Eötvös Loránd Tudományegyetem
Természettudományi Kar

Alkalmazott Modul III

3. előadás

Procedurális programozás: alprogramok, kivételkezelés

© 2011.10.03. Giachetta Roberto
groberto@inf.elte.hu
http://people.inf.elte.hu/groberto

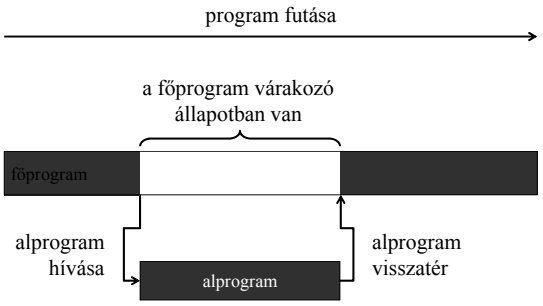
Alprogramok
Szükségessége

- A főprogram terjedelme a feladat bonyolultságával arányos
  - egy adott bonyolultságon túl a főprogram olyan méretűvé válik, hogy áttekinthetetlen lesz a programozó számára
  - előfordulhatnak benne ismétlődő szakaszok, amelyek feleslegesen növelik a kód hosszát
  - amennyiben a program egy részét egy másik programban is használni akarjuk, manuálisan kell átmásolnunk a megfelelő kódrészletet
- A megfelelő megoldás erre *kódrészletek kiemelése*, elhelyezése a program más részeiben, és egyszeri hivatkozással futtatni őket, ezáltal csökken a főprogram hossza, és megszűnnek az ismétlődő szakaszok

Alprogramok
Működése

- A kiemelt programrészeket nevezzük *alprogram*oknak (*szubrutin*nak), amelyek tetszőleges részprogramot tartalmazhatnak, és egy hívással futtathatóak
  - az utasítás végrehajtásakor meghívódik az alprogram (a főprogram átadja a vezérlést az alprogramnak), és az lefut
  - a szubrutin lefutását követően a vezérlés visszaadódik a főprogramnak, és az folytatja a működését
  - egy alprogramot bármennyiszer le lehet futtatni, és bármikor meg lehet hívni
  - természetesen alprogramok meghívhatnak más alprogramokat is, így elméletileg a végtelenségig bővíthető a hívások folyamata

Alprogramok
Működése



Alprogramok
Kommunikáció

- Az alprogramok egymásnak átadhatnak értékeket (*kommunikál*hatnak) a következő módszerekkel:
  - *globális változók/konstansok*: olyan változók, amelyek a teljes programkódban érvényben vannak, nem csak egy adott programrészen belül
  - *paraméterek*: olyan változók, amelyek az alprogram meghívásakor kapnak értéket, és a teljes alprogramban érvényesek, paraméterből bármennyit adhatunk egy alprogramnak, de a meghíváskor mindegyiknek értéket kell adnunk
  - *visszatérési érték*: az alprogram által visszaszolgáltatott érték, amely visszakerül a hívás helyére, egy alprogramnak csak egy visszatérési értéke lehet

Alprogramok
Felépítése

- Az alprogramok fajtái:
  - *eljárás*: egy utasítássorozatot hajt végre
  - *függvény*: egy számítást végez el, amelynek eredménye van, az eredményt pedig visszaadja a függvény meghívójának (ez a visszatérési érték)
- Az alprogram részei:
  - *deklarációs rész*: tartalmazza az alprogram nevét, függvény esetén a visszatérési érték típusát, illetve a paraméterek listáját
  - *alprogram törzse*: a hozzátartozó utasítássorozat
- A C# nyelvben a két rész erősen összefügg, a deklarációt az alprogram törzse követi

Alprogramok
C++-ban

- A C# alprogramok szerkezete:

```
<típus> <módosítók> <név>(<paraméterek>){
   <utasítások>  // az alprogram törzse
}
```

ahol:

- a *típus* a visszatérési érték típusa, amely **`void`**, ha eljárásról van szó
- a *módosítók* különböző kulcsszavakat tartalmaznak leginkább objektumorientált környezetben
- a *név* a függvény neve
- a *paraméterek* változók deklarációját jelenti (ugyanúgy, mint bárhol máshol a kódban)

