

## X. Kivételkezelés

A C# nyelvben és általában a .NET keretrendszerben a hibajelzés és kezelés széles körben alkalmazott formája a kivételek előidézése és elkapása/feldolgozása. Az alábbiakban megismerkedünk a nem kezelt kivétel fogalmával, a kivételek feldolgozási és továbbítási lehetőségeivel valamint előidézésükkel.

Bevezetésképpen tekintsünk egy kis programot, amelyben bekérünk két egész számot a konzolról, majd kiszámítjuk összegüket és megjelenítjük az eredményt a konzolablakban.

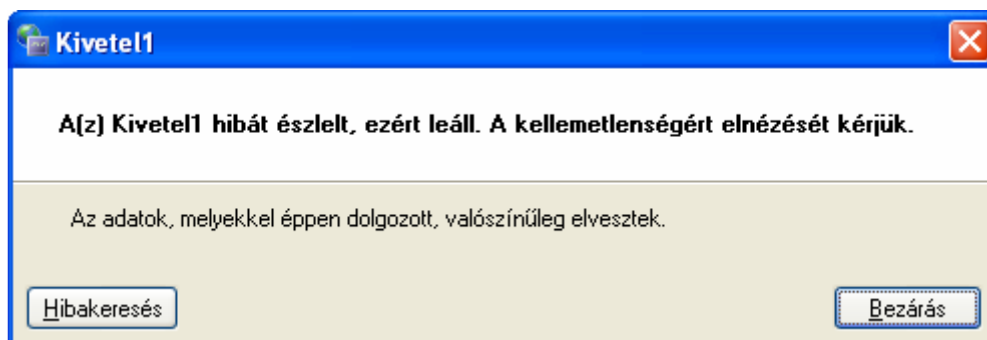
### X.1. Nem kezelt kivétel

A feladat első megoldását az X.1. ábrán látható kódrészlet tartalmazza. Az egyszerűség kedvéért az adatok beolvasását megvalósító metódust (Beolvas) statikus osztálytagnak választottuk.

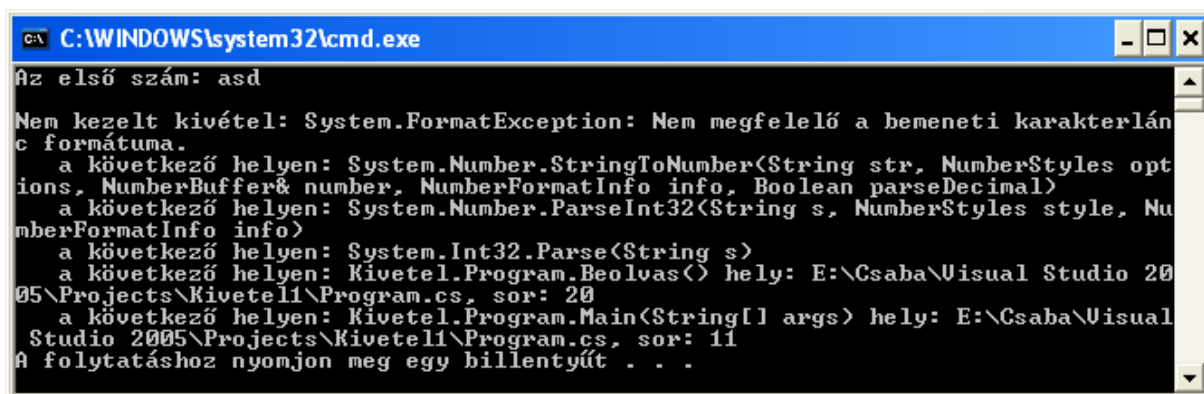
```
class Program
{
    static void Main(string[] args)
    {
        int a = Beolvas("z első");
        int b = Beolvas(" második");
        int c = a + b;
        Console.WriteLine("A két szám összege: {0}", c);
    }
    static int Beolvas(string Aktuális)
    {
        Console.Write("A"+Aktuális+" szám: ");
        string s = System.Console.ReadLine();
        int a = int.Parse(s);
        return a;
    }
}
```

X.1. ábra. Adatbeolvasás kivételek kezelése nélkül

Indítsuk el a fejlesztőrendszerben az alkalmazást (Ctrl+F5), és adjunk meg egy érvénytelen értéket, például egy betűsort. Némi várakozást követően egy figyelmeztető üzenetet kapunk (X.2. ábra), majd ennek bezárása után az X.3. ábrán látható hibaüzenet jelenik meg a konzolon, ami a nem megfelelő bemeneti karakterláncra figyelmeztet.

