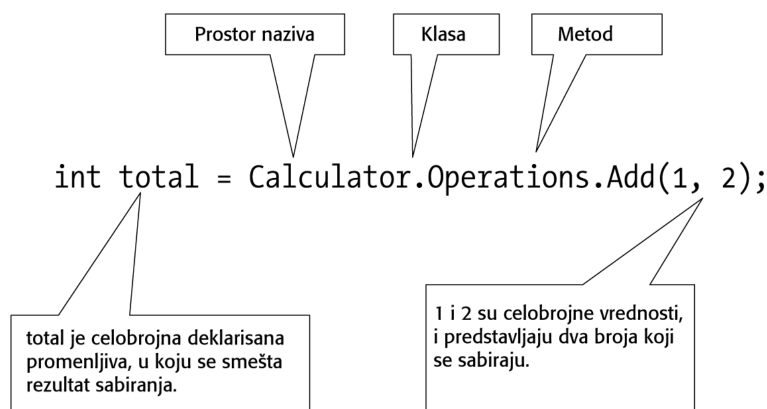
U deklaraciji `Add()` metoda, povratni parametar je identifikator int, što predstavlja celobrojni tip podataka. Metodi i parametri moraju da imaju asociran odgovarajući tip podataka, pošto je C# programski jezik u kome se eksplicitno definišu tpovi podataka. To znači da, prilikom pisanja koda, znate tačno tipove podataka koje koristite.

Pretpostavimo da vi pišete kod i da ste suočeni sa brojevima 1, 1.0 i "1.0". Za vas, ovi brojevi su identični. Ali, u kontekstu izvornog koda, to nije tako. 1 je ceo broj, 1.0 je realan broj dvostruke preciznosti, dok je "1.0" string (niz karaktera). Ukoliko biste želeli da izvršite operaciju sabiranja, oduzimanja, ili na neki drugi način želeli da manipulišete podacima, svi podaci bi morali da budu istog tipa; u suprotnom, može doći do odgovarajućih grešaka prilikom izvršavanja programa. Programski jezici sa eksplicitnom definicijom tipova podataka vam pomažu da

izbegnete prethodno opisane probleme. .NET brojni tipovi su detaljno opisani u odeljku "Razumevanje CLR numeričkih tipova", kasnije u toku ovog poglavlja.

Na osnovu deklaracije `Add()` metoda može se zaključiti da je neophodno da prosledite dve celobrojne vrednosti, a da metod na osnovu toga vraća celobrojnu vrednost. Lista parametara i povratni tip definišu specifikaciju metoda. Specifikacija metoda postaje značajna onda kada je neophodno da u drugim delovima koda pozivate `Add()` metod. U tim delovima koda neophodno je da se koriste isti tipovi, koji su navedeni u deklaraciji metoda.

Na slici 2.7 prikazan je deo koda u kome se poziva `Add()` metod, a koji predstavlja deo aplikacije koju ćemo kreirati u narednom odeljku.



SLIKA 2.7 *`Add()` metod se izvršava referenciranjem prostora naziva i klase u kojoj je definisan metod. Tačka se koristi za izdvajanje različitih identifikatora.*

Prilikom pozivanja metoda neophodno je uraditi sledeće:

- Referencirati ispravnu kombinaciju identifikatora prostora naziva, klase i metoda.
- Proslediti podatke odgovarajućeg tipa, definisanog na osnovu specifikacije metoda.

U prethodnom primeru, sabiranje brojeva 1 i 2 kao rezultat daje vrednost 3, tako da promenljiva total treba da sadrži vrednost 3 (znak jednakosti zapravo omogućava dodeljivanje vrednosti, koju vraća metod promenljivoj sa leve strane znaka jednakosti). Rekli smo "treba da sadrži vrednost", pošto prilikom pisanja koda, nikad ne možete biti stoprocentno sigurni u to. Ponekad će kod koji pišete biti pogrešan zbog toga što ste nešto prevideli, ili zbog toga što ste zaboravili da referencirate nešto.

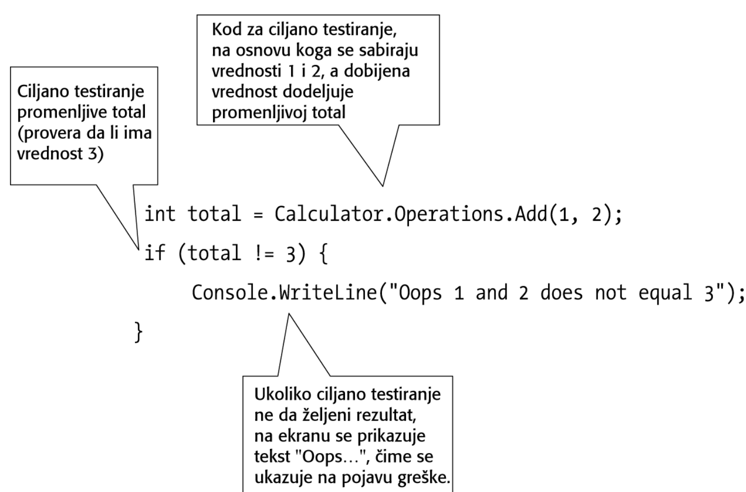
Pogledajte kod kojim se poziva metod, a zatim upitajte sami sebe koliko ste sigurni da ćete izvršavanjem `Add()` metoda, uz korišćenje vrednosti 1 i 2, dobiti rezultat 3. Odgovor je da, sa stanovišta onoga ko koristi metod, nikada ne možete sa sigurnošću da tvrdite da će total promenljiva imati vrednost 3. To što je neka kutija označena sa "Posuđe", ne znači da se posuđe zaista nalazi u toj kutiji. Slično, u izvornom kodu, neophodno je da pogledate na koji način je implementiran `Add()` metod, kako biste u potpunosti bili sigurni u to koju vrednost će imati total promenljiva nakon izvršavanja ovog metoda.

U toku razvoja softvera, stalno pristupanje implementiranom kodu u cilju provere da li se sve odvija na predviđeni način nije zgodno rešenje, pošto bi za to bilo neophodno previše vremena,

a sam postupak nije u potpunosti pouzdan. Jedino realno rešenje je da napišete kod koji će omogućiti testiranje.

Pisanje koda za testiranje `Add()` metoda

Kod za testiranje je zapravo kod koji poziva metod, prosleđujući mu odgovarajuće ciljane vrednosti parametara, sa ciljem da se dobiju željeni rezultati izvršavanja. Ukoliko se ne dobije željeni rezultat, to znači da se javila odgovarajuća greška u implementaciji metoda. Na slici 2.8 prikazan je primer koda za testiranje operacije definisane `Add()` metodom (ovo ćemo nešto kasnije dodati u sam projekat).



