Programozási nyelvek és módszerek

5. ELŐADÁS – KIVÉTELEK

Program készítésének célja

A célunk jó minőségű program készítése

Alapvető elvárás

- helyes – a specifikációnak megfelelően működik
- robosztus – nem várt helyzetekben sem történik katasztrófa

Hibalehetőségek

A programok futása közben hibák léphetnek fel.

Hardverhibák

- elromlott winchester miatti leállások,
- elfogy a memória, stb.

Hibalehetőségek

A programok futása közben hibák léphetnek fel.

Szoftverhibák

- a futtatott programban,
  - kísérlet nullával való osztásra
  - aritmetikai túlcsordulás, alulcsordulás
  - üres pointer/referencia dereferencia kísérlete
  - tömb hibás indexelése
  - hibás típus konverzió kísérlete
  - hibás input stb.
- az operációs rendszerben meglévő programozói hibák.

A program

megbízhatóságának növelése

érdekében

figyelni kell a hibákra

és meg kell próbálni

megelőzni,

vagy ha már bekövetkeztek,

kezelni

őket!

Hibakezelés históriája

Hogyan kezelték régebben?

- Abortált a program
- Minden alprogram valamilyen jelzést (hibaértéket) ad vissza
  - A lefutás OK
  - A lefutás nem sikerült

Példa

Tegyük fel, hogy szeretnénk beolvasni egy fájlból

```
ReadFile(f: in File; n1: out byte,  n2: out integer,
                     n3: out longint, n4: out real);

begin
   ReadByte(f,n1);
   ReadInt(f,n2);
   ReadLongInt(f,n3);
   ReadReal(f,n4);

end ReadFile;
```

Ellenőrzés lépésenként

Ha ellenőrizni akarjuk:

```
ReadFile(f: in File; n1: out byte, n2: out integer,
                          n3: out longint, n4: out real): boolean;
success:boolean;
begin

  success:=ReadByte(f,n1);
    if success then success:=ReadInt(f,n2);
      if success then success:=ReadLongInt(f,n3);
         if success then success:=ReadReal(f,n4);

            if not success then
              // ReadReal hiba kezelése,
            end if;

         else … ReadLongInt hiba kezelése end if ;
      else … ReadInt hiba kezelése end if ;
    else … ReadByte hiba kezelése end if ;

  return success;

end ReadFile;
```

Meggondolások

Nem biztos, hogy függvényt lehet/érdemes csinálni az eljárásból

- lehet, hogy nem visszatérési érték, hanem egy paraméter jelzi a sikert.

A kód sokkal bonyolultabb, az eredeti cél szinte elvész

- a későbbi karbantartás sokkal nehezebb!

Nem biztos, hogy a hívó figyeli a sikert jelző értéket!

Meggondolások

Lehetne, hogy globális változót használunk a lefutás helyességének figyelésére

- DE: osztott környezetben beláthatatlan következmények!

Hiba esetén meghívunk egy hibakezelő alprogramot

- és ha nem tesszük meg?

Elvárások

A hibák kezelése kapcsán a következő elvárásokat lehet megfogalmazni

1. meg tudjuk különböztetni a hibákat,
2. a hibát kezelő kód különüljön el a tényleges kódtól,
3. megfelelően tudjuk kezelni - a hiba könnyen jusson el arra a helyre, ahol azt kezelni kell,
4. kötelező legyen kezelni a hibákat

KIVÉTELKEZELÉS

Megoldás elemzése

Az első kikötés, hogy meg tudjuk különböztetni a hibákat.

- Ez általában a fellépés helyén történik.

A második kikötés azt szeretné elérni, hogy a programot először látó ember is azonnal el tudja különíteni a hibakezelő és a működésért felelős részeket.

- Ez könnyebben továbbfejleszthető, karbantartható programokhoz vezet.