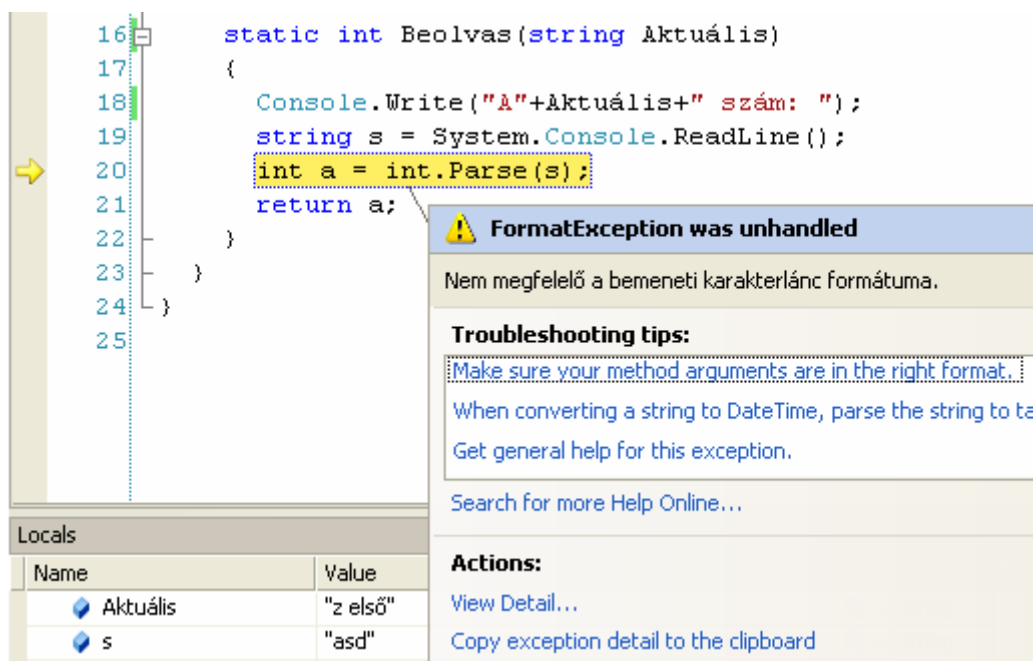


X.2. ábra.



X.3. ábra. Hibüzenet a konzolon nem kezelt kivétel esetén

Kicsit eltérő eredményt kapunk akkor, ha debug módban indítjuk (F5) alkalmazásunkat. A hibás adatok megadása után a kódszerkesztőben az érintett utasítást sárga háttérrel kiemelve jelenik meg az Exception Assistant nem kezelt kivételre figyelmeztető ablaka (X.4. ábra).



X.4. ábra. Nem kezelt kivétel debug módban

Itt is kaphatunk részletes információt a hibával kapcsolatosan a View Detail... felírra kattintva. Az egyes változók értékeit ellenőrizhetjük a fejlesztőrendszer Locals és Watch ablakaiban.

Mi is történt valójában? A fenti két esetben az *s* változóban karakterláncként tárolt adatot a *Parse* metódus megpróbálta egész számmá alakítani, és a sikertelenséget egy kivételes esemény előidézésével jelezte. Emellett egy objektumot is létrehozott, aminek típusa és a benne tárolt adatok a hiba jellegéről és részleteiről adnak felvilágosítást. Első próbálkozásunknál a hiba bekövetkezése egyben az alkalmazás leállítását is jelentette.

## X.2. Kivételek kezelése

Programunkat úgy szeretnénk továbbfejleszteni, hogy képes legyen kezelni a hibás felhasználói adatbevitelt a szám(ok) újbóli bekérésével. Ehhez olyan kódra van szükség, ami az alkalmazás leállása előtt érzékeli a kivételt, és gondoskodik a megfelelő vezérlésátadásról.

A C# nyelvben a try-catch-finally szerkezet segítségével oldhatjuk meg a feladatot. Ez egy try blokkot, egy vagy több catch blokkot és nulla vagy egy finally blokkot tartalmaz.

### X.2.1. A try-catch szerkezet

A try blokkban helyezük el azokat az utasításokat, amelyek végrehajtása során számíthatunk kivétel keletkezésére. Példánkban (X.5. ábra) a konverziós utasítás kerül ide. A catch blokk(ok)ba helyezük el azokat az utasításokat, amelyekkel a hibára kívánunk reagálni. Példánkban átveszünk egy FormatException típusú kivétel objektumot, aminek feladata a hibához kapcsolódó információk hordozása. Most csak a Message tulajdonságot használjuk fel, ami egy rövid leírást tartalmaz a hibáról. Az újbóli adatbekérést a Beolvas metódus rekurzív meghívásával oldjuk meg.