SLIKA 2.8 *Testiranje Add() metoda*

Kod na osnovu koga se testira metod veoma je sličan kodu koji ste imali priliku da vidite u prethodnom odeljku. Razlika se ogleda u tome što kod za testiranje koristi ciljane promenljive i vrednosti, dok prethodno kreirani kod sadrži proizvoljne promenljive i vrednosti. Još jedan važan zahtev koji mora da ispuni kod za tesitranje je provera rezultata koje vraća metod, na osnovu unetih ciljanih vrednosti. If naredba se koristi za proveru da li je vrednost promenljive total jednaka 3.

Prilikom pisanja koda za testiranje, način na koji se koristi `Add()` metod mora biti identičan načinu na koji se ovaj metod koristi u konzolnoj ili Windows aplikaciji. U suprotnom, rezultat testiranja će biti identičan rezultatu testiranja zimske gume u središtu pustinje Sahare - zabavno, ali irelevantno.

Kod za verifikaciju u testu je na određeni način specifičan. Da li verifikujete odgovor u toku proizvodnje? Odgovor je - verovatno. Ukoliko pišete verifikacioni kod u scenariju za testiranje, vi pokušavate da obuhvatite sve mogućnosti. Ukoliko pišete verifikacioni kod u produkcionom scenariju, definišete opšti postupak testiranja. Na primer, možete da utvrđujete ima li smisla koristiti neki tip podataka ili šta će se dogoditi ukoliko neki od podataka ne postoji.

Još jedno pitanje, koje je vezano za samo testiranje, je trenutak obavljanja testa. Da li da kreirate testove pre ili nakon implementiranja `Add()` metoda? Da biste u potpunosti razumeli problemantiku, zamislite proizvodnju jedne gume. Da li biste definisali testove za gumu pre ili

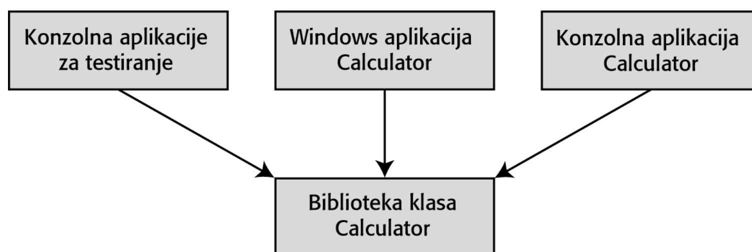
nakon završetka proizvodnje gume? Najčešće je odgovor pre, u toku i nakon završetka proizvodnje. Ovo je veoma važna pretpostavka prilikom razvoja softvera. Testovi se kreiraju pre, u toku i nakon implementacije, na sledeći način:

- Testove kreirate pre implementiranja `Add()` metoda da biste imali predstavu o tome koje prostore naziva, klase i metode ćete definisati. Definicija različitih elemenata daje projektantu ideju o tome na koji način će se elementi koristiti.
- Testove kreirate u toku implementacije `Add()` metoda da biste proverili da li izvorni kod, koji implementirate, zapravo daje željene rezultate.
- Testove kreirate nakon implementacije `Add()` metoda kao konačnu proveru da ste u toku implementacije sve uradili onako kako je neophodno i dobili željene rezultate.

Dodavanje Test projekta u postojeće rešenje

Prilikom pisanja rutina za testiranje, neophodno je da organizujete izvorni kod, a to podrazumeva identifikovanje projekta u koji se dodaju testovi. Kada se radi o Calculator aplikaciji, mogli biste da postavite rutine za testiranje u Calculator biblioteku klasa. Međutim, to nije ispravan pristup, zbog toga što se biblioteka klasa distribuira, odnosno što se menja ispravan kontekst u kome se obavlja testiranje. Zapamtite da rutine testiranja moraju da budu identične sa načinom na koji se koristi odogovarajući kod koji se testira. Zbog toga je prava lokacija rutina za testiranje zapravo aplikacija čiji se metodi testiraju.

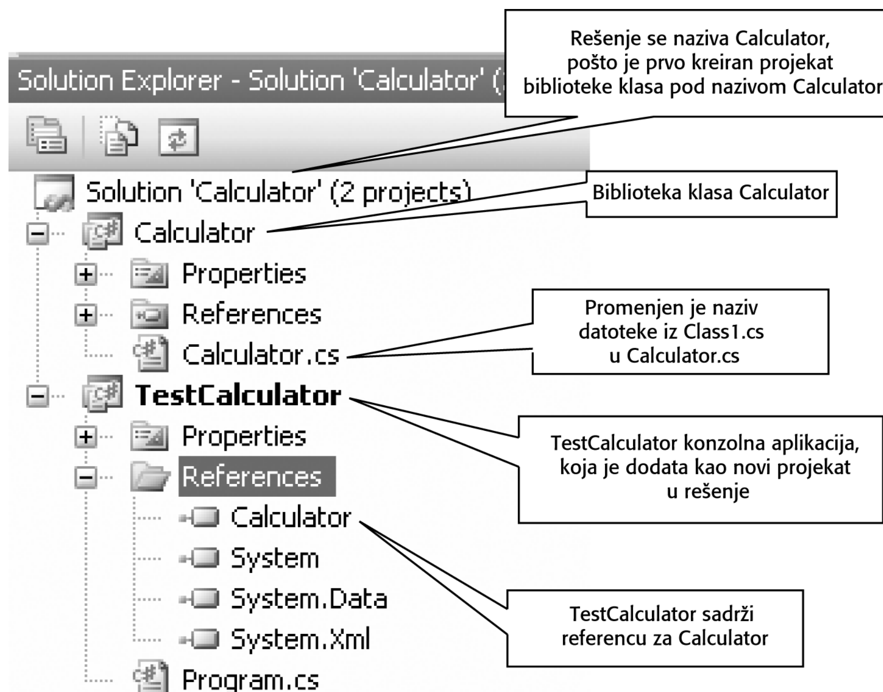
Idealan pristup je kreiranje druge aplikacije, koja predstavlja testove. Na slici 2.5 ilustrovano je kako Windows i konzolna aplikacija treba da koriste Calculator biblioteku klasa. Na slici 2.9 dodata je konzolna aplikacija za testiranje, koja takođe koristi prethodno pomenutu biblioteku klasa.



SLIKA 2.9 *Dodavanje konzolne aplikacije za testiranje, aplikacije sa ograničenom funkcionalnošću koristi se za proveru funkcionalnosti Calculator biblioteke klasa*

Konzolna aplikacija za testiranje je slična konzolnoj aplikaciji, koju ste kreirali u Poglavlju 1, a ona referencira `Calculator` biblioteku klasa. Oba projekta treba da budu deo `Calculator` rešenja.

Sada dodajte `TestCalculator` projekat u `Calculator` rešenje. Ne zaboravite da dodate referencu za Calculator biblioteku klasa (desnim tasterom miša pritisnite References, a zatim u meniju izaberite Add Reference➥Project➥Calculator stavku). Ne zaboravite da `TestCalculator` postavite kao inicijalni projekat za potrebe otkrivanja i uklanjanja grešaka. Na slici 2.10 prikazani su `TestCalculator` i `Calculator` projekti u Solution Explorer prozoru.



