

7. POGLAVLJE

Iznimke

U ovom poglavlju:

- Pogreške u kódu
- Mehanizmi hvatanja iznimki
- Hijerarhija iznimki u .NET-u
- Pravilno hvatanje iznimki

Nemojte se zavaravati – pogreške u kódu su veoma stvarne i ne događaju se nekom drugom! Pritom uopće nije bitno u kojem programskom jeziku radite ili koji razvojni alat koristite, jer su programske greške vezane uz koncept i stil programiranja. Isto tako, greške se vrlo često događaju i zbog različitih vanjskih uvjeta, bez znanja programera (primjerice, pokušavate pristupiti bazi podataka koja je baš u tom trenutku nedostupna zbog administriranja), pa skoro da i nije moguće da vaš kóđ nema niti jednu grešku, već je samo bitno da ih na pravi način uočavate i predviđate mjesta na kojima bi se mogle pojaviti.

Cilj ovog poglavlja je pokazati vam kako se, pišući kóđ u .NET-u, možete nositi s greškama i iznimnim situacijama. Tako ćete naučiti kako iskoristiti najvažnije mogućnosti u svrhu pisanja ispravnog kóda te kako u razdoblju pisanja i testiranja svojih aplikacija pronaći greške.

.NET ima još jednu posebnost o kojoj je već bilo riječi, a zove se *Garbage Collection*. Radi se o metodi upravljanja memorijom koja u potpunosti otklanja mogućnost grešaka

II. DIO: OSNOVE PROGRAMIRANJA

programera i zauzimanja memorije koja se više ne koristi, što bi dovelo do nepotrebnog zauzimanja resursa računala.

Iznimke

Krenimo redom i upoznamo se najprije s iznimkama i metodama hvatanja iznimaka. Vrlo je važno da razumijete što su to iznimke. Ukratko rečeno, radi se o situacijama u kojima vaš kôd radi neispravno. Primjera je doista bezbroj: iznimke će se tako desiti kad pokušate pisati u datoteku koja ima postavljen atribut *Read Only*, kad pokušate napraviti matematičku operaciju nad dva znakovna niza (što je moguće samo nad brojevima) ili pak kad u vašem programu pokušate pristupiti nekom resursu na Internetu, a ne postoji aktivna veza na Internet. Naravno, tipična pogreška koja uzrokuje iznimku je i pokušaj dijeljenja s nulom, što bi rezultiralo beskonačnim brojem koji računalo nije u stanju prikazati.

Sve te situacije rezultirat će prekidanjem rada vašeg programa i ispisivanjem greške. Rješenje za te probleme je jednostavno – trebate samo ostvariti mehanizme hvatanja iznimki i upravljanja njima. Primjerice, u slučaju greške pri pokušaju pisanja u *Read Only* datoteku, korisniku trebate ponuditi spremanje datoteke pod nekim drugim imenom, a pri dohvaćanju nekog resursa s Interneta kad ne postoji aktivna veza samo trebate ispisati odgovarajuću poruku u kojoj ćete upozoriti korisnika da se treba spojiti na Internet.

Korištenjem .NET-a imate idealne mogućnosti hvatanja iznimki. Tako možete sav kôd koji će barataći greškama držati na odvojenom mjestu te tako imati čist i pregledan glavni kôd. Naravno, hvatanjem iznimki možete vrlo lako uočiti probleme koji se često ponavljaju u vašem kodu u periodu testiranja te ih lako ukloniti.

Mehanizmi hvatanja iznimki

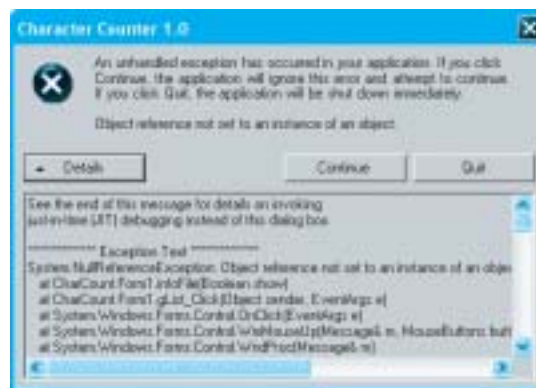
U C#-u vam na raspolaganju stoji nekoliko naredbi koje zajedno čine strukturu za hvatanje iznimki. Radi se o bloku *try catch finally* – kasnije ćemo detaljno objasniti svaku od naredbi, a prvo ćemo objasniti cijeli blok i njegovu namjenu.