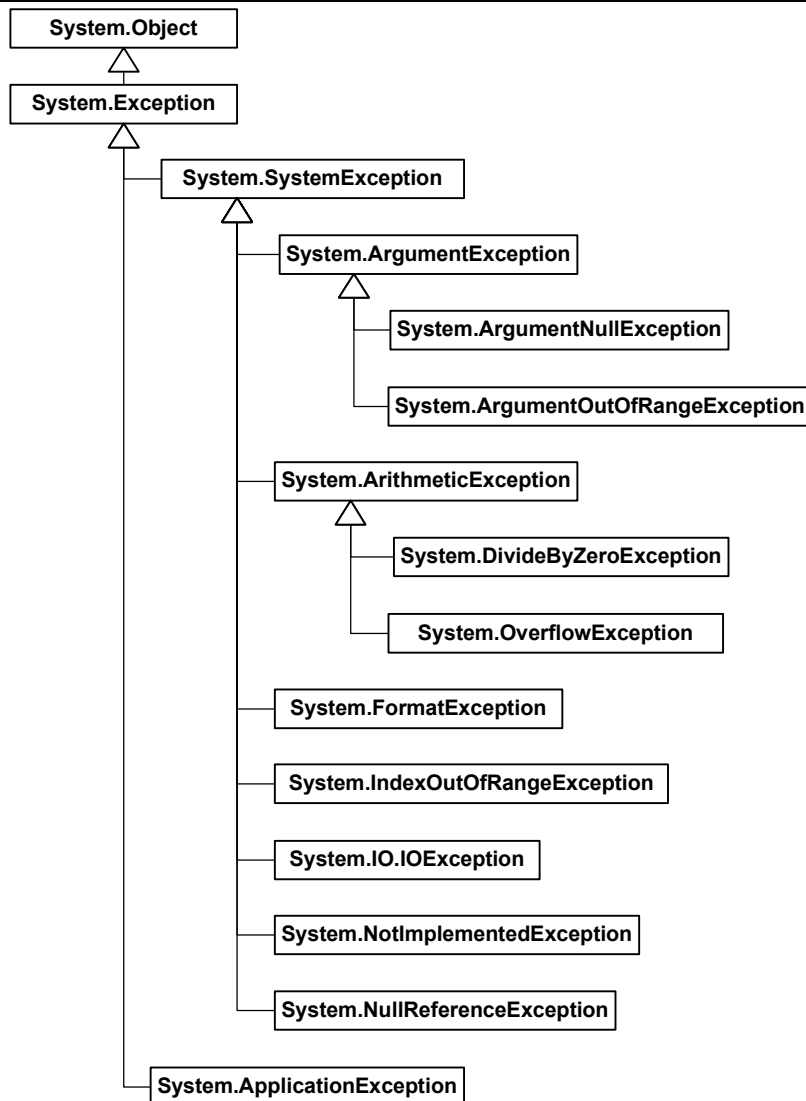
```
static int Beolvas(string Aktuális)
{
    Console.WriteLine("A"+Aktuális+" szám: ");
    int a=0;
    string s = System.Console.ReadLine();
    try
    {
        a = int.Parse(s);
    }
    catch (FormatException e)
    {
        Console.WriteLine("Hibásan adta meg a számot!");
        Console.WriteLine(e.Message);
        a = Beolvas(Aktuális);
    }
    return a;
}
```

X.5. ábra. try-catch blokk

Lépésenként végrehajtva az alkalmazást, és a már megszokott „asd” betűsört megadva végigkövethetjük, hogy a Parse meghívása után a kivétel hatására a vezérlés a catch blokk fejlécére ugrik, majd sorban végrehajtnak az ott szereplő utasítások. Érvényes értéket megadva a Beolvas metódus újbóli meghívásakor a Parse sikeres konverziót hajt végre, a vezérlés átugorja a catch blokkot, és a metódus visszaadja return-el az a változó értékét.

### X.2.2. Egy vagy több catch blokk

Sok metódus esetében a futás során többféle hiba is előfordulhat, amelyekre néha teljesen eltérően kell reagálni. A hibatípusok elkülönítését jól szolgálják a kivétel osztályok. Ezeket témakörök szerint rendszereztek egy külön ágat kialakítva az osztályhierarchia System névterében. Az ág csúcsán az Exception osztály áll, ami közvetlen leszármazottja az Object osztálynak. Az X.6. ábrán a teljesség igénye nélkül néhány gyakran alkalmazott kivételosztályt láthatunk jelezve hierarchiabeli elhelyezkedésüket is. Az X.7. ábra röviden ismerteti jellegzetes alkalmazási területüket is.



