**Lexikális egységek. Adattípusok. Nevesített konstans. Változó. Kifejezések. Utasítások. Programegységek. Paraméter kiértékelés, paraméterátadás. Blokk. Hatáskörkezelés, láthatóság. Absztrakt adattípus. Kivételkezelés.**

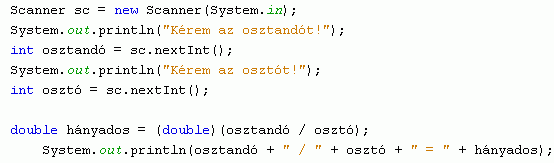
* Lexikális egységek: Hat osztályba sorolhatóak: Azonosítók, kulcsszavak, állandók, karakterláncok, operátorok és egyéb szeparátorok. A szóközöket, tabulátorokat, újsorokat, megjegyzések(közös nevükön üres helyek) a fordító nem veszi figyelembe.
  + Megjegyzések: A /\* karakterek megjegyzést vezetnek be , amely a \*/ karakterrel zárul.
  + Azonosítók: Az azonosítók a betűk és számjegyek sorozata; az első karakter betű kell, hogy legyen, A aláhúzásjel betűnek számít, A nagy és kisbetűk különbözőek.
  + Kulcsszavak: Az alábbi azonosítók a nyelv kulcs szavai, így egyéb célra nem használhatók: int else for …..
  + Állandók: Többfajta állandó van.
    - Egész állandók: A számjegyek sorozatát tartalmazó egész típusú (integer) állandót a fordító oktálisnak tekinti, ha 0-val (a nulla számjeggyel) kezdődik, egyébként decimálisnak veszi. A 8 és 9 számjegyek oktális értéke 10, ill. 11 . Az olyan számjegysorozatot, amelyet 0X vagy 0x (a 0 a nulla számjegy) előz meg, a fordítóprogram hexadecimális egésznek tekinti. Hexadecimális számjegyek az a-tól, ill. A-tól f-ig, ill. F-ig elhelyezkedő karakterek, amelyeknek értéke 10, . . ., 15. Azt a decimális állandót, amelynek értéke meghaladja a gépen ábrázolható legnagyobb előjeles egészt, a fordítóprogram long-nak veszi; hasonlóképpen long lesz az az oktális vagy hexadecimális állandó, amelynek értéke meghaladja a legnagyobb, elôjel nélküli gépi egészt.
    - Explicit long állandók: Az a decimális, oktális vagy hexadecimális egész, amelyet közvetlenül l („el” betű) vagy L követ, long (hosszú) állandó. Amint arról az alábbiakban szó lesz, bizonyos gépeken az int és long értékek azonosak.
    - Lebegőpontos állandók: A lebegőpontos állandó egész részből, tizedespontból, törtrészből, e-ből vagy E-bôl és (esetleg előjeles) kitevőből áll. Mind az egész, mind a tört rész számjegyek sorozata. Akár az egész, akár a tört rész hiányozhat (de mind a kettő nem!); ill. a tizedespont vagy az e és a kitevő közül az egyik szintén elmaradhat. Minden lebegőpontos állandó duplapontosságú.
  + Karakterláncok: A karakterlánc idézőjelek közé zárt karaktersorozat: "….”. A karakterlánc típusa szerint karaktertömb, tárolási osztálya static, és a megadott karakterek inicializálják. Az egyes karakterláncok, még az azonos módon leírtak is, külön egységet képeznek. A fordító minden karakterlánc végére elhelyezi a \0 nullabyte-ot abból a célból, hogy a karakterláncot vizsgáló programok megtalálják a karakterlánc végét. A karakterláncon belül elhelyezett „ idézôjelet \ kell, hogy megelôzze; a karakterállandóknál ismertetett összes escape-szekvencia használható. Végül megjegyezzük, hogy az \-t és az azt közvetlenül követô újsort a fordító nem veszi figyelembe