Kamo bježe neuhvaćene iznimke?

Sve iznimke koje ne uhvatite u svojim aplikacijama i na njih pravilno ne reagirate izazvat će grešku u radu aplikacije, bilo da se radi o web-aplikaciji u ASPNET-u ili o desktop

aplikaciji pisanoj uz pomoć Windows Forms-a. Rezultat jedne pogreške možete vidjeti na slici 7-1, a radi se o nečemu što ne želite da vide vaši korisnici.

7. POGLAVLJE: IZNIMKE



Slika 7-1:
Greška u Windows
Forms aplikaciji

```
try {
    // Ovdje se nalazi "problematični" kôd koji bi mogao
    // uzrokovati pojavljivanje greške.
}
catch (InvalidCastException) {
    // Kôd koji se oporavlja od InvalidCastException iznimke
    // i svih drugih proizašlih iz nje
    // (detaljnije o "tim drugim" iznimkama kasnije).
}
catch (Exception e) {
    // Kôd koji hvata sve iznimke
    // (zato jer sve iznimke proizlaze iz Exception iznimke).
}
finally {
    // Kôd koji "čisti" sve operacije započete u try bloku.
    // Taj se kôd uvijek izvršava, neovisno o pojavljivanju
    // iznimke.
}
```

U C#-u se mogu pojaviti samo CLS (*Common Language Specification*) iznimke, a to su one proizašle iz `System.Exception` (kasnije ćemo detaljnije objasniti hijerarhiju iznimki, no sad je najvažnije da shvatite da se na vrhu hijerarhije nalazi `System.Exception`, a sve druge su ispod nje, pa kažemo da su *proizašle* iz nje). Imate li *catch* blok koji hvata iznimku tipa *Exception*, zapravo hvatate sve moguće iznimke.



II. DIO: OSNOVE PROGRAMIRANJA

Ovdje postoje dvije *catch* naredbe – vi ih možete imati koliko god želite i svakom možete hvatati posebnu vrstu iznimki, no najčešće će to biti samo jedna ili dvije *catch* naredbe.

try blok

U *try* bloku se nalazi “problematičan” kôd koji bi mogao izbaciti grešku i uzrokovati pojavljivanje iznimke. Takvih primjera ima jako mnogo, neke smo spomenuli već na početku teksta, a evo ih još nekoliko: možda pokušavate pristupiti bazi podataka za koju nemate određene dozvole ili pak želite obraditi neke podatke koje je korisnik upisao, no oni su upisani u krivom obliku.

Važno je primijetiti da ste vi kao programer uočili potencijalne probleme i taj kôd smjestili unutar *try* bloka, što vam omogućava hvatanje i pravilno rukovanje s pogreškama.

catch blok

U *catch* bloku se nalazi kôd koji će se izvršiti u slučaju pojavljivanja iznimke. Izraz u zagradi nakon *catch* naredbe je zapravo filter i on određuje u slučaju koje iznimke će se izvršiti kôd. U prethodnom primjeru imali smo *InvalidCastException* filter koji bi uhvatio sve iznimke *InvalidCastException* tipa i sve one koje su, po hijerarhiji, proizašle iz nje.



Obavezno catch blokove posložite tako da specifične iznimke idu na vrhu, a općenite na dnu. Ukoliko pak napravite obrnuto, čak će vas i kompajler upozoriti ako se više specifičnih iznimki pojavljuje pri dnu, jer se taj kôd nikad neće izvršiti.