Példa

```
gyumolcs {
    alma();
    //...
    korte();
    //...
    barack();
}
```

Példa hagyományos módon

```
gyumolcs {
   ret1=alma();
   if (ret1==ok) {
      //...
      ret2=korte();
      if (ret2==ok) {
         //...
         ret3=barack();
         if (ret3==ok) {
            return ok;
         }
         else { /* 3. hiba kezelése */}
      }
      else { /* 2. hiba kezelése */ }
   }
   else {/* 1. hiba kezelése */ }
}
```

A példa kivételkezeléssel

```
class AlmaException extends Exception {}
class KorteException extends Exception {}
class BarackException extends Exception {}

gyumolcs {
   try {
      alma(); /*küldhet AlmaException-t*/
      korte();
      barack();
   }
   catch(AlmaException hiba1) { /* 1. hiba kezelése */ }
   catch(KorteException hiba2) { /* 2. hiba kezelése */ }
   catch(BarackException hiba3) { /* 3. hiba kezelése */ }
}
```

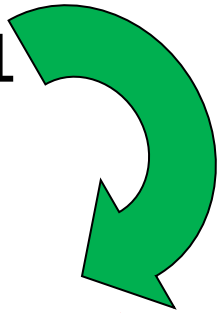
A hiba jusson el oda, ahol kezelni kell

A harmadik kikötés arra vonatkozik, hogy egy alacsonyabb szinten jelentkező hibát nem biztos, hogy az adott szint le tud kezelni, ezért a megfelelő helyre kell eljuttatni.

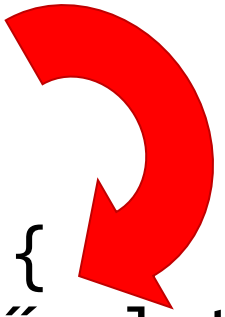
- A programok többszintűek
- Példa
  - egy fájlt nem tudunk megnyitni
  - egy üres veremből akar valaki kiolvasni

Példa

```
gyumolcs {
     alma();
     //folytatás1
}
```



```
          alma {
               jonatan();
               //folytatás2
          }
```



```
                    jonatan {
                         //műveletek
                    }
```

Hiba terjesztése hagyományosan

Tegyük fel, hogy a `jonatan()`-ban fellépő hibát a `gyumolcs()` kell lekezelje:

```
gyumolcs {
   ret=alma();
   if(ret!=ok) {
      //hibakezelés
   }
   else {
      //folytatás1
   }
}
```

```
alma {
      ret=jonatan();
      if(ret!=ok) {
         return hiba;
      }
      else {
         //folytatás2
      }
   }
jonatan {
      //műveletek
      if( művelethiba ) {
         return hiba;
      }
   }
```

Hiba terjesztése kivételkezeléssel

Kivételkezeléssel a felfelé terjesztést egyszerűen és kevesebb módosítással lehet megoldani:

```
gyumolcs {
   try {
      alma();
      //folytatás
   }
   catch(VmiExc) {
      //hibakezelés
   }
}
```

```
jonatan throws VmiExc {
   //műveletek
}


alma throws VmiExc {
…..
            jonatan();
   //folytatás
}
```

Kivétel és nem hiba

… általában..., kivéve...

```
type Napok is (Vasarnap, Hetfo, Kedd, … Szombat);


function Holnap(Ma: Napok) return Napok is
begin
  return Napok'Succ(Ma);
exception
  when Constraint_Error => return Napok’First;

end Holnap;
```

Kérdések

Hiba- vagy kivételkezelést ad-e a nyelv?

Hogyan kell kivételeket definiálni-kiváltani-kezelni?

Milyen a kivételkezelés szintaktikai formája

Mihez kapcsolódik a kezelés: utasítás-, blokk- vagy eljárás/függvényszintű?

A kivételekből lehet-e kivételcsoportokat, kivételosztályokat képezni, amelyeket a kivételkezelőben egységesen lehet kezelni?

Kérdések

Paraméterezhető-e a kivétel információ továbbadás céljából?

Támogatja-e a nyelv explicit módon valamilyen végső tevékenység („finally”) megadását?
Ha nem, akkor tudjuk-e szimulálni azt?
Milyen megszorításokkal?

A nyelv biztosít-e olyan nyelvi konstrukciót, amelynek nem kell explicit módon megadni, hogy mely kivételt kell továbbadni?

Kérdések

Újrakezdhető-e az alprogram (blokk) a hiba kezelése után?

Megadható-e /meg kell-e adni, hogy egy alprogram milyen lekezeletlen kivételeket küldhet?

Párhuzamos környezetben vannak-e speciális kivételek?

Mi történik a váratlan kivételekkel?

Kiváltható-e kivétel kivételkezelőben?

Melyik kivételkezelő kezeli a kivételt?

C++

Kivétel lehet egy objektum, aminek tetszőleges a típusa.