  + Operátorok:+ - \* % /
  + Szeparátorok: space tab ujsor stb
* Adattípusok: A programok alapvető adatobjektumai a változók és az állandók. A deklarációk felsorolják a használni kívánt változókat, közlik a típusukat, valamint az esetleges kezdeti értéküket.
* Nevesített konstans: Egy nevesített konstansnak három komponense van (típus, név, érték). Muszáj mindig deklarálni. Egy nevesített konstans a neve által van reprezentálva a forráskódban. A név mindig egy érték komponenst takar. Az érték komponenst nem lehet változtatni futási időben, a deklarálásnál eldőlt az értéke. A nevesített constans szerepe az, hogy engedjük a programozót, egy gyakran használt értéknek egy jól leíró nevet adni. Az előnye, hogy elegendő a deklarálásnál megváltoztatni az értékét, nem kell az összes előfordulásánál változtatni.
* Változó.
* Kifejezések: A kifejezések a változókból és állandókból új értéket hoznak létre.
* Utasítások. A programozásban az utasítást úgy lehet felfogni, mint [programnyelv](https://wiki.prog.hu/wiki/Programoz%C3%A1si_nyelv) legkisebb elemét.
* Programegységek: -függvények
  + blokk(amely csak másik programegység belsejében helyezkedhet el, külső szinten nem állhat.),
  + (csomag)
  + taszk(a taszk mint programegység szolgál a párhuzamos programozás megvalósítására. A taszk tehát az a nyelvi eszköz, amely mögött folyamat áll.
* Paraméter kiértékelés, paraméterátadás. érték szerinti(nem változik), cím szerinti(változik az eredeti)
* Blokk.
* Hatáskörkezelés, láthatóság.
* Absztrakt adattípus. (Az absztrakt adattípus olyan adattípus, amely megvalósítja a bezárást vagy információ rejtést. Ez azt jelenti, hogy ezen adattípusnál nem ismerjük a reprezentációt és a műveletek implementációját. Az adattípus ezeket nem mutatja meg a külvilág számára. Az ilyen típusú programozási eszközök értékeihez csak szabályozott módon, a műveleteinek specifikációi által meghatározott interfészen keresztül férhetünk hozzá.)
* Objektum(Az objektum az [objektumorientált programozás](https://wiki.prog.hu/wiki/Objektumorient%C3%A1lt_programoz%C3%A1s) egyik alapeleme. Az objektumokat általában információt hordozó, és azokkal műveleteket vagy számításokat elvégezni képes egységként fogjuk fel.)
* **Kivételkezelés:** A **kivételkezelés** egy [programozási](https://hu.wikipedia.org/wiki/Sz%C3%A1m%C3%ADt%C3%B3g%C3%A9p-programoz%C3%A1s) mechanizmus, melynek célja a [program](https://hu.wikipedia.org/wiki/Sz%C3%A1m%C3%ADt%C3%B3g%C3%A9pes_program) futását szándékosan vagy nem szándékolt módon megszakító esemény (hiba) vagy utasítás kezelése. Az eseményt magát **kivétel**nek hívjuk. A hagyományos, szekvenciális és strukturált programozási kereteken túlmutató hibakezelésre, valamint magasabb szintű hiba detektációra, esetleg korrigálásra használható.