Primijetite da se *catch* blokovi u slučaju iznimke provjeravaju od vrha prema dnu, pa je nužno staviti specifične iznimke na vrh (primjerice, *InvalidCastException*), a one općenitije pri dnu (primjerice, *Exception* koja hvata sve iznimke).

finally blok

Finally blok sadržava kôd koji će se sigurno izvršiti i taj kôd najčešće služi za čišćenje i oslobađanje resursa koji su bili korišteni u *try* bloku. Jednostavan je primjer pri korištenju datoteka:

```
using System.IO;

void koristiDatoteku(string ime)
{
    FileStream fs = null;
    try
    {
```

```

        fs = new FileStream(ime, FileMode.Open);
        // kôd koji radi s datotekom i nešto u nju zapisuje
    }
    catch (Exception)
    {
        // Pojavila se neka greška pri radu s datotekom.
    }
    finally
    {
        if (fs != null) fs.Close();
    }
}

```

U *try* bloku pristupamo datoteci i pokušavamo nešto s njom raditi. U slučaju da se pojavi bilo kakva greška, hvatamo je *catch* blokom. No u tom slučaju datoteka ostaje otvorena i u upotrebi – zato nam je nužan *finally* blok.

Ne bi bilo ispravno staviti naredbu za zatvaranje datoteke poslije bloka za hvatanje iznimki: što ako se pojavi iznimka, no ona se ne uhvati u *catch* bloku jer se koristila neka specifična iznimka kao filter? U tom slučaju će datoteka ostati otvorena i nakon što vaš program završi, što je neprihvatljiva situacija.



Sav kôd u njemu će se izvršiti u svakom slučaju, bilo da se pojavila greška ili je sve prošlo u redu. U slučaju da se pojavi greška, zatvorit će se datoteka, a u slučaju da je sve prošlo u redu, svejedno će se izvršiti *finally* blok i datoteka će se zatvoriti.

Ponekad vam i ne treba *finally* blok, jer jednostavno niste koristili resurse koje obavezno treba “počistiti” na kraju. Važno je samo primijetiti da *finally* blok dolazi poslije svih *catch* blokova i da može postojati samo jedan u *try catch finally* dijelu, za razliku od *catch* blokova, kojih može biti više.

Rad s iznimkama

Mehanizam upravljanja iznimkama može hvatati sve iznimke bazne klase *System.Exception* i svih podklasa nastalih iz nje. Stoga sve iznimke imaju neka ista svojstva koja možete koristiti pri hvatanju iznimki kao izvor korisnih informacija o grešci koja se pojavila. U tablici 7-1 nalaze se tri glavna svojstva klase *System.Exception* koje nasljeđuju sve podklase.

U praksi možete iskoristiti ova svojstva za ispisivanje informacije o grešci. No tad ćete u *catch* bloku morati, uz tip iznimke koju hvatate, upotrijebiti ime varijable koja će sadržavati sva ta svojstva. Pogledajte naredni primjer, u kojem ispisujemo informaciju o grešci (radi se o aplikaciji za konzolu):

II. DIO: OSNOVE PROGRAMIRANJA

Tablica 7-1:
Tri glavna svojstva
Exception klase

Svojstvo	Opis
Message	Tekst pogreške koji opisuje što se desilo i zašto se pojavila iznimka
Source	Ime <i>assembly</i> koji je stvorio iznimku
StackTrace	Ime metode, datoteka s pogreškom i linija u kojoj se pojavila pogreška

```
//...
try
{
    // kôd koji uzrokuje pogrešku
}
catch (Exception e)
{
    Console.WriteLine("Tekst pogreške: " + e.Message);
}
//...
```

Primjerice, pojavi li se greška dijeljenja s nulom, program će ispisati: "Tekst pogreške: Attempted to divide by zero".

