12

Virheenkäsittely poikkeusten avulla

Yksi .NET Common Language Runtime (CLR):n päämääristä on joko välttää ajonaikaiset virheet (automaattisen muistin- ja resurssienhallinnan avulla, kun käytetään hallittua koodia), tai vähintään saada ne kiinni käännöksessä (voimakkaan tyypityksen avulla). Määrätyt virheet voidaan kuitenkin saada kiinni vain suorituksen aikana ja siksi vakaan virheidenkäsittelyn pitää johdonmukaisesti toimia kaikkien Common Language Specification (CLS) -määritystä noudattavien kielten kesken. Tämä luku keskittyy CLR:n toteuttamaan virheenkäsittelyjärjestelmään, poikkeusten käsittelyyn.

Tässä luvussa opit aluksi poikkeustenkäsittelyn yleisen tekniikan ja perussyntaksin. Kun sinulla on nämä perustiedot, näet miten poikkeusten käsittely suhteutuu yleisimpiin nykyisin käytössä oleviin virheenkäsittelymenetelmiin ja opit, mitä etuja poikkeusten käsittelyllä on näihin muihin verrattuna. Sen jälkeen sukellamme yksityiskohtaisemmin .NETin poikkeusten käsittelyyn, kun tutustumme *Exception*-luokan käyttämiseen ja omien poikkeusluokkien periyttämiseen. Luvun loppuosassa käsittelemme oikeanlaista poikkeusten käsittelyn huomioimista koko järjestelmän suunnittelussa.

Johdanto poikkeusten käsittelyyn

Poikkeukset ovat virhetilanteita, jotka syntyvät, kun koodin normaalia suorituspolkua (joukko pinoon kohdistuvia metodikutsuja) on mahdoton suorittaa. Tässä on välttämätöntä ymmärtää ero poikkeuksen ja odotetun tapahtuman (kuten tiedoston lopun saavuttaminen) välillä. Jos sinulla on metodi, joka lukee tiedostoa peräkkäin tiedät, että jossain kohtaa se

saapuu tiedoston loppuun. Siksi tällainen tapahtuma ei ole luonteeltaan poikkeuksellinen eikä tietenkään saa estää ohjelman toiminnan jatkumista. Jos yrität lukea tiedostoa ja käyttöjärjestelmä varoittaa sinua levyvirheestä, tämä on poikkeuksellinen tilanne ja ilman muuta sellainen, joka vaikuttaa ohjelmasi normaaliin suoritukseen metodisi yrittäessä jatkaa tiedoston lukemista.

Useimmat poikkeukset aiheuttavat myös toisen ongelman: ympäristö (context). Katsotaan esimerkkiä. Oletetaan, että kirjoitat tiukkaa koodia (koodia, jossa yksi metodi vastaa yhdestä toiminnosta) ja pseudokoodisi näyttää tältä:

```
public void Foo()
{
    File file = OpenFile(String fileName);
    while (!file.IsEOF())
    {
        String record = file.ReadRecord();
    }
CloseFile();
}
public void OpenFile(String fileName)
{
    // Yritys lukita ja avata tiedosto.
}
```

Jos *OpenFile*-metodi epäonnistuu, se ei pysty käsittelemään virhettä. Tämä johtuu siitä, että sen ainoa velvollisuus on avata tiedostoja. Se ei pysty määrittelemään, aiheuttaako tiedoston avauksen epäonnistuminen täyden katastrofin vai pienen haitan. Siksi *OpenFile* ei pysty käsittelemään virhetilannetta, koska sanotaan, että se ei ole oikeassa ymäristössä.

Tämä on perussyy koko poikkeuksen käsittelyn olemassaoloon: yksi metodi määrittelee, että poikkeustilanne on kohdattu mutta se ei ole oikeassa ymäristössä käsitelläkseen sen. Se ilmoittaa ajonaikaiselle ympäristölle, että virhe on tapahtunut. Ajonaikainen ympäristö palaa takaisin kutsupinossa kunnes se löytää metodin, joka pystyy käsittelemään virhetilanteen asiallisesti. Tämä tulee ilman muuta tärkeämmäksi, jos kutsupino on viisi metodia syvä ja virhe tapahtuu alimmassa ja ensimmäinen metodi on ainoa, joka kykenee virheen käsittelemään. Katsotaan nyt syntaksi, jota poikkeuksen käsittelyssä käytetään.

Normaalin poikkeuksen käsittelyn syntaksi

Poikkeuksen käsittely koostuu vain neljästä avainsanasta: *try*, *catch*, *throw* ja *finally*. Tapa, jolla nämä sanat toimivat, on selvä ja yksinkertainen. Kun metodi epäonnistuu tavoitteessaan eikä pysty jatkamaan, eli kun se kohtaa poikkeustilanteen, se aiheuttaa (throws) poikkeuksen kutsuvalle metodille käyttämällä avainsanaa *throw*. Kutsuva metodi, olettaen, että se on oikeassa ympäristössä käsitelläkseen poikeuksen, vastaanottaa tämän poikkeuksen *catch*-avainsanalla ja päättää, mitä kuuluu tehdä. Seuraavissa kappaleissa katsomme poikkeuksen aiheuttamiseen ja kiinni ottamiseen liittyvää kielen semantiikkaa sekä muutamia koodipätkiä, jotka näyttävät, miten se toimii.

