Budapesti Műszaki és Gazdaságtudományi Egyetem

Java technológia

Kivételkezelés

Sipos Róbert
siposr@hit.bme.hu

2014. 02. 25.

Alapfogalmak

- A hibakezelés természetesen minden programban rendkívüli fontosságú.
- A Java nyelvi szintű hibakezelést tesz lehetővé.
- A hibakezelés úgynevezett **kivétel** (exception) objektumok segítségével történik.
- A kivétel objektumok a hiba helyén keletkeznek, és addig terjednek a programban az utasításblokkok és a hívási lánc mentén, amíg:
  - egy kódrészlet lekezeli a hibát,
  - a hibát a programon belül nem kezeljük, így a program hibaüzenettel leáll

Szóhasználat

- Egy kivétel **keletkezik** (occurs), **dobódik** (is thrown), vagy explicite **dobjuk** (throw).
- A kivétel lekezelése (handling) úgy történik, hogy a dobott kivételt **elkapjuk** (catch).
- Az el nem kapott kivétel **kifelé** (vagy **felfelé**) terjed (propagates) az utasításblokkok és a hívási lánc mentén.

Kulcsszavak

- A kulcsszavak megfelelnek a szóhasználatban említett kifejezéseknek:
  - Kivételt a **throw** kulcsszóval dobhatunk,
  - illetve a **catch** kulcsszóval kaphatunk el.
- A **catch** kulcsszóval a **try** kulcsszóval megjelölt blokkban (**try blokk**) keletkező (vagy a meghívott metódusokból dobódó) kivételeket kaphatjuk el.
- Ennek a konstrukciónak a neve **try-catch blokk**.

Szintakszis

- A try-catch blokk alakja az alábbi:

```
try {                                                          // try blokk
   ... utasítások ...
}
catch (KivételOsztály1 k) {                                    // 1. catch blokk
   ... 1. típusú kivétel kezelése, k az elkapott kivétel objektum ...
}
catch (KivételOsztály2 k) {                                    // 2. catch blokk
   ... 2. típusú kivétel kezelése, k az elkapott kivétel objektum ...
}
```

- Try-catch blokk esetén minimum egy catch blokk megléte kötelező.

Végrehajtás I.

- A try-catch blokk végrehajtása az alábbiak szerint történik:
  - Ha a try blokk utasításainak végrehajtása során **nem dobódott kivétel**, a try-catch blokk végrehajtása **normálisan befejeződik**.
  - Ha **kivétel dobódott**, sorban megpróbáljuk a keletkezett kivétel objektumot **illeszteni a catch blokkok fejében szereplő osztályokra**. Illeszkedés: a kivétel osztály megegyezik a catch blokk fejében adott osztállyal, vagy annak leszármazottja (`instanceof` reláció).
  - Ha **volt illeszkedés**, az adott **catch blokkot végrehajtjuk**, és a try-catch blokk normálisan befejeződik, illetve ha a catch blokkban újabb kivétel dobódik, kivétellel fejeződik be.
  - Ha **nincs illeszkedés**, a try-catch blokk **kivétellel fejeződik be**.