Želite li pak ispisati detaljnu informaciju o izvoru greške, upotrijebite *StackTrace* svojstvo (u gornjem primjeru ispišite *e.StackTrace*). Dobit ćete informaciju o metodi u kojoj se pogreška pojavila, datoteci u kojoj se nalazi ta metoda i liniji na kojoj se nalazi naredba koja je uzrokovala grešku. Primjerice, ta informacija može imati sljedeći oblik, što je izrazito korisno pri otkrivanju izvora pogreške:

```
at MojaAplikacija.Class1.Main(String[] args) in
e:\code\c#\MojaAplikacija\class1.cs:line 23
```

Hijerahija iznimki

U .NET-u postoji *System.Exception* klasa koja definira općenitu iznimku, a sve druge iznimke proizlaze iz nje i realizirane su obliku njenih podklasa. Najvažnije dvije klase proizašle iz *Exception* klase su *ApplicationException* i *SystemException*. *ApplicationException* klasa vam omogućava definiranje vlastitih iznimki specifičnih za vašu aplikaciju.

SystemException su iznimke koje se mogu pojaviti bilo kad za vrijeme izvršavanja aplikacije i zato se još zovu *Runtime* iznimke (pojavljuju se u *Runtimeu* odnosno za vrijeme izvršavanja). Primjeri-

Pogreška u pogreški u pogreški u...

Što ako vam se desi pogreška u *finally* bloku? Ništa posebno – ta pogreška identična je bilo kojoj drugoj koja se pojavljuje poslije *try catch finally* bloka. No kako hvatati takve pogreške? Posve jednostavno – ako želite, možete koristiti više *try catch finally* blokova jednih u drugima. Primjerice, što ako bi se “zabunili” i u gornjem primjeru u *finally* bloku nakon zatvaranja datoteke pokušali pristupiti datoteci (primjerice, ispisati njenu duljinu uz pomoć *fs.Length*)? Kôd koji bi ispravno rukovao tom greškom može izgledati ovako:

```
try
{
    FileStream fs = null;
    try
    {
        fs = new
        FileStream(ime,
        FileMode.Open);
        // kôd koji radi s
        datotekom i nešto u nju
        zapisuje
    }
}
```

```
catch (Exception)
{
    // Pojavila se neka
    greška pri radu s datotekom.
}
finally
{
    if (fs != null)
    fs.Close();
    // korištenje
    fs.Length svojstva
}
catch (Exception)
{
    // pojavila se još jedna
    greška
}
```

Na raspolaganju vam stoji neograničen broj *try catch finally* blokova. Naravno, skoro uvijek će vam biti dosta samo jedan blok u kojem predvidite sve moguće pogreške, a svakako pokušajte u *catch* i *finally* blokovima pisati što jednostavniji kôd koji ne može uzrokovati dodatne pogreške i iznimke.

ce, pokušate li pristupiti nepostojećem članu nekog polja, CLR će vam dojaviti *IndexOutOfRangeException*, koja je proizašla iz *SystemException* klase. Slično, pokušate li raditi s objektom koji još ne postoji (stvoren je, no ne i instanciran), dobit ćete *NullReferenceException*.

Slijedi sažeta hijerarhija klasa s nekoliko najvažnijih tipova iznimaka, da dobijete dojam kakve sve iznimke postoje:

```
System.Exception
System.ApplicationException
```

II. DIO: OSNOVE PROGRAMIRANJA

```

    System.Reflection.TargetException
    ...
System.IO.IsolatedStorage.IsolatedStorageException
System.SystemException
    System.ArgumentException
        System.ArgumentNullException
        System.ArgumentOutOfRangeException
    ...
System.ArithmeticException
    System.DivideByZeroException
    System.OverflowException
    ...
System.FormatException
System.IndexOutOfRangeException
System.InvalidCastException
System.IO.IOException
    System.IO.DirectoryNotFoundException
    System.IO.FileNotFoundException
    ...
System.NullReferenceException
System.OutOfMemoryException
System.Security.Policy.PolicyException
    System.Security.SecurityException
System.Threading.SynchronizationLockException
System.Threading.ThreadInterruptedException
System.TypeLoadException
    System.DllNotFoundException
    System.EntryPointNotFoundException
System.UnauthorizedAccessException
    ...
    ...
    ...