A Java kivételkezelésének célja a programfutás során keletkezett hibák kiszűrése és megfelelő kezelése. Az ilyen hibákat a Java platformon Exception-nek (kivételnek) nevezik. Két fő csoportjuk van: a futási időben és a nem futási időben keletkezett kivételek. Futási idejű kivételek az aritmetikai (pl. nullával való osztás), az indexeléssel kapcsolatos (pl. tömb nem létező eleméhez való hozzáférés), és a referenciával kapcsolatos (pl. objektumokra való hivatkozás) kivételek. Ezeket a kivételeket nem kötelező implementálni, de erősen ajánlott. Nem futási idejű kivételek a Java rendszerén kívül keletkeznek. Ilyenek az I/O műveletek során keletkező hibák (pl. a fájl nem található). Utóbbiakat kötelező lekezelni.

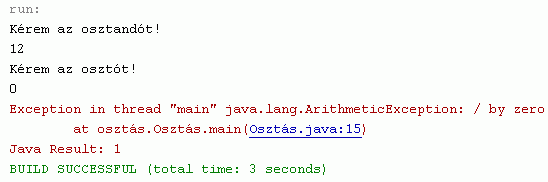
A felmerülő hibák sokféleségének kezelését a Java az objektum-orientált paradigma lehetőségeinek felhasználásával oldja meg: a kivételeket osztályok (ill. objektumaik) reprezentálják. Minden beépített - vagy általunk létrehozott - kivétel közös ősosztálya a Throwable osztály.

Amikor egy metódus futása során valamilyen hiba lép fel (pl. nullával való osztás, verem túlcsordulás, indexhatár túllépése, vagy a háttértároló megtelik stb.), akkor egy kivételobjektum (egy kivételosztály példánya) jön létre. Ez az objektum olyan információkat tartalmaz a kivétel fajtájáról és a program aktuális állapotáról, amelyeket a kivétel lekezelésekor felhasználhatunk. A kivételobjektum létrehozását és a futtatórendszer által történő lekezelését **kivételdobásnak** hívjuk. Ezeket ellenőrzött kivételeknek is nevezzük. Az ellenőrzött kivételeket kötelező lekezelni, amit a fordító már fordítási időben ellenőriz is.

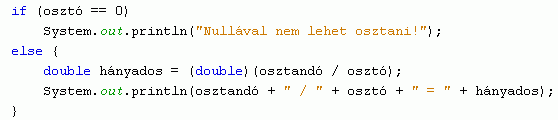
Nézzünk egy egyszerű példát a nullával való osztás kivételkezelésére! Az alábbi program alaphelyzetben semmiféle ellenőrzést nem végez az osztó értékére vonatkozóan:



A program futtatása ennek megfelelően 0 osztónál hibát jelez:



Hagyományos megoldásként az osztás művelete előtt ellenőrizzük az osztó értékét!



Ez a megoldás bonyolultabb ellenőrzések esetén az if feltételek garmadáját vonhatja magával, ami nagymértékben rontja a kód olvashatóságát és az esetleges programjavítás lehetőségét.

**A try blokk**

Kivétel mindig egy metódus törzsében keletkezhet. Ha ezeket a hibákat mind ugyanitt kezelnénk le, a programkód áttekinthetősége jelentősen romlana, és nem is lehet minden hibára előre felkészülni. Ezt szem előtt tartva olyan megoldás született, amelyben a hibás, kivételes programállapotot eredményezhető sorokat összefogva, s hozzá egy úgynevezett kivételkezelőt megadva a probléma elegánsan megoldható. Ekkor az adott blokkban fellépő kivételeket egységesen, egy helyen kezelhetjük, jól elválasztva egymástól a program fő feladatát végző, illetve a hibák lekezeléséért felelős kódrészt. Ennek megvalósítására a try - catch - finally blokkszerkezet használható. Sorrendjük szigorúan kötött!

A try blokk utasításokat zár közre, amelyeket futási időben végig felügyelete alatt tart. Lehetőleg minél kevesebb utasítást tegyünk egy ilyen blokkba, mert kivétel keletkezése esetén csak a kivétel keletkezési helyéig hajtódnak végre a blokkbeli utasítások.