Alprogramok
Eljárások

- Az eljárások nem adnak vissza értéket a hívónak
  - pl.: `void skip(){}`
  - amennyiben nem szeretnénk, hogy az alprogram lefusson a végéig, bármely ponton megszakíthatjuk a működését a **`return`** utasítással
- A függvények típusa a visszatérési értéktől függ, amelyet a return utasítással adhatunk meg
  - pl: `Int32 one(){ return 1; }`
  - hasonlóan, mint eljárások esetén, a **`return`** utasítás egyben megszakítja a függvény működését
  - a **`return`** után szerepelhet érték, változó, vagy összett kifejezés is

Alprogramok
Hívása

- Lehetőségünk van utasítás formájában lefuttatni az alprogramot bármely programblokkban (ez érvényes eljárásokra és függvényekre is):

```
// utasítások
<alprogramnév>();
// utasítások
```

- A függvények mindig visszaszolgáltatnak egy értéket, ezt felhasználhatjuk (pl. értékadásban, vagy kifejezés részeként):

```
<érték> = <függvénynév>();
// ahol az érték típusa megegyezik, vagy
// kompatibilis a függvény típusával
```


Alprogramok
Hívása

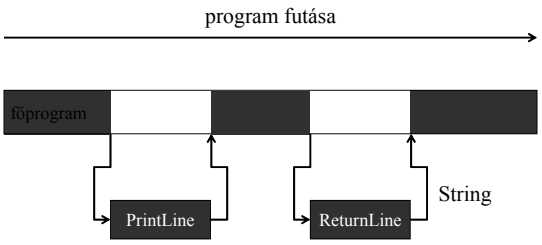
- Pl.:

```
void PrintLine(){ // eljárás
   Console.WriteLine("Ezt az eljárás írja ki.");
}

String ReturnLine(){ // függvény
   return "Ezt egy függvény adja vissza.";
}

void RunSubrutines(){ // futtató eljárás
   PrintLine(); // eljárás végrehajtása
   Console.WriteLine(ReturnLine());
      // függvény meghívása és az eredmény kiírása
}
```


Alprogramok
Hívása



Alprogramok
Módosítók használata

- A módosító kulcsszavak különböző jelzőkkel láthatják el az alprogramot, amelyek befolyásolják a hívási környezetet és az elérhetőséget
  - általában objektumorientált kontextusban jönnek elő
  - egyelőre a **`static`** kulcsszó használata szükséges, amellyel az alprogram statikus mivoltát tudjuk jelezni, azaz nem szükséges példányosítani a típust a használathoz (pl. a **`Program`** típust)
  - mivel a főprogram kötelezően használja, és statikus alprogram csak statikus alprogramot hívhat, procedurális kontextusban valamennyi szubrutin megkapja ezt a jelzőt

Alprogramok
Példa

Feladat: Írjuk ki a számokat száztól egyig egy alprogramban.

- az alprogram egy eljárás lesz, a neve **`PrintNumbers`**, a főprogramban lefuttatjuk az eljárást
- egy visszafelé haladó számlálót használunk a kiíráshoz

Megoldás:

```
class Program { // osztály

   static void PrintNumbers() {
      // számokat kiíró eljárás, ami statikus
      for (Int32 i = 100; i >= 1; i--)
         Console.WriteLine(i);
   } // alprogram vége
```


Alprogramok
Példa

Megoldás:

```
   static void Main(string[] args) {
      // főprogram
      Console.WriteLine("Visszaszámlálás 100-
                         tól:");

      PrintNumbers(); // alprogram meghívása

      Console.ReadKey();
   } // főprogram vége

} // osztály vége
```


Alprogramok
Globális változók

- A programunk egyes alprogramjai használhatnak *globális változók*at a kommunikációra
- A globális változók olyan változók, amelyek bárhol elérhetőek a programban
  - mivel számunka az osztály szintje a procedurális program határa, ezért az osztály szintjén kell létrehoznunk a globális változókat
  - a változók minden alprogram számára elérhetőek lesznek és minden alprogram ugyanazokat az értékeket fogja módosítani
  - a globális változóknak szintén adhatunk kezdőértéket
  - statikus alprogramból csak a statikus változók érhetőek el

Alprogramok
Globális változók

- A globális változók elhelyezkedése:

```
class Program {
   <globális változók deklarációja>

   <alprogramok deklarációja>
   // az alprogramokban elérhetőek a globális
   // változók

   static void Main() {
      <főprogram utasításai>
      // a főprogramban is elérhetőek a globális
      // változók
   }
}
```


Alprogramok
Példa

Feladat: Írjuk ki a számokat *n*-től egyig egy alprogramban úgy, hogy a főprogramban kérjük be *n* értékét.