SLIKA 2.10 *U Solution Explorer prozoru prikazana je konzolna aplikacija za testiranje i Calculator biblioteka klasa.*

Testiranje jednostavnog sabiranja

U Program.cs datoteku dodajte kod prikazan podebljanim slovima, kako biste u konzolnom projektu za testiranje proverili sabiranje brojeva 1 i 2:

```
namespace TestCalculator {

    class Program {
        public static void TestSimpleAddition() {
            int total = Calculator.Operations.Add(1,2);
            if (total !=3) {
                Console.WriteLine ("Oops 1 and 2 does not equal 3");
            }
        }
        static void Main(string[] args){
            TestSimpleAddition();
        }
    }
}
```

Pritisnite Ctrl+F5 kombinaciju tastera, kako biste testirali izračunavanje. Prilikom izvršavanja programa, konzolna aplikacija za testiranje poziva `TestSimpleAddition()` metod za testiranje, koji poziva i proverava funkcionalnost Calculator biblioteke klasa.

NAPOMENA:

Metod `Main()` definiše početak izvršavanja projekta. Da bi aplikacija mogla da funkcioniše, neophodo je da toj aplikaciji dodate kod `Main()` metoda.

Da biste videli šta se dešava onda kada test ne prođe, promenite Add() metod na sledeći način:

```
public static int Add(int number1, int number2)
{
  return number1 * number2;
}
```

Prilikom ponovnog izvršavanja programa, javlja se poruka o pojavi greške.

Razvojni ciklus

U prethodnom delu ovog poglavlja, imali ste priliku da koristite tri dela koda:

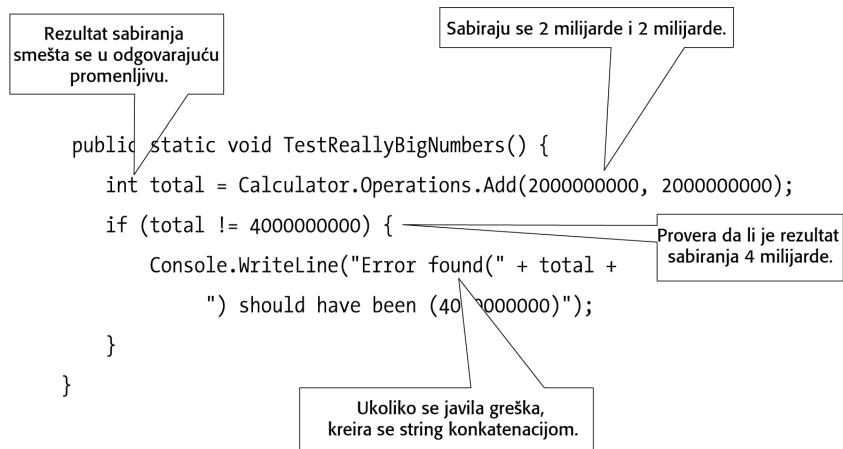
- Kodni segment kojim se implementira Add() metod, a on predstavlja komponentu koja izvršava neophodnu operaciju izračunavanja.
- Kod pomoću koga se poziva metod, koji može biti Windows ili konzolna aplikacija, a koji se smatra produkcionim kodom.
- Kod koji sadrži produkcioni kod sa nekim verifikacionim rutinama, koji zapravo predstavlja kod za testiranje. Kod za testiranje je veoma značajan pošto, ukoliko se promeni implementacija komponente, vi morate ponovo da izvršite samo kod za testiranje, kako biste se uverili da sve funkcioniše onako kako vi to želite.

Pomoću ova tri dela koda demonstriran je celokupan razvojni ciklus softvera.

Testiranje sabiranja dva veoma velika broja

Kod i projekti su organizovani na željeni način, ali nam nedostaju određeni testovi. Trenutno dostupan je samo test za sabiranje dva jednostavna broja. Dodaćemo još jedan test, u kome ćemo sabirati dva veoma velika broja, kao što su 2 milijarde i 2 milijarde.

Kod za testiranje operacije sabiranja dva veoma velika broja prikazan je na slici 2.11. Dodajte ovaj kod u Program.cs datoteku, koja se nalazi u TestCalculator projektu.



SLIKA 2.11 *Testiranje sabiranja dva velika broja*

Test kojim se proverava rezultat izvršavanja operacije sabiranja velikih brojeva je identičan testu kojim se proverava rezultat izvršavanja operacije sabiranja malih brojeva, osim samih brojeva kao argumenata. Poruka o pojavi greške se generiše malo drugačije u odnosu na prethodni primer, jer se primenjuje tehnika nadovezivanja (konkatenacije). U primeru, string se nadovezuje sa celim brojem u obliku stringa. Programski jezik C# će automatski konvertovati ceo broj u odgovarajući string.

Često ćete pisati testove u kojima je jedina realna razlika u odnosu na kod koji se testira vrednost parametara, a ne i njihov tip. Smatrate li da ima velike razlike između sabiranja dva velika broja (kao što su dve milijarde i dve milijarde) i sabiranja dva manja broja (kao što su dva i dva)? Ne, pošto se za ljude razlika ogleda samo u određenom broju nula; da li je rezultat četiri milijarde ili četiri, to izgleda da nema razlike. Ali, kada se radi o izvršavanju operacija na računarima, postoji drastična razlika između četiri milijarde i četiri, što ćete uskoro imati priliku da vidite.

Pre nego što izvršite odgovarajući test, neophodno je da dodate poziv metoda u `Main()` metod:

```
static void Main(string[] args)
{
  TestSimpleAddition();
  TestReallyBigNumbers();
}
```

Sada možete da izvršite test sa veoma velikim brojevima. Dobićete sledeći rezultat:

```
Error found(-294967296) should have been (4869586958695)
```

Generisani rezultat ukazuje na to da se pojavila greška, odnosno da sabiranjem 2 milijarde i 2 milijarde dobijate kao rezultat vrednost `-294967296`, što nema nekog smisla. Šta se dogodilo? Problem je u tome što ste prilikom deklarisanja `Add()` metoda upotrebili brojni tip int.