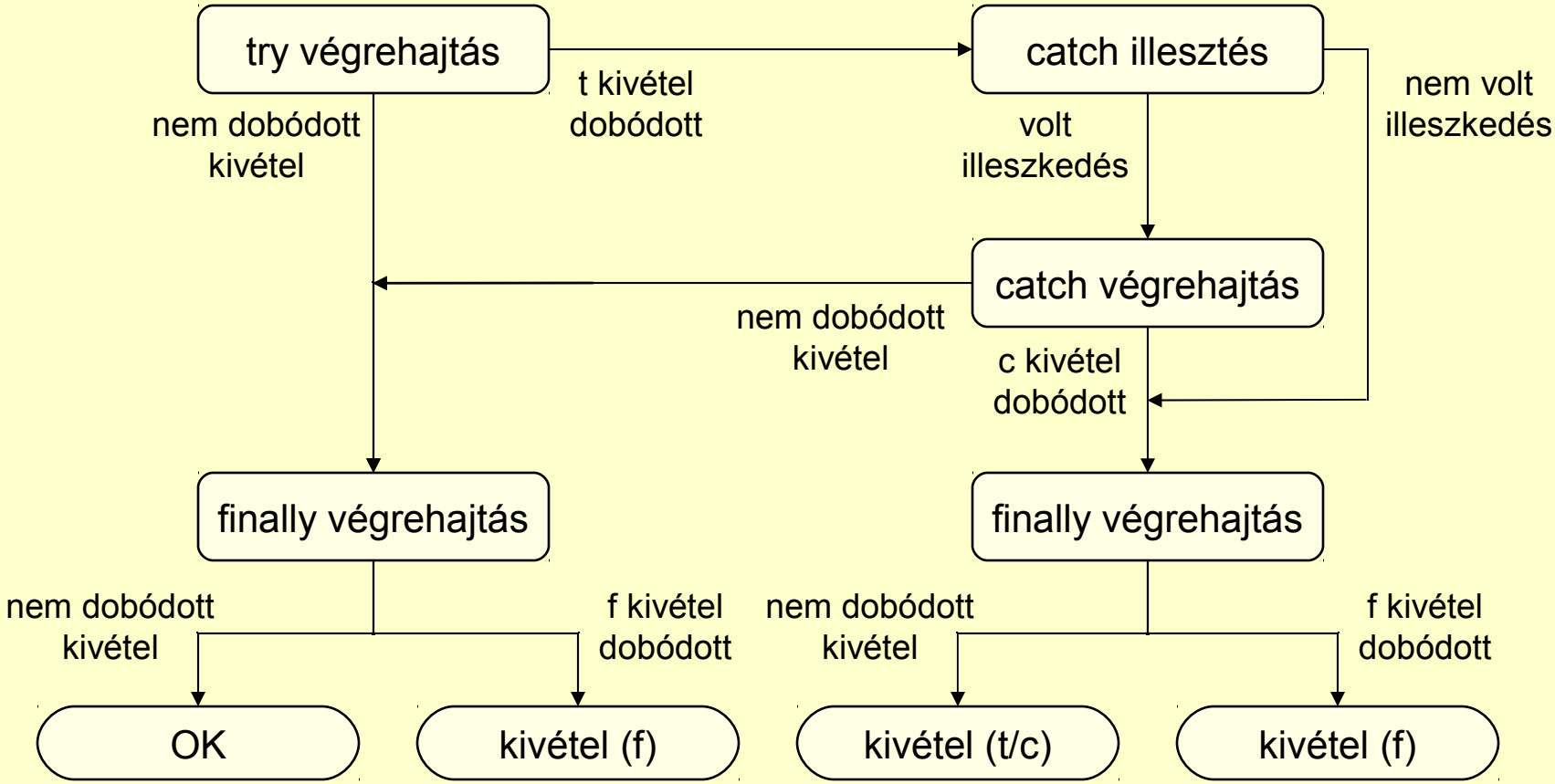
try-catch-finally

- Szükségünk lehet arra, hogy a try-catch blokk lefutásának eredményétől függetlenül végrehajtsunk egy kódrészletet. (A try-catch blokk normálisan is végetérhet, de kivétel is dobódhat belőle.)
- Erre szolgál a **finally blokk**. A catch blokkok után opcionálisan következhet egy darab finally blokk.
- Az ilyen konstrukció neve **try-catch-finally blokk**.

```
try {                                                    // try blokk
   ... utasítások ...
} catch (KivételOsztály k) {                            // catch blokk
   ... adott típusú kivétel kezelése ...
} finally {                                             // finally blokk
   ... utasítások ...
}
```

- Try-catch-finally blokk esetén a catch ágak opcionálisan el is hagyhatók.

Végrehajtás II.