- *n* értékét egy globális változón keresztül adjuk át a **`PrintNumbers`** alprogramnak

Megoldás:

```
class Program { // osztály
   static Int32 n;
   // n szintén statikus lesz, mivel az
   // alprogramok is statikusak

   static void PrintNumbers() {
      // számokat kiíró eljárás, ami statikus
```


Alprogramok
Példa

Megoldás:

```
      for (Int32 i = n; i >= 1; i--)
         // az alprogram látja az n változót
         Console.WriteLine(i);
   }

   static void Main() { // főprogram
      Console.Write("Kezdőérték:");
      n = Convert.ToInt32(Console.ReadLine());
      // globális változó bekérése
      PrintNumbers(); // alprogram meghívása
      Console.ReadKey();
   }
}
```

Alprogramok
Példa

Feladat: Adjuk meg egy valós számokat tartalmazó tömb átlagát.

- a tömböt (`valuesArray`) vegyük fel globális értékként
- legyen egy eljárás (`ReadValues`), amely beolvassa a tömb elemeit
- legyen egy függvény (`CalculateAverage`), amely kiszámítja az átlagot és visszatérési értékben visszaadja azt, ehhez az összegzés programozási tételt alkalmazzuk

Megoldás:

```
class Program { // osztály
   static Double[] valuesArray = new Double[10];
   // értékeket tároló tömb
```


Alprogramok
Példa

Megoldás:

```
   static void ReadValues() {
      // tömb elemeit beolvasó eljárás
      for (Int32 i = 0; i < valuesArray.Length;
           i++){
         Console.Write("A(z)" + (i + 1) +
                       ". érték: ");
         valuesArray[i] =
            Convert.ToDouble(Console.ReadLine());
      }
   }
```


Alprogramok
Példa

Megoldás:

```
   static Double CalculateAverage() {
      // átlagot kiszámító függvény
      Double average = 0;
      // összegzést alkalmazunk, az átlag is valós

      for (Int32 i = 0; i < valuesArray.Length;
           i++) {
         average += valuesArray[i];
      }

      return average / valuesArray.Length;
      // visszatérési érték
   }
```


Alprogramok
Példa

Megoldás:

```
   static void Main(string[] args) { // főprogram
      Console.WriteLine("Számok beolvasása:");

      ReadValues(); // eljárás meghívása

      Double result = CalculateAverage();
         // függvény meghívása
      Console.WriteLine("A számok átlaga: " +
                        result);

      Console.ReadKey();
   }
}
```


Alprogramok kommunikációja
Paraméterek

- Egy másik lehetőség a kommunikációra a *paraméterátadás*, ahol közvetlenül az alprogramnak adjuk meg, milyen értékeket vegyen át az őt meghívó programrésztől, és használjon a feldolgozás során
  - így nem csak globális, de más szubrutin lokális értékeivel is dolgozhatunk
- A paraméterátadásban két érték vesz részt:
  - *aktuális paraméter*: amit átadunk az alprogramnak a meghíváskor, lehet változó vagy konstans érték
  - *formális paraméter*: ami az alprogram paraméterlistájában deklarálva van, és amit használunk a függvény törzsében

Alprogramok kommunikációja
Paraméterek

Feladat: Írjuk ki a számokat *n*-től egyig egy alprogramban úgy, hogy a főprogramban kérjük be *n* értékét.

- *n* értékét paraméterátadással adjuk át, ezért felveszünk egy egész paramétert a `PrintNumbers` alprogramnál

Megoldás:

```
class Program { // osztály
   static void PrintNumbers(Int32 n) {
      // számokat kiíró eljárás egész paraméterrel
      // (n a formális paraméter)
      for (Int32 i = n; i >= 1; i--)
         Console.WriteLine(i);
   }
```

Alprogramok
Példa

```
static void Main() { // főprogram
   Console.Write("Kezdőérték:");
   Int32 number =
      Convert.ToInt32(Console.ReadLine());
      // változó bekérése

   PrintNumbers(number);
   // alprogram meghívása a paraméterrel
   // (number az aktuális paraméter)

   Console.ReadKey();
}
}
```