Hasznos, ha definiálunk osztályokat a felhasználó kivételeihez

Például

- `class MyException {};`

C++

Néhány predefinit kivétel:

- `bad_cast`: ha a dynamic_cast operator nem használható egy referenciára,
- `bad_typeid`: ha a typeid operátora egy null pointer dereferenciája.
  (mindkettő a typeinfo.h-ban deklarálva).

C++

Kivétel kiváltása

- `throw [expression] ";"`
  A kifejezés egy időszakos objektumba kerül.
- A normál program lefutás megszakad, a vezérlés a legközelebbi megfelelő kezelőhöz kerül
- Ha nincs kifejezés: csak kivételkezelőben, illetve innen hívott függvényben lehet, az aktuális kivétel újrakiváltása.
- A fordító nem biztos, hogy ellenőrzi, ha `throw` utasítás kivétel-objektum nélkül: `terminate` függvény hívása.

C++

Példák

- `throw 5;`
- `throw "An exception has occurred";`
- `throw MyException();`
- `throw;`

További információk: a kivételosztályban lehetnek nyilvános attribútumok, konstruktor beállíthatja stb.

```
class ExceptionWithParameters {
  public:
   int a,b;
   ExceptionWithParameters(int a0,int b0)
                                { a=a0; b=b0; }
   };
   // ...
   throw ExceptionWithParameters(0,1);
```

C++

Kivételek kezelése - try blokkal:

- `try compound_statement handler_list`
  `ahol`
  `handler_list = handler{handler}`
- `handler = catch "(" exception_declaration ")"`
  `compound_statement`
  `exception_declaration = type [ident] |"..."`

C++

A dobott kivételeknek megfelel egy catch ág, ha a következő feltételek közül valamelyik teljesül:

- A két típus pont ugyanaz.
- A catch ág típusa publikus bázisosztálya az eldobott objektumnak.
- A catch ág típusa mutató, és az eldobott objektum olyan mutató, melyet valamely standard mutató konverzióval át lehet konvertálni a catch ág típusára.

A kezeletlen kivételek továbbgyűrűződnek a hívó try blokkban, majd az azt hívóban.

- A legkülső try blokk után a *terminate* függvény kerül meghívásra.

C++

Például: class A {};

- ```
  class B: public A {};
  class C: public B {};
  class D: public A {};
  class X {};
  class AX: public A, public X {};
  catch (A) // A,B,C,D, AX típusúakat kap el
  catch (A*) // A*,B*,C*,D*,AX*
  catch (X&) // X és AX
  catch (const B&) // B és C
  catch (char*) // char*
  catch (void*) // tetszőleges pointer típusút
  ```
  ```

C++

Mindig az első kezelőt választja ki

- ha rossz sorrendet írunk, nem hiba!
- ```
  try { // ...
     }catch (A) {/* ... */}
      catch (B) {/* ... */}//soha nem jön ide!
  ```
- ```
  try {// ...
     }catch (B) { /* ... */ }
      catch (A) { /* ... */ } // így OK.
  ```

C++

Lehet a kivételobjektumra is hivatkozni:

- ```
  catch (ExceptionWithParameters e) {
      cout<< e.a; // kiírja
  }
  ```

Lehet ...

- ez bárminek megfelel - utolsó legyen a try-blokkban

A `terminate` hívása általában abort-ot jelent

- A felhasználó a `set_terminate`-tel megadhat mást is
- ez is le kell állítsa a programot.

Miközben a vezérlés átadódik a kivételkezelőnek, destruktort hív minden automatikus objektumra, ami a try-blokkba való belépés óta keletkezett.
  ```

C++

Kivétel specifikációk

- `throw "(" [type {"," type}] ")"`
- Például: `void f(int x) throw(A,B,C);`

Az f függvény csak A,B és C típusú kivételt generál, vagy azok leszármazottait.

- `int f2(int x) throw ()` - f2 nem válthat ki kivételt!
- `void f3 (int x)` - f3 bármit kiválthat!

Ha specifikáltunk lehetséges kivételtípusokat, akkor minden más esetben a rendszer meghívja az `unexpected()` függvényt

- Az alapértelmezett viselkedése a `terminate()` függvény meghívása
- Ez a `set_unexpected` segítségével szintén átállítható.