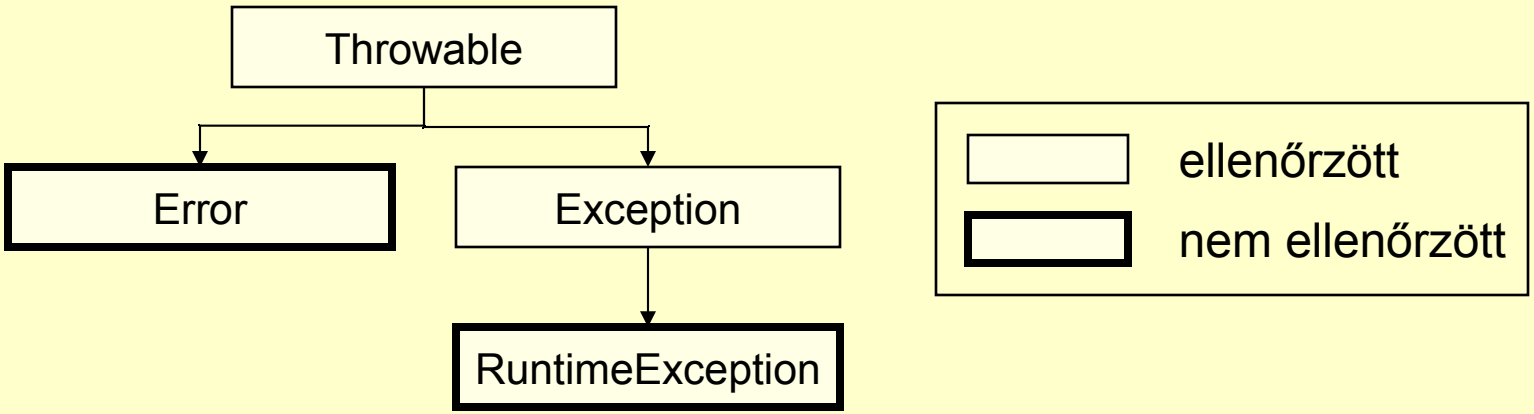


Kivételek típusai I.

- A kivételek a **Throwable** (dobható) osztályból származnak.
- A Throwable két leszármazottja az Exception, és az **Error**.
- Az **Exception** speciális leszármazottja a **RuntimeException**.
  - Az **Exception**-ből származó kivételek az úgynevezett **ellenőrzött kivétel**ek (checked exception). Ezek a kivételek a programban jól behatárolható helyen keletkező, értelmesen kezelhető hibák jelzésére szolgálnak. (például I/O hibák, konverziós hibák, stb.)
  - Az **RuntimeException**-ból származó kivételek az úgynevezett **nem ellenőrzött kivétel**ek (unchecked exception), amelyek olyan hibák jelzésére szolgálnak, amelyek a programban sok helyen keletkezhetnek, így kezelésük nehézkes lenne, és áttekinthetetlenné tenné a forráskódot. (Nullával való osztás, null referencia hiba, stb.)

Kivételek típusai II.

- A Throwable osztályból származnak, de nem ellenőrzött kivételnek számítanak az **Error**-ból származó kivételek is, ezeket többnyire a **Java virtuális gép dobja**, illetve a Java API-ban használatosak.
- Az alábbi ábra a kivételosztályok hierarchiáját foglalja össze.



Kivételek típusai III.

- Az ellenőrzött és nem ellenőrzött kivételekre némileg eltérő szabályok vonatkoznak:
  - Az **ellenőrzött kivételek** esetében, ha egy metódusban egy ilyen kivétel dobódhat, de a metódusban azt nem kezeljük, hanem azt akarjuk, hogy a kivétel továbbterjedjen a metódus hívója felé, a metódus fejében a **throws** kulcsszó után meg kell adnunk az kivétel osztály nevét. Erre azért van szükség, hogy a metódust használó tudja, milyen kivételek kezelésére kell felkészülnie.
  - A **nem ellenőrzött kivételek**et a throws kulcsszó után nem kell felsorolni, ezzel jelezve azt, hogy az ilyen kivételek nagyon sok helyen előfordulhatnak, és nem célszerű minden lehetségséges esetben lekezelni őket (de természetesen lehet).

Kivételek típusai IV.

- Az eddig elmondottakból nyilvánvaló, hogy a nem ellenőrzött kivételek használatával óvatosan kell bánni, mert éppen a kivételkezelés erejét veszítjük el azzal, hogy nem deklaráljuk, mely metódusokban milyen kivételek keletkezhetnek.
- Kivételek alapvetően három okból jöhetnek létre:
  - **throw paranccsal**, explicit kivételdobás, például:
    ```
    throw new Kivétel ("Baj van");
    ```
  - **kifejezéskiértékelés során**, implicit kivételdobás, például:
    ```
    int a = 3/0;              // nullával való osztás
    int[] a = new int [12]; a[23] = 0;// indexhiba
    ```
  - **metódushívás során**, például:
    ```
    byte b = inputStream.read ();      // I/O hiba
    ```

A Throwable osztály I.