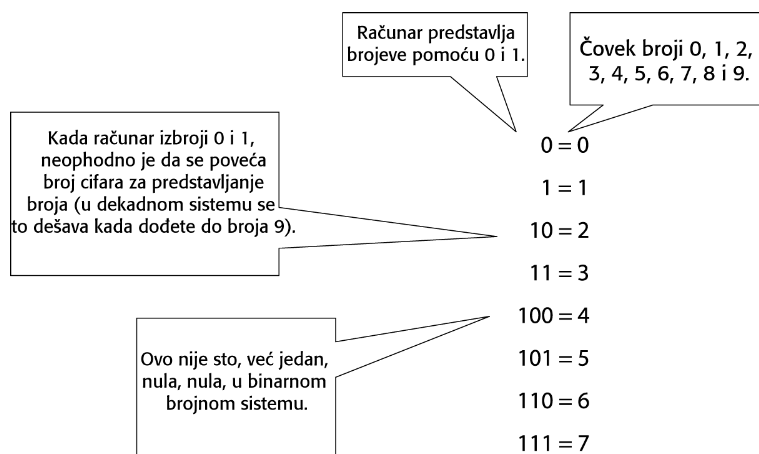
Problemi sa brojnim tipovima podataka

Ono što vi smatrate pod brojevima i ono što računar smatra pod brojevima su dve potpuno različite stvari. Kao dete, vi ste učili da brojite od 1 do 100, što je za vas tada bio ogroman broj. Kako ste rasli, naučuili ste da postoji i broj nula, kao i brojevi manji od nule. Kasnije, naučili ste da postoje razlomci i decimalne vrednosti.

U toku ovog procesa učenja različitih brojeva, vi ste uvek broj 1 i broj 1.5 tretirali na istovetan način - odnosno, to su za vas bili brojevi. Međutim, za računar ovi brojevi nisu vrednosti istog tipa.

Razlog zbog koga se brojevi drugačije tretiraju od strane računara vezan je za efikasnost računara i način na koji se brojevi smeštaju unutar računara. Kada se radi o dekadnom pozicionom brojnom sistemu, počinjete brojanje od 0, nastavljate do 9, a zatim je sledeći broj označen sa 10. Dekadni brojni sistem su inicijalno definisali Vavilonci, ali su oni umesto dekadnog koristili seksadecimalni sistem (60 jedinstvenih identifikatora, umesto 10 jedinstvenih identifikatora u dekadnom brojnom sistemu). Računar ima sličnu šemu brojanja, osim što se koristi binarni brojni sistem sa samo dva jedinstvena identifikatora: 1 i 0. Računari koriste samo dva jedinstvena identifikatora, pošto su njima označena dva jedinstvena stanja: on i off stanje. U srcu računara je tranzistor koji ima mogućnost da razlikuje on i off stanje, odosno ne postoje različite varijante ova dva stanja.

Na slici 2.12 prikazan je primer kako računar broji do 7 korišćenjem binarnog brojnog sistema.



SLIKA 2.12 *Na koji način računar broji do 7.*

Teoretski, mogli biste da brojite do smrti. Imate tu mogućnost, pošto ne postoji nikakvo ograničenje u dekadnom brojnom sistemu koji koristimo. Računar poseduje odgovarajuće ograničenje, kao što je veličina memorije sa slobodnim pristupom (randomaccess memory, RAM), disk računara i slično. Specifični numerički tipovi podataka na računaru takođe sadrže odgovarajuća ograničenja, vezana za opseg brojava koje mogu da predstavljaju. Na primer, celobrojni (int) tip podataka, koji smo koristili u prethodnom primeru, može da prikaže samo određene brojeve, odnosno može da se koristi samo za predstavljanje celih brojeva.

Različiti brojni tipovi podataka su slični meračima kilometraže. Najveći broj merača kilometraže u automobilima sadrži ograničenje od jednog miliona pređenih milja/kilometara. Zamislite šta bi se dogodilo kada biste meračima kilometraže pokušali da saberete 900,000 i 200,000. Rezultat bi bio 100,000, a ne 1.1 milion, kako biste očekivali. Ovo je upravo problem koji se javlja prilikom sabiranja 2 miliona i 2 miliona u prethodnom primeru.

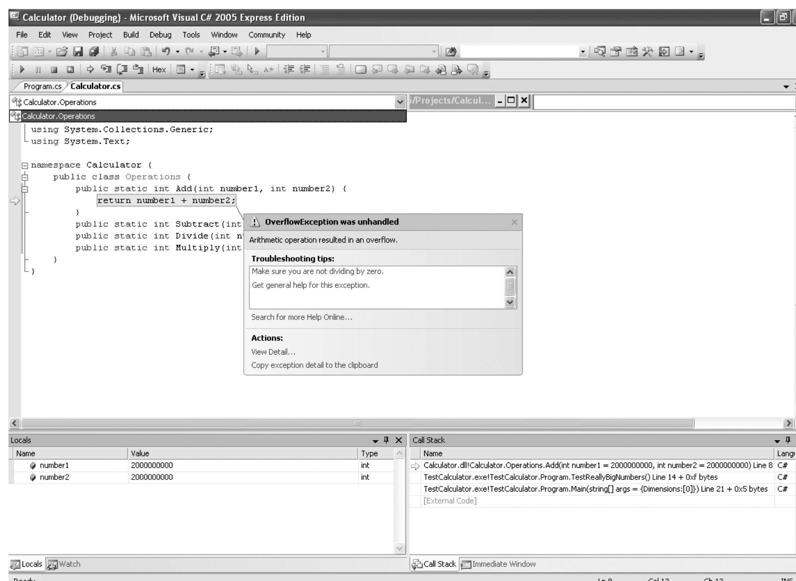
Nezgodna karakteristika merača kilometraže je da ne znate kada je dostignuta maksimalna dozvoljena vrednost. Vi možete da završite sa automobilom na čijem meraču kilometraže piše 100,000 milja/kilometara, ali se zapravo radi o tome da je automobil prešao 1.1 milion milja/kilometara. Na sreću, .NET može da detektuje situaciju u kojoj je došlo do prelaska dozvoljenog opsega kod brojnog tipa. U tehničkom žargonu ova situacija se naziva prekoračenje ili potkoračenje. Prekoračenje podrazumeva situaciju da merač kilometraže pređe dozvoljenu granicu u pozitivnom smeru (900,000 do 1.1 miliona), dok potkoračenje podrazumeva situaciju kada merač kilometraže prekorači granicu u negativnom smeru (0 do -100,000). Detektovanje bilo koje od prethodnih situacija je aktivirano kao svojstvo projekta. Primenite sledeće korake za aktiviranje detekcije prekoračenja/potkoračenja za `Calculator` biblioteku klasa:

1. Desnim tasterom miša pritisnite Calculator projekat u Calculator rešenju, a zatim selektujte Properties stavku.
2. Pritisnite Build, a zatim pritisnite Advanced.
3. Označite opciju "Check for arithmetic overflow/underflow", kako biste testirali situacije vezane za prekoračenje/potkoračenje.
4. Pritisnite taster OK, kako biste završili podešavanje.