A try - catch - finally kivételkezelő kódblokk felépítése:

**try** {  
    <utasítások>  
}  
**catch** (<kivételtípus\_1> <változónév\_1>) {  
    <utasítások>  
}  
[**catch** (<kivételtípus\_2> <változónév\_2>) {  
    <utasítások>  
}]  
[**finally** {  
    <utasítások>  
}]

Néhány fontosabb kivételosztály:

* + - Exception (általános hiba, ez az osztály minden más kivételosztályt magába foglal)
    - IOException (általános I/O hiba)
    - FileNotFoundException (a fájl nem található)
    - EOFException (olvasás a fájl vége után)
    - IndexOutOfBoundsException (indextúlcsordulás)
    - NumberFormatException (számformátum hiba)
    - ArithmeticException (aritmetikai hiba)
    - IllegalArgumentException (érvénytelen argumentum)
    - InputMismatchException (az input adat nem megfelelő típusú)

**A catch blokk**

A catch blokkok a try blokkban keletkező - típusuknak megfelelő - kivételeket kezelik le. Minden try blokkhoz megadható tetszőleges számú (ha nincs finally blokk, akkor legalább egy) catch ág, amelyek az esetlegesen fellépő hibák feldolgozását végzik. Semmilyen programkód nem lehet a két blokk között! A catch ágak sorrendje sem mindegy, mert az első olyan catch blokk elkapja a kivételt, amelynek típusa megegyezik a kiváltott kivétellel, vagy őse annak. Ezért érdemes a specifikus kivételtípusoktól az általánosabb típusok felé haladva felépíteni a catch blokkokat. A hierarchia tetején álló Exception osztály nem előzhet meg más (leszármazott) kivételosztályokat, mert azok sosem hajtódnak végre. A fordító ilyen sorrend esetén hibát jelez.

Ha valahol kivétel keletkezik, akkor a futtató rendszer megpróbál olyan catch ágat találni, amely képes annak kezelésére. Az az ág képes erre, amelynek paramétere megegyező típusú a kiváltott kivétellel (vagy annak ősével), valamint amelynek a hatáskörében a kivétel keletkezett.

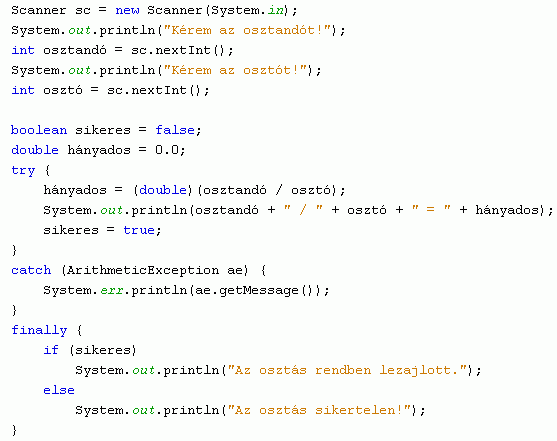
A catch ág egy szigorúan egyparaméteres metódusként fogható fel, amely paraméterként megkapja a fellépő kivételt, és ezt tetszőlegesen felhasználhatja a hibakezelés során. A kivételobjektumot nem kötelező felhasználni, mert sokszor a típusa is elég ahhoz, hogy a programot a hibás állapotból működőképes mederbe tereljük.

Ha a catch ágak egyike sem tudja elkapni a kivételt, akkor a beágyazó kivételkezelő blokkban folytatódik a keresés. Ha egyáltalán nincs megfelelő catch blokk, a program befejeződik.

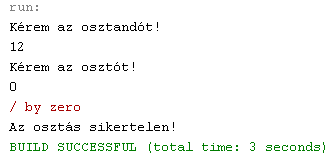
**A finally blokk**

A finally ág akkor is lefut, ha volt kivétel, akkor is, ha nem. Ebben a tetszőlegesen felhasználható blokkban kezdeményezhetjük pl. a nyitott fájlok bezárását, amit - függetlenül attól, hogy a megelőző try blokkban volt-e kivétel, vagy sem - mindig illik megtennünk. Ezt az ágat nem kötelező létrehozni, kivéve, ha nincs egyetlen catch ág sem. A kivétel lekezelése után a program végrehajtása a kivételkezelő kódblokk utáni utasításon folytatódik.