- A Throwable osztály nyújt néhány, a hatékony hibakezelést megkönnyítő szolgáltatást.
- Egy Throwable objektum létrehozásakor megadhatunk egy **másik kivétel objektumot**, amely a létrehozott kivétel **oka** (cause). Ekkor az mondjuk, hogy a kivételt **becsomagoljuk** (wrap) egy új kivételbe. Ennek akkor van értelme, ha a programunk több funkcionális egységre bomlik.
- Az alábbi kódrészlet egy adattárolásért felelős modul része lehet:

```
try {
   ... file művelet ...  // a Java file műveletek IOException-t dobnak
} catch (IOException e) {                    // elkapjuk az IOException-t
   throw new StorageException ("Adattárolási hiba",e);
  // Továbbdobjuk, mint adattárolási hibát, belecsomagolva a hiba okát
}
```

A Throwable osztály II.

- A Throwable osztály másik szolgáltatása az, hogy a kivételekben nyilvántartja a **végrehajtási történet**et (stack trace), vagyis azt az információt, hogy a kivétel milyen hívások mentén terjedt a keletkezés helyétől. Ez az információ jelentősen megkönnyíti a hiba kijavítását.
- A végrehajtási történet természetesen a kivétel becsomagolásakor sem vész el (lásd az ok megadásának lehetőségét).
- A végrehajtási történethez programból is hozzá lehet férni, illetve a printStackTrace() metódussal a képernyőre is ki lehet íratni.

```
try {
   ... valamilyen művelet, amiből kivétel keletkezhet ...
} catch (Exception e) {     // Elkapjuk az összes ellenőrzött kivételt
   e.printStackTrace ();        // Kiíratjuk a végrehajtási történetet
   System.exit (-1);                          // Kilépünk a programból
}
```

Végrehajtási történet - forráskód

```
import java.io.*;                          // Az I/O műveletek miatt szükséges
public class Test {
   public void readFile (String name) throws IOException {
      FileReader fr = new FileReader (name);    // Megnyitjuk a file-t
   }
   public void read (String name) throws ReadException {
      try {
         readFile (name);
      } catch (IOException e) {      // Becsomagoljuk az IOException-t
         throw new ReadException ("Read error (" + name + ")",e);
      }
   }
   public static void main (String[] args) {
      try {
         new Test ().read ("ilyenfile.nincs");       // Nemlétező file
      } catch (ReadException e) {
         e.printStackTrace ();  // Kiíratjuk a végrehajtási történetet
      }
   }
}
```

Végrehajtási történet – az eredmény anatómiája

Kivételek és öröklődés

- A kivételekkel és az öröklődéssel kapcsolatban egy fontos szabályt kell megjegyeznünk.
- Legyen a **B** osztály az **A** osztály egy leszármazottja. A **B** osztály **n** metódusa az **A** osztály **m** metódusát írja felül. Ha az **m** metódus throws deklarációjában az ***X***, az **n** metódusában pedig az ***Y*** kivételhalmaz szerepel, akkor mindig igaz kell legyen, hogy $Y \subseteq X$.
- Vagyis: ha egy metódust egy leszármazott osztályban felülírunk, a throws deklarációban nem sorolhatunk fel az ősben szereplő throws deklarációhoz képest új kivételt.
- Ennek oka az, hogy ha egy **B** példányt valaki egy **A** típusú referencián keresztül használ, akkor az **n** meghívásakor csak az **A**-ban, az **m** metódus throws deklarációjában szereplő kivételekre van felkészülve.

Try-with-resources (Java 7)

- A Java 7-es verziójában megjelent a try-with-resources utasítás, melynek segítségével erőforrások deklarálhatóak a try blokkhoz.
- Az erőforrás objektumok lezárásra fognak kerülni a try blokk kimenetelétől függetlenül (nincs szükség a finally blokkban kezelni).
- Az erőforrás objektumoknak implementálniuk kell a `java.lang.AutoCloseable` interfészt.

```
try
(BufferedReader br = new BufferedReader(new FileReader(filename))) {
   ...
} catch (IOException e) {
   ...
}
```