```

Potpunu hijerarhiju klasa možete dobiti potražite li u MSDN-u “Exception class” i odaberete li “Derived classes”.

Što nam zapravo govori prethodni popis? Pomoću njega dobivate jasniju sliku i razumijete zašto *System.Exception* hvata sve iznimke. Primjerice, pojavi li se iznimka tipa *System.DivideByZeroException*, nju biste uhvatili bilo kojim od sljedećih *catch* blokova:

```

    catch (DivideByZeroException)
    catch (ArithmeticException)

```


7. POGLAVLJE: IZNIMKE

```
catch (SystemException)
catch (Exception)
```

Hijerarhija klasa iznimaka ima za posljedicu i da će *ArithmeticException* blok uhvatiti sve iznimke svojih podklasa, primjerice *System.DivideByZeroException* i *System.OverflowException*. To je najveća prednost postojanja ovakve hijerarhije. Zato *System.Exception* koji se nalazi na vrhu hijerarhije hvata baš sve.



Od mnogih iznimaka moguće je oporaviti se pravilnim hvatanjem i rukovanjem, kao što je slučaj i s, recimo, spomenutom *DivideByZeroException* iznimkom. No neke se iznimke, poput *StackOverflowException*, zbog svoje prirode (stog se napunio zbog previše poziva metoda i nema resursa za daljnje naredbe) smatraju fatalnima: skoro je nemoguće oporaviti se od njih u kôdu i gotovo sigurno će uzrokovati prekid rada aplikacije.

U C#-u postoji još jedan *catch* blok, koji će poput *catch (Exception)* hvatati sve iznimke. Jednostavnije ga je napisati, jer u sebi ne sadržava niti jedan filtar iznimki:

```
catch {
    // kôd za rukovanje s bilo kojom iznimkom
}
```



Pravilno rukovanje iznimkama

Kao i kod svih drugih tehnologija i tehnika programiranja, bez poznavanja sintakse ne možete ništa, no mnogo važnije je znati kako pojedinu tehniku pravilno upotrebljavati. Isti je slučaj i s hvatanjem iznimki. Slijedi nekoliko korisnih savjeta koje možete iskoristiti u svakoj situaciji.

Za početak, pojasnimo još jednom što podrazumijeva hvatanje iznimki. Kad napišete *try catch* blok, to znači da ste u tom kôdu očekivali iznimku, da razumijete kako se desila i da znate što s njom učiniti i kako ispravno postupiti u oporavku od greške.

Lako vas može povesti mogućnost da hvatate sve iznimke korištenjem *catch (Exception)* bloka. No što u tom slučaju želite napraviti? Zar doista mislite da se u vašem kôdu može dogoditi bilo

II. DIO: OSNOVE PROGRAMIRANJA

kakva iznimka? Zar mislite da svojim kôdom u *catch* bloku možete predvidjeti sve situacije i od njih se na pravi način oporaviti?

Hvatanje svih iznimki možda i jest korisno pri izradi aplikacija kad želite saznati detaljnije informacije o svakoj grešci koja se pojavi tako da u *catch* bloku ne napravite baš ništa, osim što ispišete informaciju o grešci. No i tad je mnogo prikladnije koristiti *debugger* koji će vam javiti što je pošlo krivo u vašoj aplikaciji. Hvatanjem svih iznimki zapravo u svom kôdu govorite da ste spremni na bilo kakvu grešku i da možete u svojoj aplikaciji napraviti baš sve što je potrebno da se oporavite od svih grešaka. Zar vam se to ne čini besmislenim?

Također, kad se govori o oporavljanju od iznimaka, važno je da se od njih oporavljate na pravi način. Nužno je da, ako ste u mogućnosti, napravite ipak nešto više od pukog ispisivanja poruke o greški. Od vas se očekuje da u situacijama iznimki zatvorite datoteku koja je uzrokovala problem, otkazete daljnje izvršavanje programa koji pokušava kontaktirati nedostupnu bazu podataka ili pomognete korisniku u ispravljanju mogućih grešaka ako je do iznimke došlo njegovim upisom neispravnih podataka.