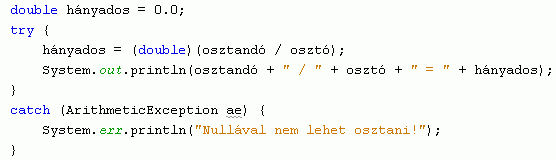
A kivételkezelő blokkok megismerése után alakítsuk át példaprogramunkat! Az osztás művelete kerüljön egy try blokkba, a hibát egy catch blokkban aritmetikai hibaként kezeljük le, valamint a finally blokkban jelezzük az osztás sikerességét!



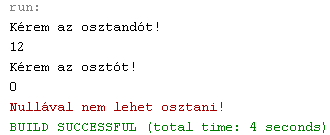
A futás eredménye nulla osztó esetén:



A vörös színben megjelenő "/ by zero" üzenet a kivételkezelő osztály hibaüzenete. Ebben a példában a finally ágra nincs is igazán szükség, így ezt elhagyva és a hibaüzenetet átírva egyszerűsíthető a program.



A kimenet is barátságosabb:



A fenti program forráskódja: [Osztás.java](http://java2.uw.hu/files/Oszt%C3%A1s.java).

**A throws használata**

Ha egy metódus végrehajtása közben kivétel keletkezik, és ezt nem akarjuk, vagy nem tudjuk helyben - az adott metóduson belül - lekezelni, akkor tovább kell küldenünk egy magasabb szintre, az adott metódust hívó metódus felé. Ez mindaddig folytatódhat, amíg el nem érjük azt a szintet (metódust), amely már elegendő információval rendelkezik a megfelelő intézkedések elvégzéséhez. Ezt a metódus fejlécében a throws kulcsszóval tehetjük meg, utána felsorolva azokat a kivételosztályokat (vagy egy ősüket), amelyeket nem kívánunk (vagy nem tudunk) helyben lekezelni. Legkésőbb a program main metódusában az ilyen kivételeket le kell kezelni.

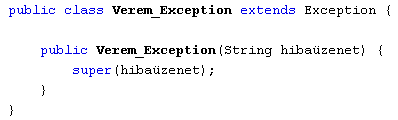
**Saját kivételek létrehozása a throw utasítással**

A **Java** kivételkezelése nyitott, ami azt jelenti, hogy bárki létrehozhat saját névvel és funkcionalitással ellátott kivételosztályokat, amelyeket célszerű az Exception osztályból származtatni. Az új kivételosztályok nevében célszerű az "Exception" szót is szerepeltetni, hogy utaljon annak szerepére.

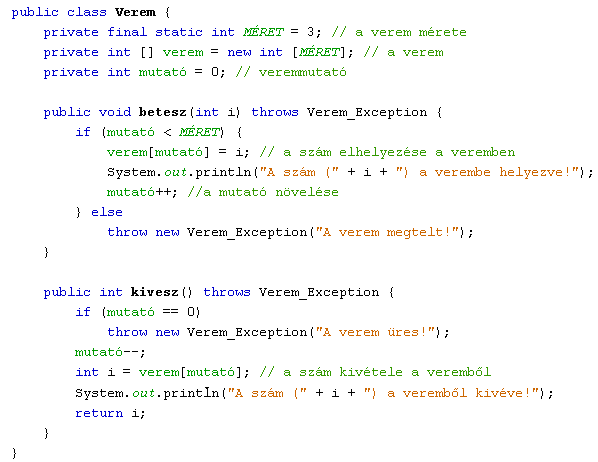
|  |
| --- |
| Saját kivételobjektum létrehozása:  **throw** **new** <Saját\_Kivételosztály\_név> ("Saját hibaüzenet"); |

Az alábbi program egy 3 db egész szám tárolására szolgáló verem adatszerkezetet modellez. Megvalósítjuk a verembe helyezést és a veremből való kivételt úgy, hogy ezen műveletek hibájának lekezelésére saját kivételosztályt használunk, és a hiba okát is megjelenítjük.