Alprogramok paraméterátadása
A paraméterátadás iránya

- A *paraméterátadás iránya* szerint a következő paramétertípusokat különböztetjük meg:
  - *bemenő paraméter*: a hívás pillanatában átadódik az az aktuális paraméter értéke a formálisnak, és onnantól az adatok módosítása nincs hatással az aktuális paraméterre
    - ez az alapértelmezett átadási mód a primitív típusokra, és bizonyos összetett típus esetén is (erről bővebben a következő előadásban)
    - lényegében az érték átmásolódik a memóriában egy új változóba, ezért a kettő egymástól független lesz, ezért szokás ezt *érték szerinti paraméterátadás*nak is nevezni

Alprogramok paraméterátadása
A paraméterátadás iránya

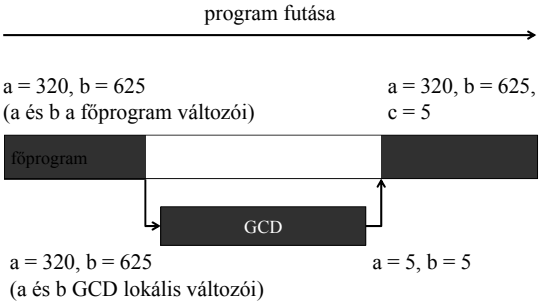
- pl.:

```
Int32 GCD(Int32 a, Int32 b){
   // euklideszi algoritmus bemenő
   // paraméterekkel
   while (a != b)
      if (a > b) a -= b; else b -= a;
   return a; // visszatérési érték
}

Int32 a = 320, b = 625, c;
// c nem kell, hogy értéket kapjanak
c = GCD(a, b);
// a bemenő paraméterek módosítása nem lesz
// hatással a és b értékére
```


Alprogramok paraméterátadása
A paraméterátadás iránya

- pl.:



Alprogramok paraméterátadása
A paraméterátadás iránya

- *kimenő paraméter*: az értékeket az alprogram állítja be, majd a hívás végeztével a formális értékek visszaíródnak az aktuális paraméterbe
  - az érték szerinti paraméterátadás egy fordított esete
  - használni úgy tudjuk, hogy az `out` kulcsszóval megjelöljük a formális és az aktuális paramétert is
  - hasonlóan, mint a visszatérési érték esetén, ezért leginkább akkor használatos, amikor több visszatérési értéket szeretnénk

Alprogramok paraméterátadása
A paraméterátadás iránya

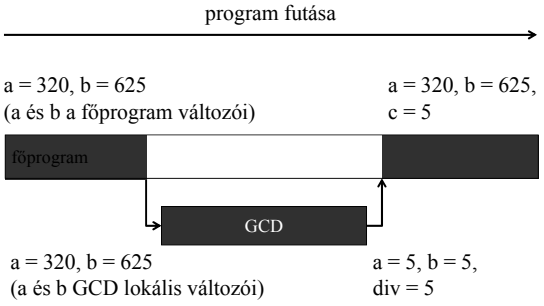
- pl.:

```
void GCD(Int32 a, Int32 b, out Int32 div){
   // euklideszi algoritmus kimenő
   // paraméterrel
   while (a != b)
      if (a > b) a -= b; else b -= a;
   div = a; // div-nek értéket kell kapnia
}

Int32 a = 320, b = 625, c;
// c nem kell, hogy értéket kapjon
GCD(a, b, out c);
// híváskor jeleznünk kell, hogy c kimenő
// paraméter, amely értéket kap
```

Alprogramok paraméterátadása
A paraméterátadás iránya

- pl.:



Alprogramok paraméterátadása
A paraméterátadás iránya

- *be- és kimenő paraméter*: az értékek a híváskor bekerülnek az alprogramba, amely azokat átírhatja, majd a hívás végeztével a módosított értékek visszakerülnek, tehát az alprogram az aktuális paraméter értékét módosíthatja
  - ez úgy oldható meg, hogy az értékek ténylegesen nem kerülnek lemásolásra a memóriában, hanem a formális paraméter az aktuális paraméterre fog hivatkozni végig, így ugyanazon változóról lesz szó
  - az ehhez használatos technika az érték helyett a memóriacímet másolja le, ezért nevezzük cím szerinti paraméterátadásnak
  - a legtöbb összetett típusra ez az alapértelmezett, primitív típusra a **`ref`** kulcsszóval használhatjuk