X.6. ábra. Fontosabb kivételosztályok és helyzetük a hierarchiában

System.Exception	az alkalmazás végrehajtása során előforduló hibákhoz társított kivételek őszosztálya
System.SystemException	a System névtér előre definiált kivételtípusainak őszosztálya
System.ArgumentException	egy metódus valamely aktuális paramétere érvénytelen
System.ArgumentNullException	null referencia nem megengedett átadása
System.ArgumentOutOfRangeException	az átadott paraméter az érvényes tartományon kívülre esik
System.ArithmeticException	aritmetikai műveletek és típuskonverzió során előálló kivételek őszosztálya
System.DivideByZeroException	nullával történő osztás
System.OverflowException	túlcsordulási hiba
System.FormatException	a paraméter formátuma nem megfelelő

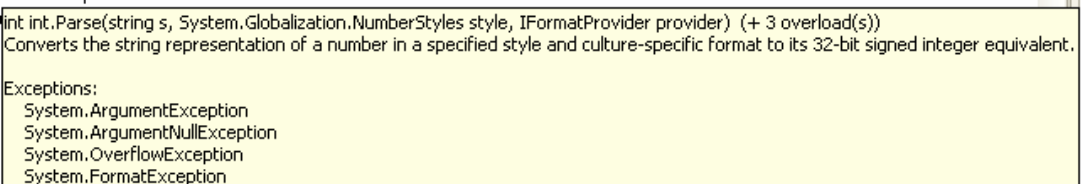
System.IndexOutOfRangeException	tömb túlindexelése
System.IO.IOException	fájlkezeléssel kapcsolatos kivételek őszosztálya
System.NotImplementedException	a meghívott metódus nem rendelkezik implementációval; például a fejlesztőrendszer a Class Details ablakban vizuálisan létrehozott metódusok vázába egy ilyen kivétel előidézését helyezi el
System.NullReferenceException	egy változón keresztül hivatkozunk egy objektum egy tagjára, és közben a változó null értékű
System.ApplicationException	a felhasználó által definiált kivételtípusainak őszosztálya

X.7. ábra. Fontosabb kivételosztályok és alkalmazási területük

Egy alkalmazás fejlesztése során tisztában kell lennünk azzal, hogy milyen kivételeket idézhetnek elő a futatókörnyezet (Common Language Runtime - CLR) vagy más forrásból származó osztályok/komponensek metódusai. A CLR esetében a fejlesztőrendszer súgójában részletes információt találunk minden metódusról, továbbá a kódszerkesztőben a metódus neve felé helyezve az egérmutatót a felbukkanó gyorslistában is információt kapunk a lehetséges kivételekről. Bár az utóbbi megoldás sokkal kényelmesebb, de alkalmazhatósága korlátozott olyankor, amikor több azonos nevű metódus is található az osztályban/struktúrában. Így például az `int` (`Int32`) struktúra `Parse` tagja esetén a gyorslista a három paraméteres változatot jeleníti meg, amihez négy kivételtípus tartozik. Az általunk alkalmazott egyparaméteres típus azonban három fajta kivételt idézhet elő. Ezek az `ArgumentNullException`, `FormatException` és az `OverflowException`.

```
int a = int.Parse(s);
```

```
return
```



X.8. ábra. Kivételtípusok a gyorslistában

Példaprogramunk futása közben az első típus gyakorlatilag nem fordulhat elő, így a megkívánt biztonság elérése érdekében a másodikra kell felkészítenünk alkalmazásunkat. Egész számként nem értelmezhető karaktorsorozat esetét eddig is kezelni tudta metódusunk, a továbbiakban a túlcsordulás, azaz az abszolút értékben túl nagy szám esetével kell foglalkoznunk. Ez kétféleképpen oldható meg. Vagy mindkét hibátípushoz külön `catch` blokkot rendelünk vagy egyetlen közös `catch` blokkot alkalmazunk.