Ponovo startujte konzolnu aplikaciju za testiranje, i dobićete informaciju o pojavi izuzetka koji je vezan za prekoračenje (izaberite da program nastavi sa radom, i nemojte da pristupite otkrivanju grešaka):

```
Unhandled Exception: System.OverflowException: Arithmetic operation resulted
in an overflow.
```

Vi možete da otkrijete gde se nalazi greška izvršavanjem aplikacije u režimu otklanjanja grešaka. Pritisnite taster F5, a zatim će se na ekranu prikazati nešto slično onome što je prikazano na slici 2.13. Da biste prekinuli dalji postupak otklanjanja grešaka, neophodno je da pritisnete Shift+F5 kombinaciju tastera.



SLIKA 2.13 *U Visual C# Express okruženju posebno je naglašena greška vezana za prekoračenje dozvoljenog opsega vrednosti.*

Situacija vezana za prekoračenje dozvoljenog opsega vrednosti predstavlja problema, ali je činjenica da .NET može da detektuje ovu grešku veoma dobra za vas. Ali, na kraju, ono što je nama od interesa je da postoji mogućnost da saberete 2 milijarde i 2 milijarde.

Konačno, Bill Gates bi verovatno više želeo da ima 4 milijarde na svom računu u banci, umesto da dobije negativnu vrednost ili neku grešku, koja bi ukazivala da na računu ne može da ima svoje 4 milijarde.

Brojni i vrednosni tipovi podataka

Tip podataka je način za opisivanje odgovarajućih podataka korišćenjem metaopisa. Na primer, ukoliko koristite double tip podataka, znate da ćete imati na raspolaganju broj dvostruke preciznosti. Mnogi tipovi podataka vam stoje na raspolaganju: int, long, short, single, double, string, enum, struct, i tako dalje. Vi možete čak i da definišete sopstvene tipove podataka. Tipovi podataka su suština CLR, odnosno programskog okruženja u kome se primenjuje jako tipiziranje podataka.

Vrednosni i referencni tipovi

CLR podržava dva načina za predstavljanje podataka: na osnovu vrednosti i na osnovu reference. Osnovna razlika između vrednosnog i referencnog tipa ogleda se u načinu na koji je smeštena odgovarajuća informacija asocirana sa tipom. Problem sa vrednosnim i referencnim tipovima se javlja zbog njihove tehničke prirode, i može da bude veoma zbunjujuće.

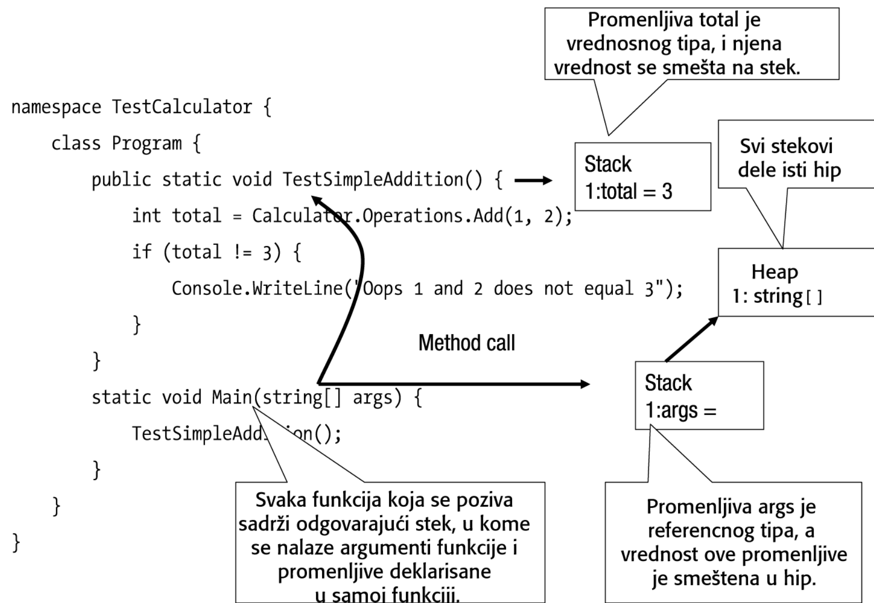
Kada CLR izvršava aplikaciju, nit izrvšava Common Language Infrastructure.

(CLI). O niti možete da razmišljate onda kada se budete šetali po gradu i kupovali stvari. Vi ste pojedinac, i možete da kupujete stvari nezavisno od drugih ljudi. U prodavnici se može naći veliki broj ljudi, koji mogu da se šetaju u prodavnici i kupuju nezavisno od drugih ljudi. Na sličan način, računar može da izvršava veći broj niti, u okviru kojih se obavljaju zadaci nezavisno od onoga što se izvršava u drugim nitima. Ukoliko se šetate u prodavnici, možete da naletite na druge ljude i uzrokujete da im ispadnu stvari iz ruku. Dok CLR pokušava da izbegne takve probleme, ukoliko vaš izvorni kod izvršava veoma složene operacije, vi možete da uzrokujete da neke druge niti "ispuste" određena izračunavanja.

Prilikom izvršavanja, nit sadrži lokalni memorijski blok, koji se naziva stek, što je ekvivalentno novčaniku koji sadrži keš i kreditne kartice. Vi nosite novčanik sa sobom, odlazeći u različite prodavnice, a na sličan način niti koriste stek prilikom izvršavanja različitih metoda. Nakon što uđete u odgovarajuću prodavnicu, želite da kupite nešto, možete da plaćate na jedan od dva moguća načina: korišćenjem keša ili kreditne/debitne kartice. Međutim, ukoliko koristite kreditnu/debitnu karticu, vi ne možete da platite odmah. Tada vam je neophodna mašina, koja poziva server u cilju provere da li vaš komad plastike sadrži dovoljno novca za neophodnu kupovinu. Plaćanje kešom je mnogo brže od plaćanja kreditnom karticom, pošto nije neophodno da se pristupa udaljenom računaru.

Pretpostavimo sada da vi i vaša supruga želite nešto da kupite. Mogli biste da koristite isti račun u banci, ali vi imate posebne kreditne kartice. Ali, ne možete da uradite isto sa kešom. Ukoliko vi imate novčanicu od 10 dolara, vaša supruga ne može da deli tu novčanicu od 10 dolara sa vama. Vaša supruga bi trebalo da ima drugu novčanicu od 10 dolara, a onda biste imali zajedno 20 dolara.

Metodi plaćanja kreiditnom karticom i u kešu su analogni vrednosnim i referencnim tipovima. Keš je analogan vrednosnom tipu, a kreiditna kartica referencnom tipu. Kada CLR izvršava kod, kod koji u jednom metodu poziva neki drugi metod zapravo koristi stek koji sadrži vrednosti promenljivih vrednosnog tipa. Vrednosni tipovi se direktno smeštaju na stek, nalik kešu. Referencni tipovi se smeštaju u obliku pokazivača na memorijske lokacije na steku, kao što vaša kreditna/debitna kartica ukazuje na keš koji se nalazi na vašem računu u banci. Pokazivač kod referencnih tipova ukazuje na deo memorije, koji se naziva hip. Ovi koncepti su ilustrovani na slici 2.14.