Poikkeuksen aiheuttaminen

Kun metodin pitää ilmoittaa kutsuvalle metodille, että virhe on tapahtunut, se käyttää *throw*-avainsanaa seuraavalla tavalla:

throw *käsky*:
throw *lauseke* $_{opt}$;

Menemme hetken kuluttua erilaisiin tapoihin, joilla poikkeuksen voi aiheuttaa. Nyt riittää, kun tiedät, että halutessasi aiheuttaa poikkeuksen, sinun pitää "heittää" *System.Exception*-objekti tai siitä periytyvä objekti. Seuraavassa on esimerkki metodista, joka huomaa korjaamattoman virhetilanteen tapahtuneen ja aiheuttaa siksi poikkeustilanteen kutsuvalle metodille. Huomaa, miten uusi *System.Exception*-objekti instantioidaan ja sen jälkeen "heitetään" ylös kutsupinoon.

```
public void SomeMethod()
{
// Jokin virhetilanne havaittu.
throw new Exception();
}
```

Poikkeuksen kiinniottaminen

Koska metodi voi heittää poikkeuksen, pitää tietenkin olla vastatoiminto eli tapa, jolla joku voi ottaa heitetyn poikkeuksen kiinni. Avainsanalla *catch* määritellään koodilohko, joka suoritetaan, kun määrätyn tyyppinen poikkeus otetaan kiinni. Lohkon koodia sanotaan *poikkeuksenkäsittelijäksi* (exception handler).

Yksi mielessä pidettävä seikka on se, että jokaisen metodin ei tarvitse käsitellä kaikkia aiheutettuja poikkeuksia, erityisesti siksi, että metodi ei välttämättä ole ympäristössä, jossa se voisi tehdä virheelle mitään. Loppujen lopuksi, poikkeustenkäsittelyn juju on siinä, että virheet käsittelee koodi, joka on sopivassa ympäristössä tehdäkseen sen, katso tämän luvun myöhempää kappaletta "Virheiden käsitteleminen oikeassa ympäristössä." Katsotaan nyt tilanteita, joissa metodi ottaa kiinni minkä tahansa kutsuvan metodin heittävän poikkeuksen. Poikkeuksen kiinniottamisessa käytetään kahta avainsanaa: *try* ja *catch*.

Jotta voisit ottaa poikkeuksen kiinni, sinun tulee sijoittaa koodi, jota yrität suorittaa, sulkujen sisään *try*-lohkoon ja sen jälkeen määritellä minkä tyyppiset *try*-lohkon koodissa tapahtuvat poikkeukset *catch*-lohko kykenee käsittelemään. Kaikki *try*-lohkon käskyt suoritetaan järjestyksessä normaalisti, kunnes jokin kutsutuista metodeista aiheuttaa poikkeuksen. Kun se tapahtuu, ohjelman suoritus siirtyy vastaavan *catch*-lohkon ensimmäiselle riville. "Vastaavalla catch-lohkolla" tarkoitan lohkoa, joka on määritelty käsittelemään aiheutetun tyyppisen poikkeuksen. Seuraavassa on esimerkki metodista (*Foo*), joka kutsuu ja ottaa kiinni toisen metodin (*Bar*) aiheuttaman poikkeuksen:

```
public void Foo()
{
    try
    {
        Bar();
    }
    catch(System.Exception e)
    {
        // Tehdään virhetilanteessa jotain.
    }
}
```

Saatat nyt kysyä, "Mitä tapahtuu, jos *Bar* aiheuttaa poikkeuksen ja *Foo* ei ota sitä kiinni?" (Tällainen tilanne on myös, jos *Bar*-metodin kutsu ei ole *try*-lohkossa.) Vastaus riippuu sovelluksen suunnittelusta. Kun poikkeus aiheutetaan, ohjelman suoritus palautetaan kutsupinoon, kunnes löytyy metodi, jossa on aiheutetun poikkeuksen tyyppinen *catch*-lohko. Jos vastaavaa *catch*-lohkoa sisältävää metodia ei löydy, sovellus kaatuu. Jos metodi kutsuu toista metodia, joka aiheuttaa poikkeuksen, pitää sovelluksen suunnitelman olla sellainen, että jokin kutsupinossa oleva metodi käsittelee poikkeuksen, jotta sovellus ei kaatuisi.

Poikkeuksen jatkaminen

On tilanteita, joissa metodi on ottanut poikkeuksen kiinni ja tehnyt kaiken, mitä se ympäristössään voi ja sen jälkeen jatkaa (rethrow) poikkeuksen takaisin kutsupinoon. Tämä tehdään hyvin helposti avainsanalla *throw*. Tässä esimerkki:

```
using System;

class RethrowApp
{
    static public void Main()
    {
        Rethrow rethrow = new Rethrow();

        try
        {
            rethrow.Foo();
        }
        catch(Exception e)
        {
            Console.WriteLine(e.Message);
        }
    }

    public void Foo()
    {
        try
        {
            Bar();
        }
        catch(Exception)
        {
            // Handle error.
            throw;
        }
    }

    public void Bar()
    {
        throw new Exception("thrown by Rethrow.Bar");
    }
}
```

Tässä esimerkissä *Main* kutsuu *Foo*-metodia, joka kutsuu *Bar*-metodia. *Bar* aiheuttaa poikkeuksen, jonka *Foo* ottaa kiinni. *Foo* tekee joitakin toimenpiteitä ja sen jälkeen jatkaa poikkeuksen takaisin kutsupinoon *Main*-metodille käyttämällä *throw*-avainsanaa.