Amennyiben az első utat választjuk, az X.9. ábrán ismertetett kódrészletet kell elhelyeznünk az eredeti `catch` blokkot követően a `return` elé. Lépésenként végrehajtva a programot nyomon követhetjük, hogy a szokásos „asd”-t megadva az első, míg a 2147483648 értéket megadva a második `catch` blokk hajtódik végre. A program működik, azonban ez a megoldás csak olyankor előnyös, ha a különböző hibák eltérő reakciót igényelnek.

```
catch (OverflowException e)
{
    Console.WriteLine("Hibásan adta meg a számot!");
}
```

```
Console.WriteLine(e.Message);  
a = Beolvas(Aktuális);  
}
```

#### X.9. ábra Túlcsordulási hiba kezelése

Példánkban azonban a második blokk az elsővel azonos utasításokat tartalmaz, így inkább a második megoldást, azaz a közös `catch` blokkot alkalmazzuk. Tudva, hogy egy ős osztályhoz létrehozott referencia változó képes tárolni bármely leszármazott osztályból példányosított objektum referenciáját, megkeressük az osztályhierarchiában a legközelebbi olyan osztályt, amely mindkét kivétel típusnak őse. Esetünkben a `SystemException` felel meg e követelménynek. Ezért ezt az osztályt adjuk meg a közös `catch` blokk fejlécében kivétel típusként (X.10. ábra).

```
catch (SystemException e)  
{  
    Console.WriteLine("Hibásan adta meg a számot!");  
    Console.WriteLine(e.Message);  
    a = Beolvas(Aktuális);  
}
```

#### X.10. ábra. Közös kivételkezelő

Általános érvénnyel elmondható, hogy egy `try` blokkhoz több `catch` blokk is kapcsolható. Ezek közül mindig csak egy hajtódik végre, és pedíg az, amelyikre elsőként teljesül felülről lefele haladva az, hogy formális paraméterének típusa vagy azonos a kivétel objektum típusával vagy őse annak. Amennyiben olyan kivételes esemény következik be, amelyikre a fenti két feltétel egyike sem teljesül, akkor az első példához hasonlóan nem kezelnek minősül a kivétel. Amennyiben a `catch` blokkban nincs ugró utasítás (pl. kivétel továbbadása, új kivétel előidézése, kilépés a programból, stb.), akkor az alkalmazás végrehajtása az utolsó `catch` blokkot követő utasítással folytatódik.

### X.2.3. Általános `catch` blokk

Az X.10. ábrán bemutatott közös kivételkezelőnket elkészíthetjük paraméter nélküli változatban is az X.11. ábrának megfelelően. Ez a megoldás a vezérlésátadás szempontjából egyenértékű azzal, mintha `Exception` típusú paramétert használnánk, tehát bármilyen kivétel esetén végrehajtódik. Az eltérés csak abban mutatkozik, hogy nem kapunk hozzáférést a kivétel objektumhoz, és nem rendelkezünk pontos információval az okra vonatkozólag.

```
catch  
{  
    Console.WriteLine("Hibásan adta meg a számot!");  
    a = Beolvas(Aktuális);  
}
```

#### X.11. ábra. Általános `catch` blokk

### X.2.4. Kivétel továbbadása

A C# nyelv lehetőséget biztosít arra, hogy a kivételt ne csak annak keletkezési szintjén érzékeljük, hanem a hívási lánc magasabb szintjein elhelyezkedő metódusokban is. Ezt a kivétel továbbadásával érhetjük el a `throw` kulcsszó segítségével. Demonstrálásként szolgáljon az X.12. ábrán bemutatott metódus, ami átvesz egy `Graphics` típusú objektumot

valamint két struktúrát, amelyek a szín és a befoglaló téglalapra vonatkozó információt hordozzák, majd rajzol egy kifestett téglalapot.

```
void Ellipszis(Graphics gr, Color cSzín, Rectangle rTéglalap)
{
    SolidBrush sbEcset = new SolidBrush(cSzín);
    try
    {
        gr.FillEllipse(sbEcset, rTéglalap);
    }
    catch (NullReferenceException e)
    {
        Console.WriteLine("Nincs hova rajzolni!" + e.Message);
        throw;
    }
    finally
    {
        sbEcset.Dispose();
    }
}
```

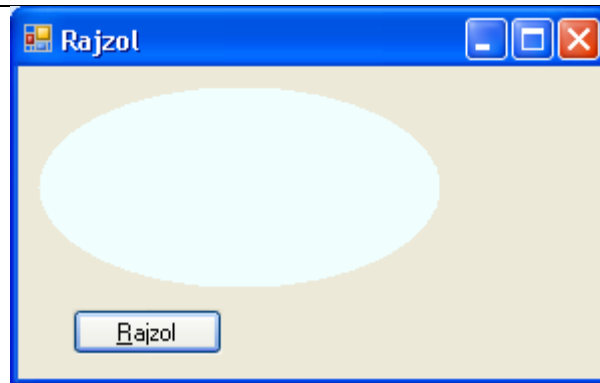
#### X.12. ábra. Kivétel továbbadása ellipszisrajzolás közben

A metódusban létrehozunk egy egyenletes színnel kifestő ecsetet, majd kísérletet teszünk a rajzolásra. Amennyiben az első paraméter null értékű, azaz nem rendelkezünk a festővászon megtestesítő Graphics típusú objektummal, akkor NullReferenceException típusú kivétel keletkezik, amit a szabványos kimenetre küldött hibüzenettel jelzünk, majd a kivételt továbbadjuk az Ellipszis metódus hívójának. A metódus egy finally blokkot is tartalmaz, amelynek magyarázata az X.2.5. szakaszban olvasható.

