Magasszintű programozási nyelvek 2

Kollekciók – Dr. Tiba Attila

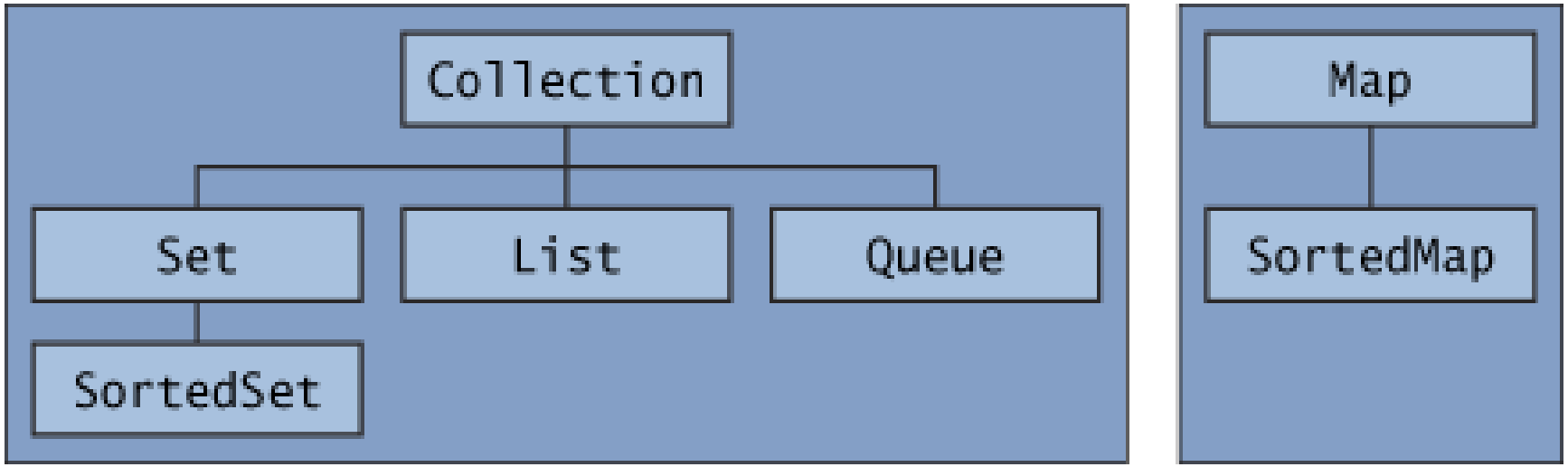
A kollekció keretrendszer – Java Collections Framework (JCF)

- A kollekciók (konténerek) olyan objektumok, melyek célja egy vagy több típusba tartozó objektumok memóriában történő összefoglaló jellegű tárolása, manipulálása és lekérdezése.
- A kollekció keretrendszer egy egységes architektúra, ami a kollekciók megvalósítására, kezelésére szolgál.
- Elemei:
  - Interfészek: absztrakt adattípusok, amelyek a kollekciókat reprezentálják. Lehetővé teszik a kollekciók implementáció független kezelését.
  - Implementációk: a kollekció interfészek konkrét implementációi.
  - algoritmusok: azok a metódusok, amelyek hasznos műveleteket valósítanak meg, mint például keresés, rendezés különböző kollekciókon.
- C++ Standard Template Library (STL)