Siivoaminen *finally*-lohkossa

Yksi hankala asia poikkeusten käsittelyssä on varmistaa, että määrätty koodi suoritetaan takuuvarmasti riippumatta siitä, onko poikkeus otettu kiinni vai ei. Oletetaan esimerkiksi, että varaat laitteen tai tiedoston tapaisen resurssin. Sitten avaat resurssin ja kutsut metodia, joka aiheuttaa poikkeuksen. Riippumatta siitä, voiko metodisi jatkaa resurssin käyttämistä,

sinun pitää silti vapauttaa tai sulkea se. Tällöin sinun pitää käyttää *finally*-avainsanaa seuraavalla tavalla:

```
using System;

public class ThrowException1App
{
    public static void ThrowException()
    {
        throw new Exception();
    }

    public static void Main()
    {
        try
        {
            Console.WriteLine("try...");
        }
        catch(Exception e)
        {
            Console.WriteLine("catch...");
        }
        finally
        {
            Console.WriteLine("finally");
        }
    }
}
```

Kuten näet, *finally*-avainsanan ansiosta sinun ei tarvitse kirjoittaa puhdistuskoodia sekä *catch*-lohkoon että *try/catch*-lohkojen perään. Nyt, riippumatta siitä, onko poikkeusta aiheutettu, *finally*-lohkoon kirjoitettu koodi suoritetaan.

Virheenkäsittelytekniikoiden vertailua

Nyt kun olet nähnyt perusasiat poikkeuksen aiheuttamisesta ja kiinniottamisesta, käytetään muutama minuutti vertailemalla eri ohjelmointikielissä toteutettuja erilaisia menetelmiä virheiden käsittelyyn.

Yleinen tapa virheiden käsittelyssä on ollut virhekoodin palauttaminen kutsuvalle metodille. Kutsuvalle metodille on silloin jätetty vastuu paluuarvon tulkinnasta ja sen mukaan toimimisesta. Paluuarvo voi olla yksinkertaisimmillaan C:n tai C++:n perustyyppi tai se voi olla osoitin monipuoliseen objektiin, joka sisältää kaiken tarpeellisen informaation virheen ymmärtämiseen. Yksityiskohtaisemmin toteutetuissa virheenkäsittelymenetelmissä

on kokonainen alijärjestelmä virheille, jolle kutsuttu metodi ilmaisee virhetilanteen ja palauttaa virhekoodin kutsujalle. Kutsuja kutsuu sitten virheenkäsittelyn alijärjestelmän yleistä metodia selvittääkseen viimeisen järjestelmään raportoidun virheen aiheuttajan. Esimerkki tällaisesta rakaisusta on Microsoft Open Database Connectivity (ODBC) SDK. Täsmällisestä toteutustavasta riippumatta perusajatus säilyy samana: metodi kutsuu jollakin tavalla metodia ja tutkii sen palauttamaa arvoa saadakseen selville metodin toiminnon onnistumisen. Tällä menetelmällä, vaikka se onkin ollut vuosia yleisesti käytössä, on vakavia puutteita monella tapaa. Seuraavassa kappaleessa kuvailen muutamia tapoja, jolla poikkeuksen käsittely tarjoaa valtavia etuja paluukoodien käyttämiseen verrattuna.

Poikkeusten käsittelyn edut paluuarvoon verrattuna

Kun käytetään paluuarvoja, kutsuttu metodi palauttaa virhekoodin ja kutsuva metodi käsittelee virhetilanteen. Koska virheenkäsittely tapahtuu kutsutun metodin näkyvyysalueen ulkopuolella, ei ole mitään varmuutta, että kutsuja tarkistaa palautetun virhekoodin. Esimerkkinä ajatellaan, että kirjoitan luokkaa nimeltä *CommaDelimitedFile*, johon sisältyy erotinmerkin sisältävien tekstitiedostojen lukeminen ja kirjoittaminen. Luokkasi tulee sisältää metodit tiedoston avaamiseen ja tietojen lukemiseen avatusta tiedostosta. Vanhan virhekoodin palauttavan menetelmän käyttäminen tarkoittaisi, että nämä metodit palauttaisivat jonkin muuttujatyypin, jonka kutsujan tulee tarkistaa selvittääkseen metodikutsun onnistumisen. Jos luokkasi käyttäjä kutsuu *CommaDelimitedFile.Open*-metodia ja sen jälkeen yrittää kutsua *CommaDelimitedFile.Read*-metodia tarkistamatta *Open*-kutsun onnistumista, tämä voi (ja luultavasti aiheuttaakin ns. demoefektin tärkeän asiakkaan edessä) aiheuttaa vähemmän odotettuja tuloksia. Jos luokan *Open*-metodi aiheuttaisi poikkeuksen, kutsuja pakotettaisiin sopeutumaan siihen tosiasiaan, että *Open*-metodi epäonnistui. Tämä johtuu siitä, että joka kerta, kun metodi aiheuttaa virheen, ohjelman suoritus palautetaan takaisin kutsupinoon, kunnes se otetaan kiinni. Seuraavassa esimerkki siitä, miltä tällainen koodi voisi näyttää:

```
using System;

class ThrowException2App
{
    class CommaDelimitedFile
    {
        protected string fileName;

        public void Open(string fileName)
        {
            this.fileName = fileName;
```

(jatkuu)

```
            // Yritetty avata tiedosto ja aiheutettu
            // poikkeus virhetilanteessa.
            throw new Exception(“open failed”);
        }

        public bool Read(string record)
        {
            // Code to read file.
            return false; // EOF
        }
    }

    public static void Main()
    {
        try
        {
            Console.WriteLine(“attempting to open file”);

            CommaDelimitedFile file = new CommaDelimitedFile();
            file.Open(“c:\\test.csv”);

            string record = “";

            Console.WriteLine(“reading from file”);

            while (file.Read(record) == true)
            {
                Console.WriteLine(record);
            }

            Console.WriteLine(“finished reading file”);
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
    }
}
```