C++

Példák:

- ```
  class X {};
  class Y: public X {};
  class Z {};
  void A() throw(X,Z) // A X,Y,Z-t dobhat
  { /* ... */ }
  void B() throw(Y,Z)
  {
    A(); // unexpected, ha A X-t dob, ok  Y,  Z
  típusúra
  }
  void C(); // semmit sem tudunk C-ről
  ```
- ```
  void D() throw()
    {
      C(); // ha  C bármit kivált, unexpected -t hív
          throw 7; // unexpected-t hív
    }
  ```

Java

```
try {
    …
    throw new EgyException (“parameter”);
}
catch (típus változónév) {
    ...
}
finally {
    ...
}
```

Java

A C++ szintaxistól nem sokban különbözik.

- Eltérések a szemantikában.

A továbbiakban csak az eltérések:

- `finally` nyelvi konstrukció, ezzel a Java megbízhatóbb programok írását segíti elő.

Java

Egy függvény által kiváltható kivételek specifikálása:

- `void f (int x) throws EgyikException, MasikException;`

Egy szálon futó program esetén ha a virtuális gép nem talál a kivétel kezelésére alkalmas kódot, akkor a VM és a program is terminál.

- Ha többszálú a program, akkor egy `uncaughtException()` metódus fut le.

Java

Minden kivétel a `java.lang.Throwable` leszármazottja.

Ha olyan kivételt szeretnénk dobni, amely nem a `Throwable` leszármazottja, akkor az fordítási hibát okoz.

A kivételek két nagy csoportba sorolhatóak

- ellenőrzöttek: `Exception` leszármazottjai
- nem-ellenőrzöttek: `Error` leszármazottjai

Java

Azért volt arra szükség, hogy a kivételeket a fenti két csoportba sorolják, mert számos olyan kivétel van, amely előre nem látható és fölösleges lenne mindenhol lekezelni őket.

- Ezeket a kivételeket nevezzük nem-ellenőrzött kivételeknek.
- Például nem ellenőrizzük le minden utasítás végrehajtása előtt, hogy van-e elég memóriánk stb.

Java

Sajnos, a Java a fenti két csoportosítást nem konzisztens módon végzi

- Az `Exception` egyik gyermek osztálya, a `RuntimeException` és leszármazottjai sem ellenőrzöttek.

Az ellenőrzött kivételek esetén fordítási hiba lép fel, ha nincsenek specifikálva vagy elkapva.

- Továbbá ha olyan ellenőrzött kivételt kívánunk elkapni, amely hatókörön kívül van.

Vermes példa

```
class VeremException extends Exception {}

class VeremMegteltException extends VeremException
{
	private int utolso;
	public VeremMegteltException (int i) {
		utolso = i;
	}
	public int miVolt () {
		return utolso;
	}
}
```

Vermes példa

```
class Verem {
   final static public int MERET = 10;
   private int tarolo [] = new int [MERET];
   private int mutato = 0;
   public void betesz (int i) {
     try {
       if (mutato < MERET)
          tarolo [mutato++] = i;
       else
          throw new VeremMegteltException (i);
     } catch (VeremMegteltException e) {
        System.out.println (e.miVolt () + " nem fert be"); throw;}
     finally {
       System.out.println ("finally: mindig lefutok!");
     }
   }
}
```

Mi a gond ezzel?

Java

Néhány predefinit nem ellenőrzött kivétel

A `RuntimeException` leszármazottjai

- `ArithmeticException`
- `ClassCastException`
- `IndexOutOfBoundsException`
  - `ArrayIndexOutOfBoundsException`
  - `StringIndexOutOfBoundsException`
- `NullPointerException`.

Az `Error` leszármazottjai

- `OutOfMemoryError`
- `StackOverflowError`
- stb.