A kollekció keretrendszer használatának előnyei

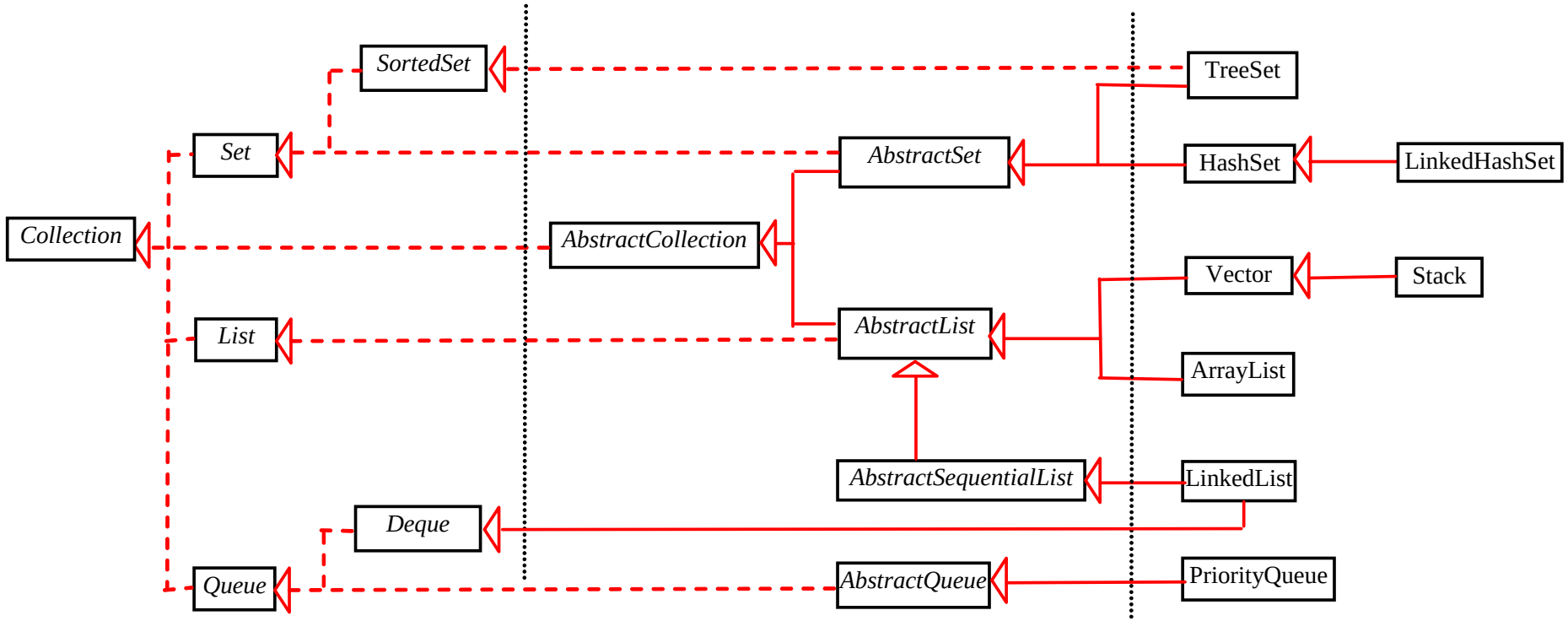
- Csökkenti a fejlesztési időt
- Növeli a program sebességét és minőségét
- Megkönnyítik az API-k használatát, tervezését
- Elősegíti a szoftver újrafelhasználhatóságát

Kollekció interfészek

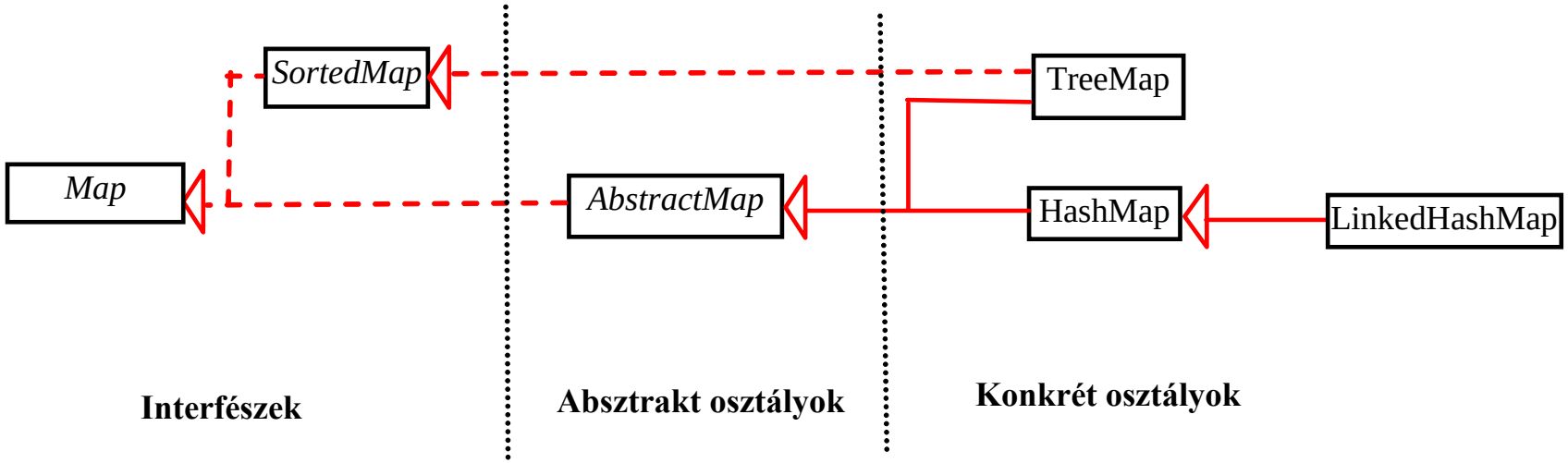


Java kollekció keretrendszer, hierarchia

A Set és List a Collection alinterfészei.