Jos esimerkissä joko *CommaDelimitedFile.Open*-metodi tai *CommaDelimitedFile.Read*-metodi aiheuttaa poikkeuksen, kutsuva metodi on pakotettu huomioimaan se. Jos kutsuva metodi ei ota sitä kiinni eikä mikään muukaan metodi kutsupinossa ota tämän tyyppistä virhettä kiinni, sovellus kaatuu. Huomaa erityisesti se tosiasia, että koska *Open*-metodin kutsu on sijoitettu *try*-lohkoon, virheellistä lukua (käyttäen esimerkkiämme, jossa *Open*-metodi on aiheuttanut poikkeuksen) ei voida edes yrittää. Tämä johtuu siitä, että ohjelman

suoritus siirtyy *try*-lohkon *Open*-metodin kutsusta *catch*-lohkon ensimmäiselle riville. Siten yksi poikkeustenkäsittelyn suurimmista eduista verrattuna virhekoodin käyttämiseen on se, että poikkeuksia on vaikeampi ohjelmallisesti olla huomioimatta.

Virheiden käsitteleminen oikeassa ympäristössä

Hyvän ohjelmointikäytännän yleinen periaate on tiukka koheesio, joka viittaa metodin tavoitteeseen tai tarkoitukseen. Metodit, jotka noudattavat tätä periaatetta, suorittavat vain yhden tehtävän. Tiukan koheesin suurin etu saavutetaan siinä, että metodi on helpommin siirrettävissä ja käytettävissä eri yhteyksissä, koska se suorittaa vain yhden toiminnon. Yhden toiminnon suorittava metodi on myös helpompi testata ja ylläpitää. Tiukka koheesio aiheuttaa kuitenkin yhden ison ongelman, joka liittyy virheenkäsittelyyn. Katsotaan tätä ongelmaa kuvaavaa esimerkkiä ja sitä, miten poikkeuksen käsittely ratkaisee sen.

Tässä esimerkissä luokkaa (*AccessDatabase*) käytetään generoimaan ja käsittelemään Microsoft Access -tietokantoja. Luokalla on staattinen metodi *GenerateDatabase*. Koska *GenerateDatabase*-metodia käytetään Access-tietokannan luomiseen, sen tulee suorittaa useita erillisiä tehtäviä. Sen tulee esimerkiksi luoda varsinainen tietokantatiedosto levylle, luoda määrätyt taulut (ja sovelluksesi tarvitsemat sarakkeet ja rivit) ja määritellä tarvittavat indeksit ja suhteet. *GenerateDatabase*-metodi voi jopa joutua luomaan oletuskäyttäjät ja oikeudet.

Ohjelman ongelma on seuraava: jos virhe tapahtuu *CreateIndexes*-metodissa, minkä metodin tulee käsitellä se ja miten? Metodin, joka alun perin kutsui *GenerateDatabase*-metodia, tulee tietenkin käsitellä virhe, mutta miten se voi sen tehdä? Sillä ei ole aavistustakaan, miten tulee käsitellä virhe, joka tapahtui kutsupinossa muutamia metodeja alempana. Kuten olemme nähneet, kutsuvan metodin sanotaan olevan väärässä ympäristössä käsitelläkseen virheen. Toisin sanoen, ainoa metodi, joka pystyy luomaan asiallisen informaation virheestä, on metodi, jonka suoritus ei onnistunut. Täten, jos *AccessDatabase*-luokassamme käytetään virhekoodeja, kunkin kutsupinon metodin tulee tarkistaa jokainen virhekoodin, jonka mikä tahansa kutsupinon muista metodeista voi palauttaa. Selvä ongelma tässä on se, että kutsuvan metodin tulee mahdollisesti käsitellä valtavan suuri määrä virhekoodeja. Lisäksi ylläpito on vaikeaa. Joka kerta, kun johonkin kutsupinon metodiin lisätään virhekoodi, niin jokaiseen sovelluksen osaan, jossa tuota metodia kutsutaan, tulee päivittää uuden virhekoodin käsittelyosuus. Tarpeetonta sanoakaan, että tämä on kovin tehotonta ajatellen sovelluksen elinaikana muodostuneita kustannuksia.

Poikkeuksen käsittely ratkaisee kaikki nämä ongelmat siksi, että se mahdollistaa kutsuvan metodin ottavan kiinni halutun tyyppiset poikkeukset. Jos käyttämässämme esimerkissä luokka *AccessDatabaseException* olisi periytynyt luokasta *Exception*, sitä voitaisiin käyttää kaiken tyyppisiin poikkeuksiin, joita *AccessDatabase*-metodin kutsumissa metodeissa tapahtuu. (Puhun *Exception*-luokasta ja omien luokkien periyttämisestä siitä myöhemmin tässä luvussa kappeleessa "*System.Exception*-luokan käyttäminen.") Jos nyt *CreateIndexes*-metodi epäonnistuu, voimme rakentaa ja heittää *AccessDatabseException*-tyyppisen poikkeuksen. Kutsuva metodi voi ottaa sen kiinni ja tutkia *Exception*-objektista, mikä tarkasti ottaen meni pieleen. Sen sijaan, että käsittelisi kaikki mahdolliset paluukoodit, jotka *GenerateDatbase* tai jokin sen kutsumista metodeista voi palauttaa, kutsuva metodi voi olla varma, että jos jokin sen kutsupinossa olevista metodeista epäonnistui, palautetaan tarpeellinen virheinformaatio. Poikkeuksen käsittely tarjoaa vielä lisäedun: koska virheinformaatio sijaitsee luokassa, uusia virhetilanteita voidaan lisätä, mutta kutsuvaa metodia ei tarvitse muuttaa. Ja eikö tämä (mahdollisuus tehdä jotain uutta ja lisätä se ilman vanhan koodin muuttamista tai rikkomista) ollutkin yksi alkuperäisistä olioperusteisen ohjelmoinnin lupauksista? Tämän vuoksi virheiden kiinniottaminen ja käsitteleminen oikeassa ympäristössä on merkittävin etu poikkeusten käsittelyn käyttämisestä