Java

Predefinit kivételek előfordulása:

```
class A {// ...
}
class B extends A {// ...
}
```

Java

```
class C {
    void X() {
      A a= new A;
      B b= (B)a; // ClassCastException
    }
    void Y() {
      int ia[]= new int[10];
      for (int i=1; i<=10; i++)ia[i]=0;
   /* amikor i==10 lesz,  ArrayIndexOutOfBoundsException  */
       }
    void Z() {
      C c=null;
      c.X(); // NullPointerException
    }
  }
```

Java

Kivételek kezelése

```
try {
  // utasítások
}
catch (MyException e) {
  // utasítások MyException kezelésére
}
catch (AnotherException e) {
  // utasítások AnotherException kezelésére
}
finally {
  // mindig végrehajtódó utasítások
}
```

Java

```
class A extends Exception {}

class B extends A {} ...

try {
    // ...
  }
  catch (A a) { /* ... */ }
  catch (B b) { /* ... */ }

void MyMethod() throws MyException {
    // utasítások
    throw new MyException();
    // utasítások
  }
```

Java

Példa

```
class ExcA extends Exception {}
void A() throws ExcA {
    // ...
    throw new ExcA();
    // ...
}
```

Java

```
void B1() {
    A(); // Hiba: ExcA nincs lekezelve B1-
ben
}
void B2() {
  try {
     A(); // OK
  } catch (ExcA e) {
     // exception kezelése
  }
}
```

Java

```
void B3() throws ExcA {
    A(); // OK
}

class ExcD extends RuntimeException {}
  void D() throws ExcD {
    // ...
}
```

Java

```
void E() {
    D(); // ok, ExcD leszármazottja
         //RuntimeException-nak, így nem kell jelezni
}

class ExcF extends ExcA {}
    void F() throws ExcA {
      throw new ExcF();   // OK
}
```

Eiffel

Újrakezdés

- A komponens írásakor egy kivétel lehetőségét előre lehet látni és egy alternatív megoldást találni a szerződés betartatására.
- Ekkor a végrehajtás megpróbálja ezt az alternatívát.

Szervezett pánik

- Ha nincs rá mód, hogy teljesítsük a szerződést, akkor az objektumokat egy elfogadható állapotba kell hozni (típusinvariáns helyreállítása), és a felhasználónak jelezni kell a kudarcot.

Eiffel – Újrakezdés

```
try_once_or_twice is
  local
    already_tried : BOOLEAN
  do
    if not already_tried then
      method_1
    else
      method_2
    end
  rescue
    if not already_tried then
      already_tried := true;
      retry
    end
 end -- try_once_or_twice
```

Példa

```
transmit: (p: PACKET)
-- Transmit packet `p'
require
      packet_not_void: p /= Void
local
      current_retries: INTEGER
      r: RANDOM_NUMBER_GENERATOR
do
      line.send (p)
rescue
      if current_retries < max_retries then
            r.next
            wait_millisecs (r.value_between (20, 500))
      current_retries := current_retries + 1
            retry
      end
end
```

Eiffel – Szervezett pánik

```
attempt_transaction(arg : CONTEXT) is
  -- megpróbáljuk a transaction-t arg argumentummal;
  -- ha nem sikerül, az akt. obj. invariánsát visszaállítjuk
  require
     ...
  do
     ...
  ensure
     ...
  rescue
     reset(arg)
 end -- attempt_transaction
```

Eiffel

```
 default_rescue is
do
end --default_rescue

 class C creation
   make ...
inherit
   ANY
      redefine default_rescue end
   ...
feature
   make, default_rescue is
       -- nincs előfeltétel
      do ...
      end;
   ...
end -- class C
```

C# - .NET

Közös elv alapján a .NET-ben

Nagyon hasonlít a Javához, de van különbség is

Hasonlóság:

- try – catch blokk, (ellenőrzi a jó sorrendet)
- finally lehetősége
- közös ősosztály (System.Exception)

Különbség:

- Nincs exception-specifikáció

C# - .NET

Miért nincs a C#-ban ellenőrzött kivétel?

- Verziókezelés
  - Ha egy következő verzióban egy metódus új kivételt dob, akkor az őt hívó metódust is változtatni kell
  - Ez máshol is probléma, ha le akarjuk kezelni az új kivételt, de legtöbbször nem kezelik
  - 10 az 1-hez a try-finally és a try-catch aránya

Irodalom: http://artima.com/intv/handcuffsP.html

C# - .NET

Miért nincs a C#-ban ellenőrzött kivétel?