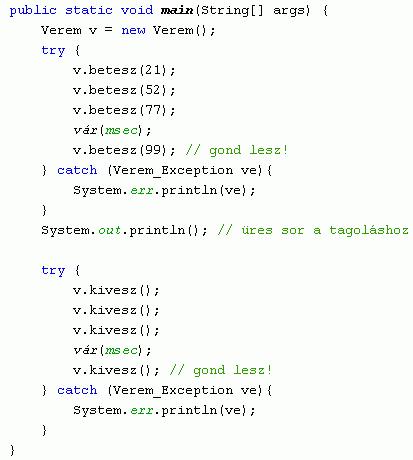
Első lépésként a saját kivételosztályunkat definiáljuk, amely hibaüzenet átvételére is alkalmas.



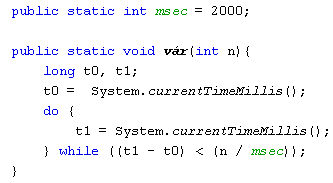
Majd következik a verem implementálása. A verem fontos jellemzője a mérete és a veremmutató. A betesz() metódus a paraméterként megadott számot a verem - mutató által jelzett - tetejére helyezi, a kivesz() - paraméter nélküli - metódus pedig a verem tetején levő számot "emeli ki". A szám helyének "kinullázása" nem kötelező, mert a verem telítettségét a mutató állása jelzi, nem a tartalma.

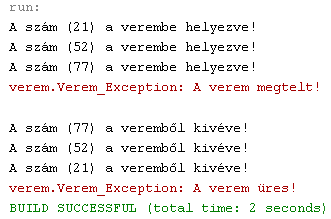


Próbáljunk háromnál több elemet elhelyezni a veremben, majd ezután a megtelt veremből háromnál többet kivenni!



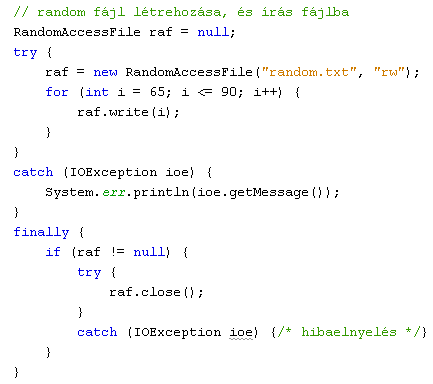
Az eredmény kiírása kissé rapszodikus sorrendben történik, mivel a kivételek kezelése külön programszálon fut, így nem a várt időpillanatban írják ki a hibaüzenetüket. Ezért a hibát kiváltó művelet előtt alkalmazzunk egy 2 mp-es késleltetést, amelyet a vár() nevű saját készítésű metódussal állítunk elő. Az msec osztályváltozó tárolja a késleltetés idejét milliszekundumban.

  
Az eredmény:

  
 Jól látható, hogy a 4. szám elhelyezése és az üres veremből való 4. kivétel is hibát okozott. Figyeljük meg a verem működését is! Az utoljára bekerült elem elsőként lett kivéve (LIFO adatszerkezet: Last In - First Out, utoljára be - elsőként ki).

A fenti program forráskódja: [Verem.rar](http://java2.uw.hu/files/Verem.rar).

Térjünk vissza egy rövid kiegészítés erejéig a 9. heti tananyag I/O műveleteire! Az ott alkalmazott forráskódok a jobb áttekinthetőség miatt nem tartalmazták a fájl bezárását megvalósító close() metódus biztonságos kivételkezelését, pedig ez a művelet is okozhat kivételt (pl. időközben megszűnt a kapcsolat a fájllal), így ezt az utasítást is érdemes try-catch szerkezetbe foglalni a következők szerint:

  
Figyeljük meg, hogy a close() utasítást "védő" try-catch szerkezet a fájlkezelő műveletek finally ágában helyezkedik el, tehát minden körülmények között lefut. Viszont a 9. fejezetben szereplő példák mindegyikében hiába volt a close() utasítás a try blokk által védett, ha előtte bekövetkezett egy kivétel (pl. a fájlba írás során), akkor soha nem került rá a vezérlés, tehát a fájl nyitva maradhatott.