Koodin luettavuuden paraneminen

Kun käytetään poikkeuksen käsittelykoodia, paranee koodin luettavuus huomattavasti. Tämä puolestaan vähentää kuluja koodin ylläpidossa. Tapa, miten koodissa käsitellään paluukoodeja verrattuna poikkeusten käsittelyyn, on syy tähän luettavuuden paranemiseen. Jos käyttäisit paluukoodeja aiemmin mainitussa *AccessDatabase.GenerateDatbase*-metodissa, tarvitsisit seuraavan esimerkin tapaista koodia virhetilanteiden käsittelemiseksi:

```
public bool GenerateDatabase()
{
    if (CreatePhysicalDatabase())
    {
        if (CreateTables())
        {
            if (CreateIndexes())
            {
                return true;
            }
            else
            {
                // Käsittele virhe.
                return false;
            }
```

```
        }
        else
        {
            // Käsittele virhe.
            return false;
        }
    }
    else
    {
        // Käsittele virhe.
        return false;
    }
}
```

Lisää vielä edellä olevaan koodiin muuta tarkistuskoodia ja sinulla on ääretön määrä virhetarkistuksia suorittavaa koodia sekaisin liiketoimintalogiikan toteuttavan koodin kanssa. Jos sisennät koodiasi 4 välilyönnillä lohkoa kohden, saattavat ensimmäiset merkit joillakin riveillä olla vasta sarakkeessa 20 tai vieläkin kauempana. Tämä ei ole tuhoisaa itse koodin kannalta, mutta se tekee siitä vaikean lukea ja ylläpitää ja vanha totuushan on, että vaikeasti ylläpidettävä koodi on hyvä alusta virheille. Katsotaanpa, miltä tämä sama esimerkki näyttäisi, jos olisi käytetty poikkeusten käsittelyä:

```
// Kutsuva koodi.
try
{
    AccessDatabase accessDb = new AccessDatabase();
    accessDb.GenerateDatabase();
}
catch(Exception e)
{
    // Otetaan poikkeus kiinni.
}

// AccessDatabase.GenerateDatabase-metodin määrittely.
public void GenerateDatabase()
{
    CreatePhysicalDatabase();
    CreateTables();
    CreateIndexes();
}
```

Huomaa, miten paljon "puhtaampi" ja tyylikkäämpi toinen ratkaisu on. Tämä johtuu siitä, että virheen tunnistus- ja käsittelykoodia ei enää sekoiteta kutsuvan koodin logiikkaan. Koska poikkeusten käsittely on tehnyt koodista suoraviivaisemman, on myös sen ylläpito tullut helpommaksi.

Poikkeuksen aiheuttaminen muodostimissa

Toinen merkittävä etu poikkeusten käsittelyllä muihin virheenkäsittelymenetelmiin verrattuna on objektien luonnissa. Koska muodostimella ei voi olla paluuarvoa, ei ole helppoa ja järkevää tapaa ilmoittaa muodostinta kutsuneelle metodille, että objektin luonnin aikana tapahtui virhe. Poikkeuksia voidaan kuitenkin käyttää, koska kutsuvan metodin pitää vain "kääriä" objektin luonti *try*-lohkoon, kuten seuraavassa koodissa on tehty:

```
try
{
    // Jos AccessDatabase-objektin luonti eäonnistui ja
    // aiheutti poikkeuksen, se voidaan ottaa kiinni.
    AccessDatabase accessDb = new AccessDatabase();
}
catch(Exception e)
{
    // Otetaan poikkeus kiinni.
}
```

System.Exception-luokan käyttäminen

Kuten aiemmin mainitsin, kaikkien aiheutettavien poikkeuksien pitää olla tyyppiä *System.Exception* (tai periytetty siitä). Itse asiassa *System.Exception*-luokka on niiden useiden eri poikkeusluokkien kantaluokka, joita voit C#-koodissasi käyttää. Useimmat *System.Exception*-luokasta periytyvät luokat eivät lisää toiminnallisuutta kantaluokkaan. Miksi siis C# vaivautuu periyttämään luokkia, jos ne eivät merkittävästi eroa kantaluokastaan? Syy on se, että yhdessä *try*-lohkossa voi olla useita *catch*-lohkoja, joista kukin on määritelty erityyppiselle poikkeukselle. (Näet tämän kohta.) Tämän avulla koodisi voi käsitellä erityyppiset poikkeukset kullekin tyypille parhaiten sopivalla tavalla.

Exception-objektin luominen

Tämän kirjoitushetkellä *System.Exception*-luokalla on neljä erilaista muodostinta:

```
public Exception ();
public Exception(String);
protected Exception(SerializationInfo, StreamingContext);
public Exception(String, Exception);
```

Ensimmäinen muodostin yllä olevassa luettelossa on oletusmuodostin. Sille ei välitetä parametreja ja se asettaa kaikki jäsenensä oletusarvoihin. Tämä poikkeus aiheutetaan seuraavasti:

```
// Virhetilanne.
throw new Exception();
```