SLIKA 2.14 *Stekovi koji su kreirani, odnosno interakcije sa hipom u toku CLR izvršavanja*

Kada je reč o vrednosnim tipovima, prilikom dodeljivanja se kopira sadržaj. Ukoliko modifikujete kopiju, vrednost originala se neće menjati. Nasuprot tome, ukoliko menjate vrednost referencnog tipa, menja se vrednost svih pokazivača na taj referencni tip. Ukoliko se vratimo na primer sa kreditnim karticama i kešom, ukoliko imate 10 dolara, kao i vaša supruga, i vi potrošite 8 dolara, tada to neće uticati na 10 dolara koje poseduje vaša supruga, što je identično onome što se postiže korišćenjem modela vrednosnih tipova. Međutim, ukoliko vi i vaša supruga imate 10 dolara na raspolaganju na kreditnoj kartici, a vi potrošite 8 dolara, ostaće samo 2 dolara, što je ekvivalentno sa onim što se postiže korišćenjem referencnog tipa.

Postoje situacije u kojima ćete koristiti vrednosne tipove, odnosno situacije u kojima ćete koristiti referencne tipove, kao što postoje situacije u kojima plaćate korišćenjem keša, odnosno situacije u kojima koristite kreditnu karticu. Obično kreditne kartice koristite prilikom plaćanja skupocenih stvari, pošto ne želite da nosite sa sobom velike svote novca u kešu. To se odnosi i na vrednosne i referencne tipove, u smislu da ne želite da veliku količinu podataka držite na steku.

Ukoliko znate razliku između steka i hipa, automatski znate i razliku između vrednosnog i referencnog tipa, pošto postoji direktna veza između ovih struktura podataka i tipova podataka. Vrednosni tipovi se obično smeštaju na steku, a sadržaj referencnih tipova se smešta na hip.

CLR brojni tipovi

CLR sadrži dva glavna tipa brojeva: cele brojeve i razlomljene brojeve. Oba ova tipa brojeva su tipovi podataka zasnovani na vrednostima, kao što je opisano u prethodnom odeljku. Metod Add() je koristio tip int, koji predstavlja celobrojni vrednosni tip. Kao što ste imali priliku da vidite, celi brojevi imaju odgovarajuća ograničenja, na osnovu prostora za smeštanje.

Razmotrite sledeći broj:

123456

Ovaj broj sadrži šst cifara. Ilustracije radi, zamislite da na stranici koju upravo čitate, možete da smestite samo šest brojeva. Na osnovu ove informacije, najveći broj koji možete da napišetena stranici je 999,999, dok je najmanji broj jednak 0. Na sličan način, specifični brojni tipovi zahtevaju da CLR definiše odgovarajuća ograničenja vezana za broj cifara koje se mogu koristiti za prikazivanje konkretnog broja. Svaka pozicija je 1 ili 0, odnosno CLR može da predstavlja brojeve u binarnoj notaciji.

Računari mogu da koriste binarnu notaciju, ali ljudi više vole da koriste dekadne cifre, pa za određivanje najvećeg mogućeg broja koji može da se predstavi određenim tipom podataka, broj 2 podignite na stepen koji definiše broj binarnih cifara za predstavljanje broja, a zatim oduzmite 1. Kada se radi o int tipu podataka, za predstavljanje broja se koriste 32 binarne cifre. Pre nego što izračunate najveći mogući broj koji može da se predstavi pomoću int tipa podataka, neophodno je da uzmete u razmatranje i negativne brojeve. Gornja granica za int nije zapravo 4,294,967,295 (rezultat operacije 2^{32} - 1), pošto int tip podataka se koristi i za predstavljanje negativnih brojeva. Drugim rečima, možete da snimite negativan ceo broj, kao što je -2.

Računar koristi trik u prvoj poziciji za predstavljanje broja, jer je ta pozicija rezervisana za definisanje znaka broja (plus ili minus). Kada se radi o int tipu, znači da se samo 31 pozicija koristi za predstavljanje broja, najveći broj koji se može predstaviti je 2,147,483,647, a najmanji -2,147,483,648. Ukoliko se vratimo na primer sa sabiranjem brojeva, to znači da prilikom predstavljanja rezutata sabiranja, 4 milijarde, koja se predstavlja pomoću 32 binarne pozicije, int tip podataka ne sadrži dovoljan broj pozicija za predstavljanje ovog rezultata.

.NET okruženje sadrži brojne tipove podataka, koji su predstavljeni u tabeli 2.1, a ovi tipovi podataka se razlikuju u veličini raspoloživog prostora za predstavljanje brojeva i načina predstavljanja brojeva. Sledeća terminologija se koristi za opisivanje brojnih tipova podataka:

- *Bit* je osnovna jedinica memorije, a 8 bitova predstavlja jedan bajt.
- *Integer* tipom se predstavljaju celi brojevi.
- **Floating-point** tipovi su razlomljeni brojevi.
- *Označi* brojevi su brojevi koji sadrže jednu binarnu poziciju za predstavljanje znaka broja (pozitivan ili negativan).

Tabela 2.1 .NET brojni tipovi podataka

Tip	Opis
`byte`	Neoznačeni osmobitni ceo broj; najmanja vrednost je 0, a najveća vrednost je 255.
`sbyte`	Označeni osmobitni ceo broj; najmanja vrednost je -128, a najveća vrednost je 127.
`ushort`	Neoznačeni šesnaestobitni ceo broj; najmanja vrednost je 0, a najveća vrednost je 65535.
`short`	Označeni šesnaestobitni ceo broj; najmanja vrednost je -32768, a najveća vrednost je 32767.
`uint`	Neoznačeni tridesetdvobitni ceo broj; najmanja vrednost je 0, a najveća vrednost je 4294967295.
`int`	Označeni tridesetdvobitni ceo broj; najmanja vrednost je -2147483648, a najveća vrednost je 2147483647.
`ulong`	Neoznačeni šezdesetčetvorobitni ceo broj; najmanja vrednost je 0, a najveća vrednost je 18446744073709551615.
`long`	Označeni šezdesetčetvorobitni ceo broj; najmanja vrednost je -9223372036854775808, a najveća vrednost je 9223372036854775807.