Az Ellipszis metódus kipróbálása érdekében készítsünk egy grafikus felületű alkalmazást a Windows Application sablon segítségével. Az ablakon helyezzünk el egy nyomógombot (btRajzol), és készítsünk hozzá egy a kattintásra reagáló eseménykezelőt btRajzol\_Click néven. Ebben határozzuk meg a befoglaló téglalapot és hívjuk meg az Ellipszis metódust (X.13. ábra).

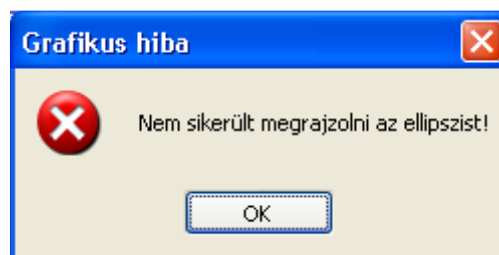
```
private void btRajzol_Click(object sender, EventArgs e)
{
    Rectangle rTéglalap = new Rectangle(0, 0, 200, 100);
    try
    {
        Ellipszis(this.CreateGraphics(), Color.Azure, rTéglalap);
    }
    catch
    {
        MessageBox.Show("Nem sikerült megrajzolni az ellipszist!",
            "Grafikus hiba", MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
}
```

#### X.13. ábra. Ellipszis rajzolás nyomógombon történő kattintásra

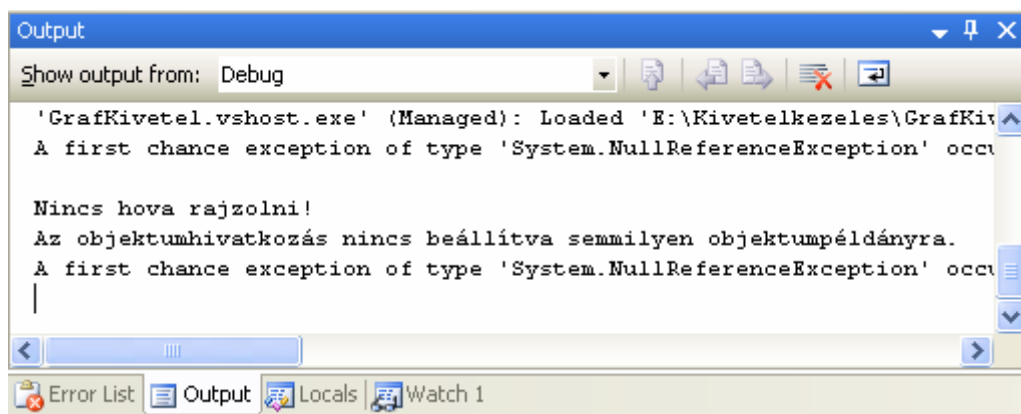


X.14. Eredmény

A program futásának eredményét az X.14. ábrán láthatjuk. A kivételkezelés kipróbálása érdekében cseréljük le az Ellipszis metódus meghívásának első aktuális paraméterét, azaz a `this.CreateGraphics()` helyett írjunk `null`-t. Újból lefuttatva a programot a nyomógomb eseménykezelőjének `catch` blokkjában előírt üzenetablak jelenik meg (X.15. ábra). A fejlesztőrendszer Output ablakában a „Show output from:” listában a Debug-ot kiválasztva láthatjuk, hogy a rendszer által generált hibüzenetek között megjelent az általunk definiált „Nincs hova rajzolni!” sor is (X.16. ábra).



X.15. ábra. Hibüzenet



X.16. ábra. Hibüzenet a standard kimeneten

## X.2.5. A finally blokk használata

Előfordulhat, hogy egy vagy több utasítást, például lefoglalt erőforrások felszabadítását a kivétel bekövetkezésétől függetlenül mindenképpen végre kell hajtanunk a `try`-jal vizsgált blokk után. Ez nem okoz különösebb gondot, ha egyik `catch` blokk sem tartalmaz olyan ugró utasítást, amelynek hatására az utolsó `catch` blokkot követő kódsort kihagyva valahol máshol folytatódna az alkalmazás futása. Amennyiben azonban fennáll az átugrás veszélye,

akkor a végrehajtást csak úgy biztosíthatjuk, ha az említett utasításokat egy `finally` blokkban helyezük el.