Toinen muodostin saa ainoana parametrinaan *String*-tyyppisen arvon, joka esittää virheilmoitusta. Tämän kirjan esimerkeissä näet yleensä käytettävän tätä muotoa. Parametrina annettu virheilmoitus voidaan hakea poikkeuksen kiinniottaneessa koodilohkossa *System.Exception.Message*-ominaisuudesta. Tässä yksinkertainen esimerkki sekä poikkeuksen aiheuttamisesta käyttäen kakkosmuodostinta sekä sen kiinniottamisesta ja paramatrina annetun tekstin näyttämisestä.

```
using System;

class ThrowException3
{
    class FileOps
    {
        public void FileOpen(String fileName)
        {
            // Poikkeuksen  aiheuttaminen
            throw new Exception(“Oh bother”);
        }

        public void FileRead()
        {
        }
    }

    public static void Main()
    {
        // Koodi poikkeuksen kiinniottoon.
        try
        {
            FileOps fileOps = new FileOps();

            fileOps.FileOpen(“c:\\test.txt”);
            fileOps.FileRead();
        }
        catch(System.Exception e)
        {
            Console.WriteLine(e.Message);
        }
    }
}
```

Kolmas muodostin alustaa *Exception*-luokan instanssin serialisoiduilla (tallennetuilla) tiedoilla.

Viimeinen eli neljäs muodostin antaa sinun määritellä virheilmoituksen lisäksi ns. *sisäisen poikkeuksen* (inner exception). Syy tähän on se, että kun käsittelet poikkeuksia, haluat

joskus muuttaa poikkeusta jollakin kutsupinon tasolla hieman ennen kuin palautat sen ylemmälle tasolle. Katsotaan esimerkkiä siitä, miten voit käyttää tällaista mahdollisuutta.

Sanotaan, että helpottaaksesi luokkasi käyttäjien taakkaa, päätät aiheuttaa vain yhden tyyppisiä poikkeuksia. Siten luokan käyttäjän pitää ottaa kiinni vain yksi poikkeus ja viitata sen *InnerException*-ominaisuuteen. Tässä on se lisäetu, että jos päätät muokata metodia aiheuttamaan uuden tyyppisen poikkeuksen sen jälkeen, kun luokkaa käyttävä koodi on kirjoitettu, koodia ei tarvitse muuttaa kuin siinä tapauksessa, että uuden tyyppisen poikkeuksen tapahtuessa halutaan tehdä jotain erikoista.

Huomaa seuraavassa esimerkissä, että luokkaa käyttävä koodi ottaa kiinni ylimmän tason *System.Exception*-objektin ja tulostaa sen sisäisen poikkeuksen virheilmoituksen. Jos *DoWork*-metodi joskus myöhemmin aiheuttaa eri tyyppisen poikkeuksen (niin kauan kuin se tekee sen *System.Exception*-objektin sisäisenä poikkeuksena), luokkaa käyttävä koodi toimii muutoksitta.

```
using System;
using System.Globalization;

class FooLib
{
    protected bool IsValidParam(string value)
    {
        bool success = false;

        if (value.Length == 3)
        {
            char c1 = value[0];
            if (Char.IsNumber(c1))
            {
                char c2 = value[2];
                if (Char.IsNumber(c2))
                {
                    if (value[1] == '.')
                        success = true;
                }
            }
        }

        return success;
    }
```

```
    public void DoWork(string value)
    {
        if (!IsValidParam(value))throw new Exception
                ("", new FormatException("Invalid parameter specified"));
        Console.WriteLine("Work done with '{0}'", value);
    }
}

class FooLibClientApp
{
    public static void Main(string[] args)
    {
        FooLib lib = new FooLib();
        try
        {
            lib.DoWork(args[0]);
        }
        catch(Exception e)
        {
            Exception inner = e.InnerException;
            Console.WriteLine(inner.Message);
        }
    }
}
```

StackTrace-ominaisuuden käyttäminen

Toinen käyttökelpoinen *System.Exception*-luokan ominaisuus on *StackTrace*. Sen avulla voit aina, kun sinulla on käytössäsi kelvollinen *System.Exception*-objekti, määritellä sen hetkisen kutsupinon sisällön. Katsotaan seuraavaa koodia:

```
using System;

class StackTraceTestApp
{
    public void Open(String fileName)
    {
        Lock(fileName);
        // ...
    }
    public void Lock(String fileName)
```

(jatkuu)

```
    {
        // Virhetilanne.
        throw new Exception("failed to lock file");
    }
    public static void Main()
    {
        StackTraceTestApp test = new StackTraceTestApp();

        try
        {
            test.Open("c:\\test.txt");
            // Tiedoton käyttö.
        }
        catch(Exception e)
        {
            Console.WriteLine(e.StackTrace);
        }
    }
}
```

Tämä esimerkki tulostaa seuraavaa:

```
at StackTraceTest.Main()
```

Koska *StackTrace*-ominaisuus palauttaa kutsupinon sillä hetkellä, kun poikkeus otettiin kiinni, se on hyvin käyttökelpoinen virheenetsintätilanteissa.

Useiden poikkeustyyppien kiinniottaminen

Eteesi saattaa tulla tilanteita, että haluat *try*-lohkon ottavan kiinni eri tyyppisiä poikkeuksia. Esimerkiksi määrätyn metodin on dokumentoitu aiheuttavan useita erilaisia poikkeutyyppejä tai haluat tehdä useita metodikutsuja yhdessä *try*-lohkossa, joista jokainen aiheuttaa eri tyyppisen poikkeuksen. Tällainen tilanne hoidetaan lisäämällä *catch*-lohko jokaista käsiteltävää poikkeustyyppiä kohden:

```
try
{
    Foo(); // Voi  aiheuttaa FooException-poikkeuksen.
    Bar(); // Voi  aiheuttaa BarException-poikkeuksen.
}
catch(FooException e)
{
    // Käsitellään virhe.
}
catch(BarException e)
{
    // Käsitellään virhe.
```

```
}
catch(Exception e)
{
}
```