Java kollekció keretrendszer hierarchia, folyt.



Kollekció interfészek

- A kollekció interfészek generic (általános) típusú paraméterekkel dolgoznak. Például:
  - `public interface Collection<E> ...`
- Az `<E>` szintaxis azt jelenti, hogy az interfész általános (generikus) típussal működik.
- Amikor deklarálunk egy *Collection*-t, meg tudjuk határozni (ajánlott), hogy milyen típusú objektumot tartalmaz a kollekció.
- A típus paraméter a fordítóprogram számára lehetővé teszi, hogy csak az adott típusú objektumot engedje belerakni a kollekcióba, így csökkenti a futásidejű hibák számát.

Kollekció interfészek

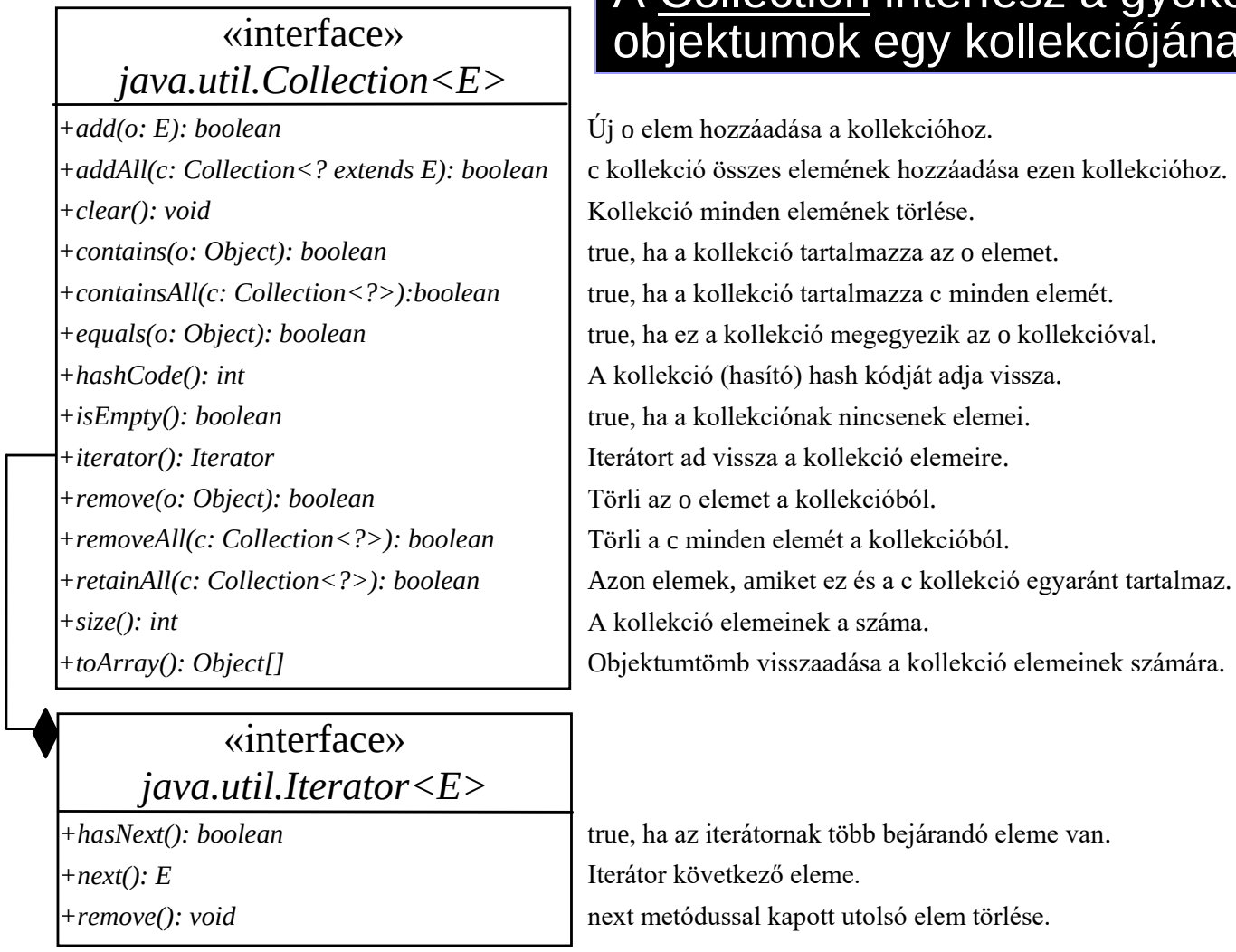
- `Collection`: akkor érdemes ezt választani, ha a lehető legnagyobb rugalmasságra van szükség.
- `Set`: matematikai halmaz modellezése
- `List`: sorszámozott kollekció, hozzáférés az elemekhez index segítségével
- `Queue`: várakozási sor létrehozására
- `Map`: az egyedi kulcsokat értékekké képezi le.
- `SortedSet`
- `SortedMap`