Az X.12. ábrán bemutatott `Ellipszis` metódusban létrehozunk egy `SolidBrush` típusú ecset objektumot. Használata után a `Dispose` metódus meghívásával fel kell szabadítanunk az általa lefoglalt erőforrásokat attól függetlenül, hogy a rajzolás sikeres volt-e vagy sem. A feladatot megoldó utasítást egy `finally` blokkba helyezük el. Lépésenként végrehajtva az alkalmazást az `Ellipszis` metódus fentiekben bemutatott hibás paraméterezése mellett, láthatjuk, hogy a `throw` végrehajtásakor a metódusból történő kilépés előtt még végrehajtódik a `Dispose` is.

## X.2.6. Túlcsordulási kivétel

Túlcsordulásról beszélünk akkor, ha egy adatot a szükségesnél kisebb tárolóképességű változóba kívánunk elhelyezni. Konstans értékek esetén a fejlesztőrendszer már fordítási időben kijelzi, míg változóknál csak futási időben derül ki a probléma, amennyiben be van kapcsolva a túlcsordulás figyelése.

Próbaképp futtassuk le újból a két szám bekérését és összeadását végző programunkat első számként az `int`-ben tárolható legnagyobb értéket, azaz 2147483647-et és második értéként 1-et megadva. Az eredmény -2147483648 lesz, ami egyértelműen utal a túlcsordulás bekövetkeztére, kivétel azonban nem keletkezett.

A túlcsordulás figyelését és bekövetkezésekor kivétel előidézését kétféleképpen érhetjük el. A teljes megoldásra vonatkozóan bekapcsolja az ellenőrzést a fordító `/checked+` parancssori kapcsolója, amit a fejlesztőrendszerben úgy állíthatunk be, hogy kiválasztjuk a `Project` menü `Projektnév Properties...` menüpontját, majd az előbukkanó párbeszédpanelen a `Build` fület választjuk, ezt követően az `Advanced` gombon kattintunk, és végül bekapcsoljuk a `Check for arithmetic overflow/underflow` jelölőnégyzetet. A beállítás elvégzése után újból próbálkozva az alkalmazásunk `OverflowException` kivétellel leáll az összeadásnál.

A fordítóprogram kapcsolóitól függetlenül egy kifejezésre (X.17. ábra) vagy egy utasításcsoportra (X.18. ábra) is szabályozhatjuk a túlcsordulás figyelését a `checked` és `unchecked` kulcsszavak alkalmazásával.

```
int c = checked(a + b);
```

X.17. ábra Túlcsordulás figyelése egy kifejezés kiértéklése során

```
int a, b, c, d;
checked
{
    a = Beolvas("z első");
    b = Beolvas(" második");
    c = a + b;
    d = a*b;
}
```

X.18. ábra Utasításcsoportra kiterjedő túlcsordulásfigyelés

## X.3. Kivételek előidézése

Az eddigiekben az általunk meghívott metódusokban előidézett kivételek kezelésével és továbbadásával foglalkoztunk. Nézzük most meg, hogyan készíthetünk mi magunk is olyan kódrészletet, ami egy kivétel segítségével jelzi a hívó számára egy hiba bekövetkezését.

A feladat legyen egy átlagszámító módszer elkészítése, amely egy `double` típusú elemekből álló tömbre vonatkozó referenciát vesz át, és visszaadja az abban tárolt adatok átlagát. A lehetséges hibák jelzésére a következő kivétel osztályokat használjuk fel:

`ArgumentNullException` a módszert null referenciával hívták meg, azaz nem létezik a tömb;

`ArgumentException` a tömb elemeinek száma nulla vagy a tömb valamely eleme érvénytelen (nem szám vagy végtelen érték);

`ArithmeticException` a számok összege meghaladja a `double` típusban maximálisan tárolható értéket (`double.MaxValue`).