- Méretezhetőség:
  - Nagy projektekben, négy-öt alrendszerrel a sok kivétel már kezelhetetlenné válik
  - Ilyenkor keletkezik sok catch {} üresen
- Nem szigorú szabályok kellenek a kezeletlen kivételek ellen, hanem elemző eszközök, amelyek felkutatják a gyanús kódokat, lehetséges réseket

C# - .NET

Példák:

```
using System;
class ExceptionTestClass {
       public static void Main() {
          int x = 0;
                  try {
                        int y = 100/x;
     }
     catch (ArithmeticException e) {
                Console.WriteLine("ArithmeticException Handler: {0}",
                                        e.ToString());
     }
     catch (Exception e) {
                Console.WriteLine("Generic Exception Handler: {0}",
                                              e.ToString());
     }
   }
}
```

C# - .NET

```
using System;
class ArgumentOutOfRangeExample {
   static public void Main() {
      …
      try {
      …
      }
      catch (ArgumentOutOfRangeException e) {
       Console.WriteLine("Error: {0}",e);
      }
      finally {
        Console.WriteLine(„It is always executed.");
      }
   }
}
```

C# - .NET

Saját exception-osztály:

```
using System;
public class EmployeeListNotFoundException:
            ApplicationException {
public EmployeeListNotFoundException() { }
public EmployeeListNotFoundException(
            string message) : base(message) { }
public EmployeeListNotFoundException(
            string message, Exception inner) :
            base(message, inner) { }
}
```

Delphi

Eltérő szintaktikára példa

```
begin
  Try
    // The code we want to execute
    ...
  Except
    // Exception handling
    ...
  Finally
    // Finally block
    ...
  end;
end;
```

Delphi

Több kivételtípus kezelése

```
except
    // IO error
    On E : EInOutError do
      ShowMessage('IO error : '+E.Message);
    // Dibision by zero
    On E : EDivByZero do
      ShowMessage('Div by zero error : '+E.Message);
    // Catch other errors
    else
      ShowMessage('Unknown error');
 end;
```

Delphi

Kivétel kiváltása

- `raise object at address`
- Példa
  - `raise EMathError.Create;`
  - `raise Exception.Create at @MyFunction;`

SWIFT

A kivételeket az Error protokolt megvalósító felsorolási típusok segítségével reprezentálja

- SWIFT esetén a protokoll megközelítőleg a Java interfész fogalomnak felel meg

Például

```
enum VendingMachineError: Error {
    case invalidSelection
    case insufficientFunds(coinsNeeded: Int)
    case outOfStock
}
```

SWIFT

Az egyes esetek paraméterezhetők, amivel a hiba / kivétel specifikusabban megadható.

Kiváltani a throw utasítással lehet

- `throw VendingMachineError.insufficientFunds(coinsNeeded: 5)`

SWIFT

Kivételek kezelése

- Egyrészt, a Java-hoz hasonlóan egy függvénynek jeleznie kell, hogy kivételt dobhat
  - `func canThrowErrors() throws -> String`
  - `func cannotThrowErrors() -> String`
- Ha kritikus függvényt hívunk, akkor `try` kulcsszóval kell tenni
  - `try canThrowErrors()`

SWIFT

A kapott kivételt vagy kezelni kell, vagy jelölni, hogy tovább tud dobódni.

Kivétel kezelés:

```
do {
    try buyFavoriteSnack(…)
} catch VendingMachineError.invalidSelection {
    …
} catch VendingMachineError.outOfStock {
    …
} catch VendingMachineError.insufficientFunds(let
c) {
    …
}
```

SWIFT

További lehetőségek

- Értékadásnál, ha nem sikerült, akkor null értéket ad az új változónak
- Ezt később lehet kezelni
  - Optional – formájában

```
let x = try? someThrowingFunction()
```

  - Összeköthető feltétel vizsgálattal is

```
func fetchData() -> Data? {
    if let data = try? fetchDataFromDisk()   { return data }
    if let data = try? fetchDataFromServer() { return data }
    return nil
}
```

SWIFT

További lehetőségek

- Amennyiben biztosak vagyunk, hogy a hívott függvény valóságban nem dobhat kivételt akkor

```
let photo = try! loadImage(atPath: "./…")
```

- Ebben az esetben, ha mégis keletkezik kivétel, akkor a futás megszakad és egy futás idejű hiba keletkezik

SWIFT

Hibától függetlenül lefutó kód

```
func processFile(filename: String) throws {
    if exists(filename) {
        let file = open(filename)
        defer {
            close(file)
        }
        while let line = try file.readline() {
            …
        }
        // close(file) is called here, at the end of the scope.
    }
}
```

- Tehát a defer részben megadott hívás a blokk befejeződése előtt lefut