Általános célú implementációk

Interfaces	Implementations				
	Hash table	Resizable array	Tree	Linked list	Hash table + Linked list
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList (Vector, Stack)		LinkedList	
Queue					
Map	HashMap (HashTable)		TreeMap		LinkedHashMap

A `Collection` interfész

A Collection interfész a gyökér interfész objektumok egy kollekciójának manipulálásához.



Új o elem hozzáadása a kollekcióhoz.
c kollekció összes elemének hozzáadása ezen kollekcióhoz.
Kollekció minden elemének törlése.
true, ha a kollekció tartalmazza az o elemet.
true, ha a kollekció tartalmazza c minden elemét.
true, ha ez a kollekció megegyezik az o kollekcióval.
A kollekció (hasító) hash kódját adja vissza.
true, ha a kollekciónak nincsenek elemei.
Iterátort ad vissza a kollekció elemeire.
Törli az o elemet a kollekcióból.
Törli a c minden elemét a kollekcióból.
Azon elemek, amiket ez és a c kollekció egyaránt tartalmaz.
A kollekció elemeinek a száma.
Objektumtömb visszaadása a kollekció elemeinek számára.

true, ha az iterátornak több bejárandó eleme van.
Iterátor következő eleme.
next metódussal kapott utolsó elem törlése.

A `Collection` interfész - példa

```
import java.util.*;

public class HengerProgram {

  public static void main(String[] args) {
    Collection<Henger> hengerek=
                          new ArrayList<Henger>();
    hengerek.add(new Henger(2,10));
    hengerek.add(new Henger(3,6));
    hengerek.add(new TomorHenger(2,10,2));
    hengerek.add(new Cso(2,10,2,1));
    System.out.println(hengerek.size());
    hengerek.clear();
    System.out.println(hengerek.size());
  }
}
```

A kollekciók bejárása

1. A „`for-each`” ciklussal

```
for (Object o : collection)
  System.out.println(o);
```

```
for(Henger henger:hengerek)
  System.out.println(henger.terfogat());
```

2. Iterátorral

Az iterátor egy olyan objektum, ami megengedi, hogy bejárjuk a kollekciót, és eltávolítsuk az elemeket a kollekcióból, ha akarjuk.

Egy kollekció bejárásához az `iterator` metódusának a meghívásával kérhetünk egy `Iterator` objektumot.