A módszer kódját az X.19. ábra tartalmazza. A kivételek előidézése a `throw` kulcsszó segítségével történik, amit egy kivétel objektum létrehozása követ. a kivételosztály konstruktorának paraméterként átadtunk egy rövid hibaüzenetet.

```
static double ÁtlagSzámít(double[] Tömb)
{
    if (Tömb == null)
        throw new ArgumentNullException("A tömb nem létezik!");
    int Méret=Tömb.Length;
    if (Méret == 0)
        throw new ArgumentException("A tömb nem tartalmaz elemeket!");
    double Átlag = 0;
    for (int i = 0; i < Méret; i++)
    {
        if (double.IsNaN(Tömb[i]) || double.IsInfinity(Tömb[i]))
            throw new ArgumentException("A tömb " + i.ToString() +
                ". eleme érvénytelen értéket tartalmaz!");
        Átlag = Átlag + Tömb[i];
        if (double.IsInfinity(Átlag))
            throw new ArithmeticException("A tömb elemeinek összege" +
                " túl nagy érték!");
    }
    Átlag /= Méret;
    return Átlag;
}
```

X.19. ábra Kivétel előidézése

## X.4. Jótanácsok

Az alábbiakban összefoglaltunk néhány ajánlást a kivételek alkalmazásával és kezelésével kapcsolatosan.

- A kivételek előidézése és kezelése jelentős erőforrás igényel jár és lassítja az alkalmazás végrehajtását az újabb osztályok betöltése és a veremkezelési feladatok következtében. Lehetőleg csökkentsük minimális mértékűre az általunk előidézett kivételek számát.
- Csak olyan kivételeket dolgozzunk fel, amelyek esetén ismert az őket előidéző hiba kezelési módja. a többi kivétel feldolgozását engedjük át magasabb hívási szinten elhelyezett kivételkezelőknek.
- Gondoskodjunk mindig azon nem használt erőforrások felszabadításáról, amelyek nem tartoznak az automatikus szemégyűjtő mechanizmus hatálya alá.
- Kerüljük újabb kivétel előidézését a `finally` blokkban, ugyanis ha egy kivételt nem fogunk el egyetlen `catch` blokkal sem, akkor végrehajtnak a `finally` blokkban elhelyezett utasítások, és az újabb kivétel előidézése következtében az eredeti elvész.
- Amennyiben egy programszakaszon belül több egymás utáni utasításnál is elképzelhető kivétel keletkezése, úgy ne készítsünk ezekhez külön `try-catch-finally`

szerkezeteket, mert nehezen olvashatóvá tennék a kódot. Helyezzük el inkább az érintett utasításokat egyetlen `try` blokkban, és készítsünk minden kivétel típushoz `catch` blokkot.

- Több `catch` blokkot tartalmazó kivételkezelésnél az egyes blokkokat az osztályhierarchia figyelembe vételével úgy kell sorba rendezni, hogy fentről lefele haladva a specifikusabb osztályok megelőzzék az általánosabbakat.

## **X.5. Ellenőrző kérdések**

1. Mikor minősül nem kezeltnek egy kivétel?
2. Ellenőrizni tudjuk-e a fejlesztőrendszer segítségével, hogy egy kivétel bekövetkezésekor az érvényességi körükön belül levő változók milyen értékkel rendelkeznek?
3. Hogyan tudjuk kideríteni, hogy a .NET osztályhierarchia egy metódusa milyen kivételeket idézhet elő?
4. A kivétel objektum mely tagjából olvashatjuk ki a hiba rövid szöveges leírását?
5. Mikor célszerű közös `catch` blokkot készíteni több kivétel típushoz?
6. Ha több `catch` blokkot készítünk egy kivételkezeléshez, akkor bármilyen sorrendben elhelyezhetjük ezeket?
7. Mire szolgál a `new` kulcsszó a kivételt előidéző utasításban?
8. Keletkezik mindig kivétel túlcsoportulási hiba esetén?
9. Megoldható-e, hogy egy kivétel feldolgozása ne abban a metódusban történjék, ahol azt elfogtuk?
10. Kötelező-e a `finally` blokk használata?

Exception Assistant .....	2	throw .....	6, 10
kivétel		try-catch-finally .....	3
előidézés .....	9	általános catch blokk.....	6
kezelés .....	2	catch blokk.....	3
nem kezelt .....	1	finally blokk.....	8
továbbadás.....	6	try blokk.....	3
X. Kivételkezelés .....	1		
X.1. Nem kezelt kivétel.....	1		
X.2. Kivételek kezelése.....	2		
X.2.1. A try-catch szerkezet .....	3		
X.2.2. Egy vagy több catch blokk .....	3		
X.2.3. Általános catch blokk .....	6		
X.2.4. Kivétel továbbadása .....	6		
X.2.5. A finally blokk használata.....	8		
X.2.6. Túlszordulási kivétel .....	9		
X.3. Kivételek előidézése.....	9		
X.4. Jótanácsok .....	10		
X.5. Ellenőrző kérdések .....	11		