Tip	Opis
`float`	Tridesetdvobitni relan broj u pokretnom zarezu; najmanja vrednost je 1.5 × 10 - 45, a najveća vrednost je 3.4 × 10 38, sa preciznošću od 7 cifara.
`double`	Šezdesetčetvorobitni realan broj u pokretnom zarezu; najmanja vrednost je 5.0 × 10 - 324, a najveća vrednost je 1.7 × 10 308, sa preciznošću od 15 do 16 cifara.
`decimal`	Specijalni stodvadesetosmobitni tip podataka; najmanja vrednost je 1.0 × 10 -28, a najveća vrednost je 1.0 × 10 28, sa preciznošću od najmanje 28 značajnih cifaraa

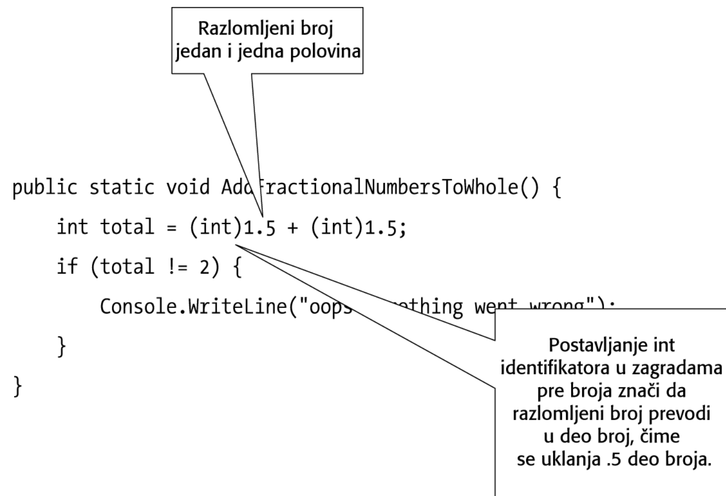
`decimal` tip podataka se često koristi za finansijske podatke, pošto ponekad izračunavanja dovode do vrednosti koja je za jedan peni manja od ispravne vrednosti (na primer, 14.9999, umesto 15.00) zbog greške vezane za zaokruživanje.

Pošto postoji veliki broj tipova podataka koje možete da koristite, možda se pitate kada koji tip podataka treba da se koristi. Brz odgovor je da to zavisi od vaših potreba. Prilikom izvršavanja naučnih izračunavanja, verovatno će biti neophodno da koristite double ili float tip podataka. Ukoliko izračunavate hipoteke, verovatno treba da koristite decimal tip podataka. A ukoliko izvršavate izračunavanja nad skupovima, verovatno biste koristili int ili long tip podataka. Sve zavisi od toga koliko precizno želite da bude izračunavanje, odnosno koliku numeričku preciznost želite.

Numerička preciznost je veoma značajna tema, i važnost preciznosti ne smete nikako da potcenite. Razmotrimo sledeći primer: svaka zemlja obavlja popis stanovništva, a nakon završetka popisa, mogu da se izvuku određeni rezultati. Na primer, u Kanadi, 31% ljudi je razvedeno. Kanada ima populacioni sat koji definiše da svaki minut i 32 sekunde se rodi po jedno dete. U trenutku pisanja ove knjige, Kanada je imala 32,789,736 stanovnika. Dakle, u trenutku pisanja ove knjige 10,164,818 ljudi je bilo razvedeno. Razmislite malo o onome što ste upravo pročitali. Rekli smo da postoji direktna veza između ljudi koji su razvedeni i ukupnog broja rođenja u Kanadi (zapravo, 31%). Možda ćete biti iznenađeni što je broj rođenja i razvoda jednak 10,164,818 - ne 10,164,819 ili 10,164,820 - ljudi će se razvoditi. Naravno, ovo je malo cinično rečeno, sa ciljem da se istakne činjenica da su brojevi zokruženi na određenu vrednost.

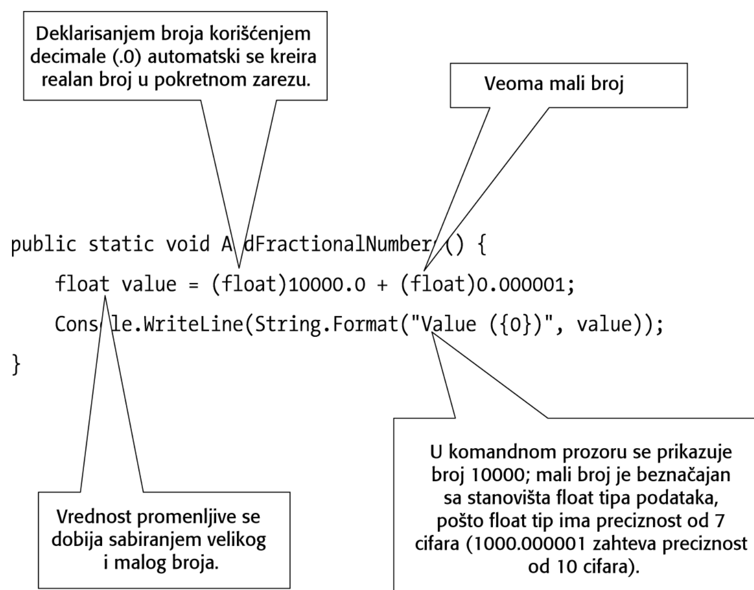
Ne možemo da kažemo da će se, recimo, 10,164,818 ljudi razvesti, pošto to nije dovoljno precizno bez odgovarajućeg brojanja. Možda bismo mogli da kažemo da će se razvesti, recimo, 10,164,818 plus ili minus 100,000. Na ovaj način dobijate opseg od 10,064,818 do 10,264,818, ili grubo govoreći, 10.2 miliona ljudi. Broj 10.2 miliona je ono što bi novinar prikazao u izveštaju, to je ono što bi se navelo u literaturi, odnosno ono što bi većina ljudi koristila u konverzaciji. Tako, ukoliko saberemo 10.2 miliona i 1,000, da li će rezultat biti 10,201,000? Vrednost 10.2 je zaokružena vrednost na najbliži deseti deo miliona, a dodavanje hiljade podrazumeva dodavanje broja koji je mnogo manji od broja na koji se zaokružuje. Odgovor je da ne možete da dodate 1,000 na 10.2 miliona, pošto je 1,000 mnogo manje u odnosu na broj 10.2 miliona. Ali, vi možete da dodate 1,000 na 10,164,818, kako biste dobili 10,165,818, pošto je najznačajnija vrednost ceo broj.

Ako se vratimo na brojne tipove, to znači da celobrojni tipovi imaju tačnost od jednog celog boja. Dodavanje 1.5 i 1.5 kao celih brojeva daje kao rezultat 2, što je prikazano na slici 2.15.



SLIKA 2.15 *Dodavanje razlomljenih brojeva korišćenjem int tipa podataka*

Ovaj koncept značajnih cifara možete da proširite na brojni tip kojim se predstavljaju relani brojevi u pokretnom zarezu, float, pa možete da razmotrite primer koji je prikazan na slici 2.16.



SLIKA 2.16 *Sabiranje razlomljenih brojeva korišćenjem float tipa podataka*

Kao što je prikazano na slici 2.16, ukoliko želite da očuvate preciznost prilikom dodavanja malog broja velikom broju, neophodno je da koristite double tip podataka. Ali, čak i double tip

podataka ima odgovarajuća ograničenja, odnosno preciznost je ograničena na 15 do 16 dekadnih cifara.