Veoma je važno i koristiti *finally* blok kad god je to moguće i potrebno. Iako možete *try catch finally* blok napisati bez *finally* bloka i u njemu samo provjeravati kôd i izvršiti jednostavno hvatanje iznimki, veoma je korisna mogućnost koju *finally* nudi. Ponovimo još jednom, korištenjem *finally* bloka osiguravate se da će kôd sadržan u njemu doista i biti izvršen. Ako se dogodi iznimka, to je ključna stvar, jer će se u protivnom prekinuti daljnje izvršavanje, dok ćete korištenjem *finally* bloka još imati na raspolaganju izvršavanje nekoliko naredbi kojima ćete osloboditi korištene resurse i slične stvari. Čak i ako se ne dogodi iznimka, *finally* blok će se izvršiti, pa je tako njegova uloga višestruka.



Hvatanje iznimki nije ovisno o jeziku u kojem je pisan dio programa u kojem se dogodila iznimka, jer su sve iznimke tipa *System.Exception*, a on je ugrađen u sam *.NET framework*. Primjerice, možete u C#-u korištenjem ovdje opisane *try catch finally* sintakse hvatati iznimke koje se dešavaju u metodama pisanim u VB.NET-u: u prethodnom primjeru je metoda *provjeriKorisnika* mogla biti spremljena u posebnom modulu pisanom u VB.NET-u, a to ne bi uopće utjecalo na rad programa – pojavila bi se iznimka i ona bi bila uhvaćena u C# dijelu programa.

Baci bombu, goni iznimku...

Osim što u .NET-u možete hvatati iznimke, vi ih možete i stvarati odnosno *bacati*. To postižete korištenjem *throw* naredbe kojom stvarate novu iznimku i prosljeđujete je ostatku programa. Primjerice, u glavnom dijelu programa imate realiziran *try catch finally* blok u kojem obavljate neke akcije, pozivate različite vlastite funkcije i provjeravate pojavljivanje iznimki.

No kad se u nekoj od pozivanih funkcija dogodi nešto što onemogućava njeno daljnje ispravno izvršavanje (primjerice, proslijeđeni parametri nisu u odgovarajućem obliku), vi možete *baciti* iznimku. Ona će biti uhvaćena u glavnom programu i bit će ispisane detaljnije informacije o greški. Pogledajte sljedeći kôd:

```
void GlavnaMetoda() {
    try
    {
        // uvodni kôd
        provjeriKorisnika("Hrvoje
        Horvat");
        // kôd koji očekuje da
        određeni korisnik postoji
    }
    catch (NotSupportedException
    e)
    {
```

```
        // ispis informacije o
        grešci, oslobađanje resursa...
    }
}

void provjeriKorisnika(string
korisnik) {
    // ako korisnik ne postoji...
    throw new
    NotSupportedException("Korisnik
    " + korisnik + " ne postoji!");
}
```

U ovom smo primjeru stvorili vlastitu iznimku! Ako u funkciji *provjeriKorisnika* doista bude utvrđeno da korisnik "Hrvoje Horvat" ne postoji i izvrši se *throw* naredba, u *catch* dijelu u glavnoj metodi može se ispisati poruka iznimke korištenjem *e.Message* koja će glasiti "Korisnik Hrvoje Horvat ne postoji!".

Sama sintaksa *throw* naredbe je jednostavna – ona baca novu iznimku, stoga iza nje slijedi *new* i tip iznimke (podsjetimo se, korištenjem *new* operatora stvara se novi objekt određenog tipa). Parametara nove iznimke može biti više, no možete se odlučiti i da samo upišete poruku koju će iznimka sadržavati. Tu poruku zatim možete ispisati iz drugih dijelova programa u kojem tu iznimku uhvatite.