Alprogramok paraméterátadása
A paraméterátadás iránya

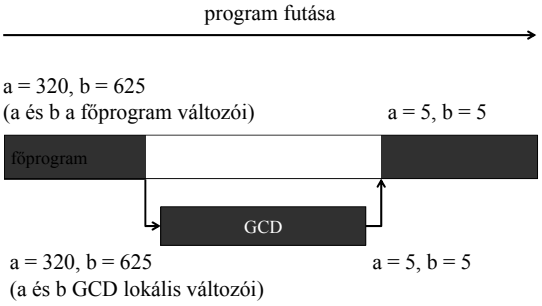
- pl.:

```
void GCD(ref Int32 a, ref Int32 b){
   // euklideszi algoritmus be- és kimenő
   // paraméterrel
   while (a != b)
      if (a > b) a -= b; else b -= a;
} // nem kell visszatérési érték

Int32 a = 320, b = 625;
GCD(ref a, ref b);
// a és b is módosítják az értéküket, így
// mindkettőben meglesz az eredmény, de a
// régi értékek elvesznek
```


Alprogramok paraméterátadása
A paraméterátadás iránya

- pl.:



Alprogramok
Példa

Feladat: Adjuk meg egy valós számokat tartalmazó tömb átlagát.

- a tömböt (**`valuesArray`**) vegyük fel a főprogramban, és kérjük be a
- legyen egy eljárás (**`ReadValues`**), amely beolvassa a tömb elemeit
- legyen egy függvény (**`CalculateAverage`**), amely kiszámítja az átlagot és visszatérési értékben visszaadja azt, ehhez az összegzés programozási tételt alkalmazzuk

Megoldás:

```
class Program { // osztály
   static Double[] valuesArray = new Double[10];
   // értékeket tároló tömb
```


Alprogramok
Példa

Megoldás:

```
static void ReadValues() {
   // tömb elemeit beolvasó eljárás
   for (Int32 i = 0; i < valuesArray.Length;
        i++){
      Console.Write("A(z)" + (i + 1) +
                    ". érték: ");
      valuesArray[i] =
         Convert.ToDouble(Console.ReadLine());
   }
}
```

Alprogramok
Példa

Megoldás:

```
static Double CalculateAverage() {
   // átlagot kiszámító függvény
   Double average = 0;
   // összegzést alkalmazunk, az átlag is valós

   for (Int32 i = 0; i < valuesArray.Length;
        i++) {
      average += valuesArray[i];
   }

   return average / valuesArray.Length;
   // visszatérési érték
}
```


Alprogramok
Példa

Megoldás:

```
static void Main(string[] args) { // főprogram
   Console.WriteLine("Számok beolvasása:");

   ReadValues(); // eljárás meghívása

   Double result = CalculateAverage();
      // függvény meghívása
   Console.WriteLine("A számok átlaga: " +
                     result);

   Console.ReadKey();
}
}
```


Alprogramok paraméterátadása
A főprogram paraméterei

- Nem csak az alprogramok, hanem a főprogram is kaphat paraméterezést
  - a főprogramnak átadott aktuális paraméterek a parancssorban megadott, fájlnév utáni szavak lesznek
  - bármennyi, bármilyen paramétert megadhatunk neki, amelyet a főprogrammal ki tudunk értékeltetni
  - pl.: `program.exe input.txt 75`
- A főprogram formális paramétere a `string[] args` tömb, amely tartalmazza ezen értékeket:

```
void Main(string[] args){
   // itt használhatjuk args elemeit
}
```


Kivételkezelés
Működése

- A kivételek futás közben történő felismerését, feldolgozását, majd a programfutás megfelelő állapotba való visszaállítását nevezzük *kivételkezelés*nek (*exception handling*)
  - a kivételek *kiváltód*nak (vagy úgymond dobódnak (*throw*)), és azokat lehetőségünk van a program valamely szintjén *lekezelni* (úgymond elkapni (*catch*))
  - a kivételkezelés rendszerint külön *kivételkezelő szakasz*t (blokkot) igényel a kódban, amely dedikáltan figyelni a kivételek előfordulását, és arra megfelelő reakció lefuttatását
  - a kivételkezelést a korábbi nyelvek (pl. C, Pascal) nem támogatják, itt a hasonló szituációk hibához (hibakódhoz), vagy extremális érték visszaadáshoz vezetnek