Kukin poikkeustyyppi voidaan nyt käsitellä omassa erillisessä *catch*-lohkossaan. Äärimmäisen tärkeä seikka huomata tässä on se, että kantaluokka käsitellään viimeiseksi. Kaikki poikkeuksen periytyvät luokasta *System.Exception* ja jos menet sijoittamaan sen *catch*-lohkon ensimmäiseksi, muihin *catch*-lohkoihin ei ikinä päästä. Tästä johtuen kääntäjä ei hyväksy seuraavaa koodia:

```
try
{
    Foo(); // Voi  aiheuttaa FooException-poikkeuksen.
    Bar(); // Voi  aiheuttaa BarException-poikkeuksen.
}
catch(Exception e)
{
    // ***VIRHE - TÄMÄ EI KÄÄNNY
}
catch(FooException e)
{
    // Käsitellään virhe.
}
catch(BarException e)
{
    // Käsitellään virhe.
}
```

Omien *Exception*-luokkien periyttäminen

Kuten sanoin, saattaa tulla tilanteita, jolloin haluat tarjota ylimääräistä informaatiota tai muokata poikkeusta ennen kuin heität sen asiakasohjelmalle. Haluat ehkä tarjota oman poikkeusten kantaluokan myös siksi, että luokkasi aiheuttaisi vain yhden tyyppisiä poikkeuksia Siten luokkasi käyttäjän pitää varautua vain tämän kantaluokan tyyppisten poikkeusten kiinniottamiseen.

Toinen esimerkki tilanteesta, jossa mahdollisesti haluat periyttää oman Exception-luokkasi, on silloin, jos haluat suorittaa joitain toimenpiteitä (esimerkiksi tehdä lokimerkinnän tai lähettää sähköpostiviestin tukihenkilölle) joka kerta, kun poikkeus aiheutetaan. Tässä tapauksessa perit oman Exception-luokkasi ja kirjoitat tarvittavan koodin luokan muodostimeen, esimerkiksi näin:

```
using System;

public class TestException : Exception
```

(jatkuu)

```
{
    // Voit kirjoittaa omia metodeja ja ominaisuuksia tähän
    // niiden lisäksi, jotka perit Exception-kantaluokan
    // muodostimesta.
    public TestException()
        :base() {}
    public TestException(String message)
        :base(message) {}
    public TestException(String message, Exception innerException)
        :base(message, innerException) {}
}
public class DerivedExceptionTestApp
{
    public static void ThrowException()
    {
        throw new TestException("error condition");
    }
    public static void Main()
    {
        try
        {
            ThrowException();
        }
        catch(Exception e)
        {
            Console.WriteLine(e.ToString());
        }
    }
}
```

Omien Exception-luokkien periyttäminen

On hyvän ohjelmointitavan mukaista (ja yhdenmukaista useimpien näkemiesi C#-koodien kanssa), vaikkakaan ei mikään ehdoton sääntö, että nimeät poikkeusluokkasi siten, että se päättyy sanaan "Exception." Jos esimerkiksi haluat periyttää Exception-luokan luokalle *MyFancyGraphics*, anna sille nimeksi *MyFancyGraphicsException*.

Toinen nyrkkisääntö periytettäessä omia poikkeusluokkia on se, että toteutat kaikki kolme *System.Exception*-luokan muodostinta. Tämä ei ole ehdottoman välttämätöntä, mutta se lisää yhdenmukaisuutta niitten C#-koodien kanssa, joita asiakkaasi käyttävät.

Tämä koodi tulostaa seuraavaa. Huomaa, että *ToString*-metodi muodostaa yhdistelmän objektin ominaisuuksista: merkkijonon, joka sisältää poikkeusluokan nimen, merkkijonon, joka välitettiin parametrina luokan instantioinnissa ja *StackTrace*-ominaisuuden.

```
TestException: error condition
   at DerivedExceptionTestApp.Main()
```

Poikkeusten käsittelyn suunnittelu

Tähän mennessä olemme käsitelleet poikkeusten käsittelyn periaatteet ja menetelmät, joilla poikkeuksia aiheutetaan ja otetaan kiinni. Katsotaan nyt yhtä tärkeää poikkeusten käsittelyn puolta: poikkeusten käsittelyn ottaminen huomioon sovelluksen suunnittelussa. Oletetaan, että sinulla on kolme metodia: *Foo*, *Bar* ja *Baz*. *Foo* kutsuu *Bar*-metodia, joka puolestaan kutsuu *Baz*-metodia. Jos *Baz* ilmoittaa, että se on aiheuttanut poikkeuksen, tuleeko *Bar*-metodin ottaa se kiinni, vaikka se ei pysty tekemään sille mitään? Miten koodi tulee jakaa ottaen huomioo *try*- ja *catch*-lohkot?

try-lohkon suunnitteluperiaatteet

Tiedät, miten otetaan kiinni kutsuvan metodin mahdollisesti aiheuttama poikkeus ja tiedät, että ohjelman suoritus siirtyy kutsupinossa ylöspäin kunnes vastaava *catch*-lohko löytyy. Kysymys kuuluukin: tuleeko *try*-lohkon ottaa kiinni jokainen mahdollinen poikkeus, jonka siihen sisältyvä metodi on aiheuttanut? Vastaus on ei, jotta poikkeusten käsittelyn suurimmat edut (vähentynyt koodaus ja edullisempi ylläpito) sovelluksesi kannalta tulisi hyödynnettyä. Seuraava esimerkki esittää ohjelmaa, jossa poikkeus on jatkettu käsiteltäväksi toisessa *catch*-lohkossa.

```
using System;

class WhenNotToCatchApp
{
    public void Foo()
    {
        try
        {
            Bar();
        }
        catch(Exception e)
        {
            Console.WriteLine(e.Message);
        }
```