Ukoliko vam ni ovo nije dovoljna preciznost, možete da koristiite decimal tip podataka, ali je decimal pogodniji za finansijska izračunavanja. Kada se radi o finansijskim izračunavanjima, imaćete problem prilikom sabiranja velikih i malih brojeva. Zamislite da ste Bill Gates, i da na računu u banci imate nekoliko milijardi dolara. Kada banka računa kamatu, želećete da znate koliko centi ste zaradili, što u nekom dužem vremenskom periodu može da bude poprilična svota. Zapravo, neki programeri su "krali" novac od banaka sakupljanjem delova centa i njihovim akumuliranjem.

Pošto smo opisali određene probleme sa kojima se možete susresti prilikom korišćenja brojeva u svojim aplikacijama, možemo da nastavimo sa kreiranjem kalkulatora.

Završavanje Calculator aplikacije

Originalna deklaracija `Add()` metoda za kalkulator je funkcionisala, ali je imala određena ograničenja vezana za brojeve koji se mogu sabirati. Da biste završili kalkulator, neophodno je da deklarišete `Add()` metod korišćenjem drugog tipa podataka, a zatim da dodate preostale operacije. Za deklarisanje `Add()` metoda mogli bismo da koristimo jedan od tri tipa:

- `long:` rešava problem sabiranja dva veoma velika broja, kao što su 2 milijarde, ali postoji problem da ne možete da saberete dva razlomka, kao što su 1.5 i 1.5.
- `double:` rešava problem sabiranja veoma velikih brojeva ili veoma malih brojeva, a može se primenjivati prilikom sabiranja razlomljenih brojeva. Uopšteno govoreći, double tip je veoma dobar izbor, ali se može javiti problem značajnih cifara, ukoliko sabirate veoma veliki i veoma mali broj.
- `decimal:` generalno dobar pristup, i pogodan za sve tipove preciznosti, ali se odlikuje veoma malom brzinom prilikom sabiranja, oduzimanja ili izvršavanja nekih drugih matematičkih operacija.

Najjednostavniji, opšte prihvaćeni tip među brojnim tipovima podataka je double, pošto obezbeđuje dovoljnu preciznost, a relativno je jednostavan. Celokupna implementacija kalkulatora je prikazana sledećim kodom:

```
public class Operations {
    public static double Add(double number1, double numer2) {
        return number1 + number2;
    }
    public static double Subtract(double number1, double number2) {
        return number1 - number2;
    }
    public static double Divide(double number1, double number2) {
        return number1 / number2;
    }
    public static double Multiply(double number1, double number2) {
        return number1 * number2;
    }
}
```

Četiri operacije prikazane su pomoću metoda sa različitim identifikatorima, ali identičnim opisom funkcije, čime je pojednostavljen postupak primene svakog pojedinačnog metoda. Svaka od operacija može da ima odgovarajući skup testova, na osnovu kojih se proverava ispravnost implementacije. Testovi ovde nisu prikazani, ali su implementirani u priloženom izvornom kodu primera. Preporučujemo da pogledate ove testove, kako biste se uverili da razumete svaki pojedinačni deo koda.

Ono što je neophodno da zapamtite

U ovom poglavlju ste naučili kako se razvija biblioteka klasa, koja se koristi za izvršavanje određenih izračunavanja.

Neophodno je da zamaptite sledeće:

- Organizovanje vaših ideja, projekata i funkcionalnosti je od velike koristi prilikom kreiranja softvera.
- Prilikom kreiranja softvera, uvek budite fokusirani na određene rezultate. Veoma jednostavno se može dogoditi da krenete pogrešnim putem prilikom razvoja softvera, pošto je sam razvoj softvera veoma specifičan. Uspešan projektant uvek mora da bude organizovan i fokusiran.
- Softver se projektuje korišćenjem odgovarajuće arhitekture, koja može da bude implementirana korišćenjem top-down ili bottom-up metoda razvoja.
- U okviru arhitekture, pojedinačni delovi se nazivaju komponente, a komponente se uklapaju tako da se kreira kompletna aplikacija.
- Testove kreirate zato što ne možete u potpunosti da proverite funkcionalnost komponente na osnovu identifikatora, parametara ili povratne vrednosti.
- Prilikom implementiranja komponenti, vi razvijate testove pre, za vreme i nakon pisanja izvornog koda određene komponente.
- Test je deo izvornog koda, koji se koristi za pozivanje komponente korišćenjem tačno izabranih ulaznih podataka, a rezultati izračunavanja se proveravaju u odnosu na očekivane rezultate izvršavanja. Ukoliko se očekivani i dobijeni rezultati ne poklapaju, to znači da komponenta ne funkcioniše na željeni način.
- CLR vam omogućava korišćenje velikog broja različitih tipova podataka, a osnovna razlika je postojanje vrednosnih i referencnih tipova.
- CLR sadrži nekoliko brojnih tipova podataka, ali svi brojni tipovi su zapravo vrednosni tipovi.
- Kod brojnih tipova mogu da se jave prekoračenja i potkoračenja. Neophodno je da aktivirate odgovarajuće podešavanje kompajlera, kako biste se uverili da će CLR na pravi način da detektuje ove situacije.
- Prilikom izbora odgovrajućeg tipa podataka, veliki deo odluke isključivo zavisi od toga koju preciznost prilikom izračunavanja želite da postignete.

Za samostalan rad

U ovom odeljku prikazana su pitanja vezana za ono što ste naučili u ovom poglavlju:

1. Prilikom pisanja koda, na koji način možete organizovati svoj kod? Na primer, možete li da koristite određene identifikatore naziva? Možete li da birate šemu kodiranja? Možete li da koristite odgovarajuće komentare u kodu?

2. U programerskoj zajednici postoji rasprava o tome da li organizovanje softvera treba da uključuje formalne strukture ili treba da bude shodno potrebama. Razmislite o tome na koji način bi trebalo da organizujete svoj softver.
3. Uopšteno rečeno, možete li da izvršite testiranje komponente koja koristi bazu podataka, kako biste utvrdili da li ta komponenta funkcioniše na pravi način? Opišite proces navođenjem određenih faza.
4. Uopšteno rečeno, na koji način biste ispitivali ispravnost upisivanja podataka u odgovarajuću datoteku? Da biste mogli da na pravi način sagledate prirodu samog problema, neophodno je da utvrdite na koji način operativni sistem manipuliše datotekama.
5. Ukoliko CLR nije obezbedio mehanizam za detektovanje stanja prekoračenja ili potkoračenja, na koji način biste mogli da onemogućite pojavu prekoračenja i potkoračenja u svojim aplikacijama?
6. Za Pentium CPU (32 bita), koji tip brojeva bi omogućio najbrža izračunavanja?
7. U primeru u ovom poglavlju, klasa Operations je projektovana tako da izvršava aritmetičke operacije korišćenjem double tipa podataka. Na koji način biste mogli da promenite to, tako da izračunavanja budu generička?