Kivételkezelés
Működése

- A kivételek úgy keletkeznek a programban, hogy egy hibához vezető lépést megakadályozunk egy kivétel kiváltásával, pl.:
  - mielőtt túlindexelnénk a tömböt, ellenőrizzük az indexet, és ha az hibás, kivételt váltunk ki
  - mielőtt osztanánk, ellenőrizzük az osztót, és ha az 0, kivételt váltunk ki
- A beépített típusok műveletei sok esetben keltenek kivételeket, hogy felhívják a figyelmet a helytelen működésre, vagy inkonzisztens állapotra, ezek a kivételek különböző típusúak lehetnek
- A kivétel általános típusa az `Exception`, de a műveltek rendszerint ennek valamely speciálisabb változatát keltik

Kivételkezelés
A kivételkezelő blokk

- A le nem kezelt kivétel ugyanúgy megszakítja a program futását, mint a hiba, de általában nem hagyja inkonzisztens állapotban a mentett adatokat (pl. fájlban, vagy adatbázisban)
- Kivételt kezelni egy kivételkezelő (`try-catch-finally`) szakasszal tudunk, amelyben meg kell adnunk az elfogandó kivétel típusát, és kivétel esetén lefuttathatunk egy megadott utasítássorozatot:

```
try {
   <kivételkezelt utasítások>
} catch (<elfogott kivétel típusa>){
   <kivételkezelő utasítások>
}
finally { <mindenképp lefuttatandó utasítások> }
```

Kivételkezelés
Példa

- Pl.:

```
int i = 0; // nem kivételkezelt utasítások
String s = Console.ReadLine();
try {// kivételkezelt utasítások
   i = Convert.ToInt32(s);
   // a ToInt32 három féle kivételt is dobhat
   Console.WriteLine("Sikeres konverzió!");
}
catch (Exception) { // kivétel kezelése
   Console.WriteLine("Gáz van!");
}
finally { // minden esetben végrehajtandó
   Console.ReadKey();
}
```


Kivételkezelés
A kivételkezelő szakasz

- Kivételkezelő szakaszt bármely programblokkon belül elhelyezhetünk a programban
  - ha a **`try`** blokkban kivétel keletkezik, akkor a vezérlés a **`catch`** ágra ugrik, az utána következő utasítások így nem futnak le
  - a program ellenőrzi, hogy a kivétel típusa egyezik-e, vagy speciális esete a **`catch`**-ben megadottnak, különben tovább dobja a kivételt
  - ha elfogta a kivételt, akkor futtatja a **`catch`** ág utasításait
  - a **`finally`** blokk használata nem kötelező, de amennyiben van, úgy az abban lévő utasításokat akkor is futtatja, ha keletkezett (akár le nem kezelt) kivétel, és akkor is, ha nem

Kivételkezelés
Kivételkezelő ágak

- Lehetőségünk van különböző típusú kivételek elfogására is, amennyiben több **`catch`** ágat készítünk a szakaszhoz
  - a kivétel típusát sorban egyezteti az ágakon, és az első találat ágát futtatja
  - amennyiben biztosan el akarunk kapni bármilyen kivételt, kapjuk el az általános **`Exception`** típust is
  - pl.:

```
try { // kivételkezelt utasítások
   …
} // kivételkezelő ágak:
catch (ArgumentException) { … }
catch (NullReferenceException) { … }
catch (Exception) { … }
```


Kivételkezelés
Kivételek üzenetei

- A kivételek üzenettel rendelkeznek, amelyet a kivétel **`Message`** tulajdonságán keresztül kérhetünk le
- Ha egy kivételkezelő szakaszban ki akarjuk írni az üzenetet, akkor ehhez egy változónevet adunk a kivételnek, amely felhasználhatunk a **`catch`** ágban
  - pl.:

```
try { // kivételkezelt utasítások
   …
}
catch (Exception ex) { // kivétel kezelése
   // a kivétel az ex változónevet kapja
   Console.WriteLine(ex.Message);
   // kivétel üzenetének kiírása
}
```


Kivételkezelés
Példa

Feladat: Írjuk ki a számokat *n*-től egyig egy alprogramban úgy, hogy a főprogramban kérjük be *n* értékét.