(jatkuu)

```
    }

    public void Bar()
    {
        try
        {
            Baz();
        }
        catch(Exception e)
        {
// Bar ei tee tälle poikkeukselle
// mitään, vaan jatkaa sen eteenpäin

throw;
        }
    }

    public void Baz()
    {
        throw new Exception("Exception originally thrown by Baz");
    }

    public static void Main()
    {
        WhenNotToCatchApp test = new WhenNotToCatchApp();
        test.Foo(); // Tämä metodi tulostaa lopulta
                    // virheilmoituksen.
    }
}
```

Tässä esimerkissä *Foo* ottaa kiinni poikkeuksen, jonka *Baz* aiheutti, vaikka se ei tee sille muuta kuin jatkaa sen eteenpäin *Bar*-funktiolle. *Bar* ottaa vuorostaan kiinni jatketun poikkeuksen ja tekee sen informaatiolla jotain eli tulostaa *Exception*-objektin *message*-ominaisuuden. Seuraavassa muutama syy, miksi metodien, jotka jatkoivat poikkeusta eteenpäin, ei tulisi sitä edes ottaa kiinni:

- Koska metodi ei tehnyt poikkeuksella mitään, sinulla on jäljellä *catch*-lohko, joka on vähintään ylimääräinen. Koskaan ei kannata jättää sovellukseen koodia, jolla ei ole mitään varsinaista tehtävää.
- Jos *Baz*-metodin aiheuttaman poikkeuksen tyyppi tulee muuttumaan, sinun pitää muuttaa sekä *Bar*-metodin että *Foo*-metodin *catch*-lohko. Miksi pistät itsesi tilanteeseen, jossa joudut muuttamaan koodia, joka ei tee mitään?

Koska CLR automaattisesti jatkaa kutsupinoa ylöspäin, kunnes löytää metodin, joka ottaa poikkeuksen kiinni, välitason metodit voivat "kieltäytyä" poikkeuksista, joita ne eivät voi käsitellä. Kunhan varmistat, että jokin metodi ottaa poikkeuksen kiinni, koska kuten aiemmin mainitsin, jos poikkeus on aiheutettu eikä kutsupinossa ole yhtäkään *catch*-lohkoa, joka sen käsittelisi, koko sovellus kaatuu.

catch-lohkon suunnitteluperiaatteet

Ainoa koodi, jonka pitää olla *catch*-lohkossa, on koodi, joka ainakin osittain käsittelee kiinni otetun poikkeuksen. Kun metodi ottaa poikkeuksen kiinni, tee mitä se poikkeuksen käsittelemiseksi pystyy ja jatka poikkeusta, jotta myöhempi virheenkäsittely voi tehdä loput. Seuraava esimerkki kuvaa tätä. *Foo* on kutsunut *Bar*-metodia, joka tekee tietokannassa jotain toimintoja *Foo*-metodin puolesta. Koska käytetään tapahtumakäsittely-ominaisuuksia, *Bar*-metodin pitää ottaa kiinni jokainen tapahtuva virhe ja peruuttaa tietokantaan tehdyt muutokset, ennen kuin jatkaa poikkeuksen takaisin *Foo*-metodille.

```
using System;

class WhenToCatchApp
{
    public void Foo()
    {
        try
        {
            Bar();
        }
        catch(Exception e)
        {
            Console.WriteLine(e.Message);
        }
    }

    public void Bar()
    {
        try
        {
            // Käynnistetään tapahtuma (transaction)
            Console.WriteLine(“setting commitment boundary”);

            // Pyydetään Baz-metodia tallentamaan tiedot.
            Console.WriteLine(“calling Baz to save data”);
            Baz();

            Console.WriteLine(“commiting saved data”);
        }
```

(jatkuu)

```
        catch(Exception)
        {
                // Tässä tapauksessa Bar'n tulee ottaa
                // poikkeus kiinni, koska sen pitää
                // peruuttaa tietokantamuutokset.

                Console.WriteLine("rolling back uncommited changes " +
                         "and then rethrowing exception");

            throw;
        }
    }

    public void Baz()
    {
        throw new Exception("db failure in Baz");
    }

    public static void Main()
    {
        WhenToCatchApp test = new WhenToCatchApp();
        test.Foo(); // Tämä metodi tulostaa lopulta
                    // virheilmoituksen.
    }
}
```

Bar-metodn pitää ottaa poikkeus kiinni tehdäkseen oman virheenkäsittelynsä. Kun se on tehty, se palauttaa poikkeuksen kutsupinoon, missä *Foo* ottaa sen kiinni ja tekee oman virhekäsittelynsä. Tämä mahdollisuus tehdä kullakin kutsupinon tasolla sen verran virheenkäsittelyä kuin on mahdollista, on yksi seikka, joka osoittaa, miten tarpeellista poikkeusten käsittelyn huomioiminen koko sovelluksen suunnittelussa on.

Yhteenveto

C#:n poikkeusten käsittelyn syntaksi on yksinkertainen ja suoraviivainen ja poikkeusten käsittelyn toteuttaminen sovelluksissa on yhtä helppoa kuin metodien suunnittelu. .NET CLR:n ensisijainen päämäärä on auttaa estämään ajonaikaiset virheet vahvan tyyppijärjestelmän avulla ja käsitellä sulavasti ajonaikaiset virheet. Poikkeusten käsittelykoodin käyttäminen tekee sovelluksesta vakaamman, luotettavamman ja lopulta helpomman ylläpitää.