- amikor a felhasználó megadja *n* értékét, lehetséges, hogy nem számot ad, készítsük fel a programot ennek a lekezelésére
- nem szám esetén a **`Convert.ToInt32`** művelet egy **`FormatException`** kivételt fog dobni, ezért ezt a részt helyezzük kivételkezelés alá
- a kivételkezelt részt ezen felül behelyezzük egy ciklusba, amelyet egy logikai változó (**`success`**) vezérel, és csak akkor lépünk ki a ciklusból, ha a felhasználónak sikerült számot megadnia

Kivételkezelés
Példa

Megoldás:

```
void Main(string[] args) {
   Int32 number = 0;
   Boolean success = false;
   // változó, amivel megmondjuk, sikeres volt-e
   // a bekérés

   do {
      try { // kivételkezelt utasítások
         Console.Write("Kezdőérték:");
         number =
            Convert.ToInt32(Console.ReadLine());
            // a konverziónál történhet hiba
```

Kivételkezelés
Példa

Megoldás:

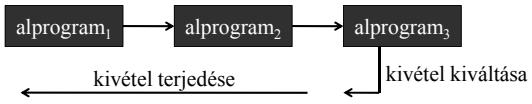
```
            success = true;
            // ha sikertelen volt a konverzió, ez az
            // utasítás már nem fut le
         }
         catch (FormatException) {
            // formátumhiba lekezelése
            success = false;
            Console.WriteLine("Nem számot írtál be,
                               próbáld újra!");
         } // kivételkezelés vége
      } while (!success); // addig nem lép ki a
         // ciklus, amíg sikertelen volt a művelet
      …
```


Kivételkezelés
Kivételek kiváltása

- Kivételt mi is kiválthatunk tetszőleges pontján a programnak, amelyet egy őt meghívó alprogramban kezelni tudunk
  - lekezelhetjük ugyanabban a alprogramban is, de annak általában nincs értelme
- Kivételt kiváltani a `throw` utasítással tudunk:
  `throw new <kivétel típusa>(<kivétel szövege>);`
  - az utasításra a program futása megszakad, és a következő kivételelfogó utasításra kerül a vezérlés
  - a kivétel típusa lehet egy beépített kivételtípus (pl. `ArgumentException`, `IndexOutOfRangeException`, `Exception`), valamint mi is definiálhatunk kivétel típusokat
  - a kivétel szövegét nem kötelező megadni, ekkor üres lesz a `Message` tulajdonság

Kivételkezelés
Kivételek terjedése

- A kiváltott kivételek terjednek a programban az alprogramhívásokon keresztül, amíg le nem kezeljük őket
  - pl. ha egy alprogram meghív egy másikat, ő egy harmadikat, a harmadik szubrutin által kiváltott kivétel a másodikban, vagy az elsőben is kezelhetjük



  - ezáltal tetszőleges szintjén állíthatjuk meg a megszakítását a programnak, azon a szintjén, amely a kivételt megfelelő módon le tudja kezelni

Kivételkezelés
Kivétel-biztonság

- A programot *kivétel-biztos*nak (*exception-safe*) nevezünk, amennyiben garantáltan nem kerül abnormális állapotba
  - ehhez különböző invariánsokat garantálunk a program futása során
  - ettől függetlenül a program előállíthat hibás adatokat, illetve azokat el is mentheti
- A kivétel-biztonságnak a következő szintjeit tartjuk nyilván:
  1. *kivétel-biztonság mentes*: a program nem garantálja az invariánsok teljesülését, kivétel hatására terminálhat
  2. *minimális kivétel-biztonság*: a program menthet és előállíthat hibás adatokat, de nem kerülhet abnormális állapotba (ez egyszerűen elállítható a főprogramban való kivételkezeléssel)

Kivételkezelés
Kivétel-biztonság

  3. *alap kivétel-biztonság*: a program nem menthet hibás adatokat, de a kivételt okozó műveletek részben lefuthatnak, és okozhatnak mellékhatásokat
  4. *erős kivétel-biztonság* (változás-mentes garancia): a műveletek vezethetnek kivételhez, de ebben az esetben az eredeti adatok helyreállnak
  5. *kiváltás-mentes garancia*: minden kivétel a kiváltás szintjén kezelve van, tehát minden művelet hibamentessége garantált
- Sokszor a kiváltás-mentes garancia nem adható meg, de az erős kivétel-biztonság igen
  - pl. ha bővíteni akarunk egy vektort új elemmel, de nincs elég memória, akkor a korábbi elemek megmaradnak