A `Set` interfész

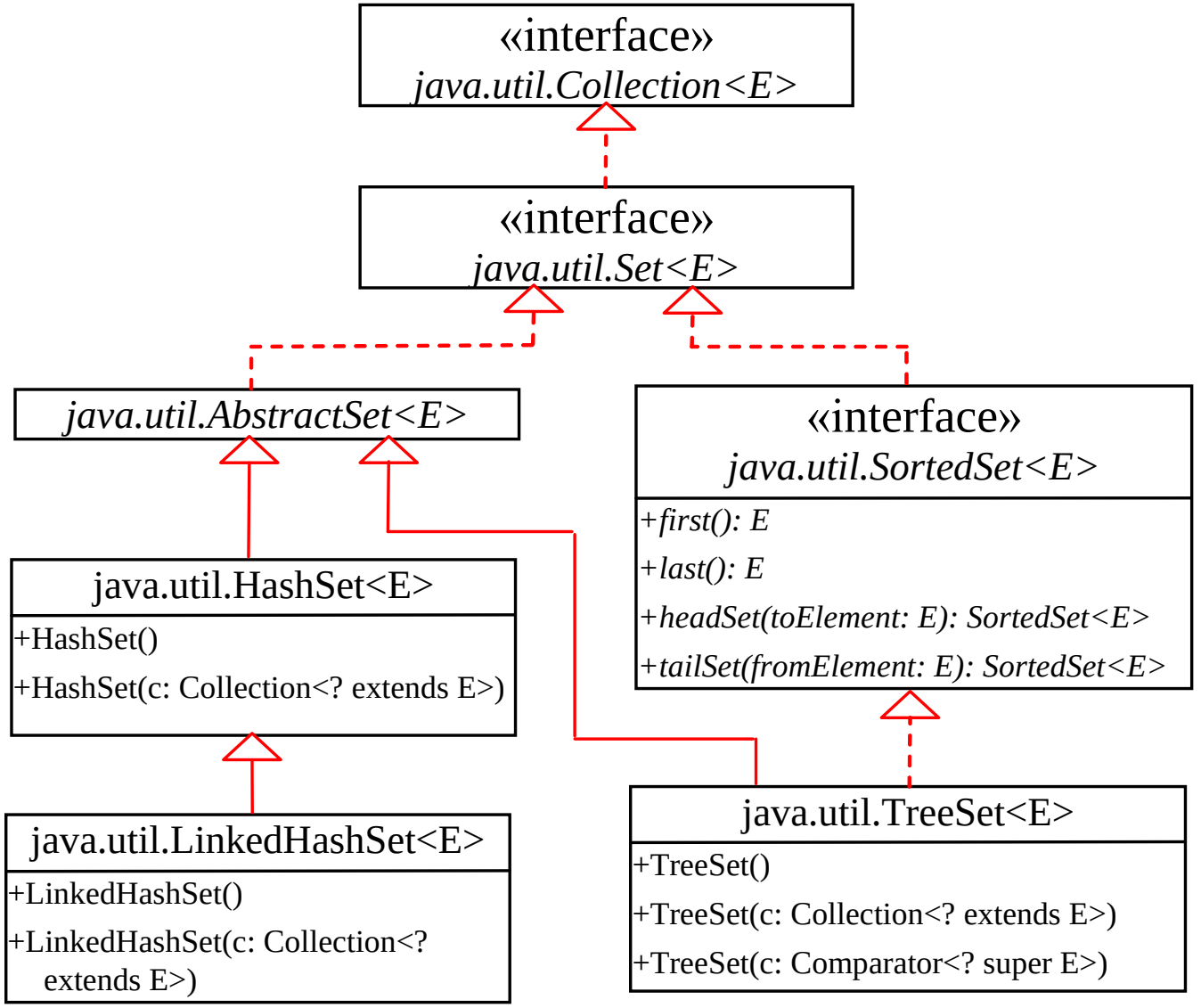
- `Set` egy speciális `Collection`, amely nem tartalmaz ismétlődő elemeket.
- A `Set` csak a `Collection`-tól örökölt metódusokat tartalmazza, azzal a megszorítással kiegészítve, hogy nem engedi meg az elemek ismétlődését.

```
public interface Set<E> extends Collection<E> {
  // Alapvető műveletek
  int size();
  boolean isEmpty();

...

}
```

A `Set` interfész



Halmaz első eleme.

Halmaz utolsó eleme.

headSet/tailSet a halmaz egy része, ami kisebb, mint toElement/nagyobb egyenlő, mint fromElement.

Fahalmaz létrehozása adott Comparatorral.

A `Set` interfész

- A Java platform három általános célú `Set` implementációt tartalmaz: a `HashSet`, `TreeSet`, és `LinkedHashSet`.
- `HashSet` esetén az elemeket egy hash táblában (hasítótáblában) tárolja, ez a legjobb választás, ha a bejárás sorrendje nem fontos.
- A `TreeSet` az elemeket egy fában tárolja az értékük szerint rendezetten. Lényegesen lassabb, mint a `HashSet`.
- A `LinkedHashSet` ötvözi a láncolt listát a hash táblával. Beszúrás szerinti rendezettség.

A `Set` interfész - példa

```
import java.util.*;

public class SetDemo {
  public static void main(String[] args) {
    Collection<Integer> szamok=new ArrayList<Integer>();
    for(int i=0;i<50;i++)
      szamok.add(Math.round((float)Math.random()*99)+1);
    for(Integer iobj:szamok)
      System.out.print(iobj+"  ");
    System.out.println();
    Set<Integer> szamokh=new HashSet<Integer>(szamok);
    for(Integer iobj:szamokh)
      System.out.print(iobj+"  ");
    System.out.println();
  }
}
```

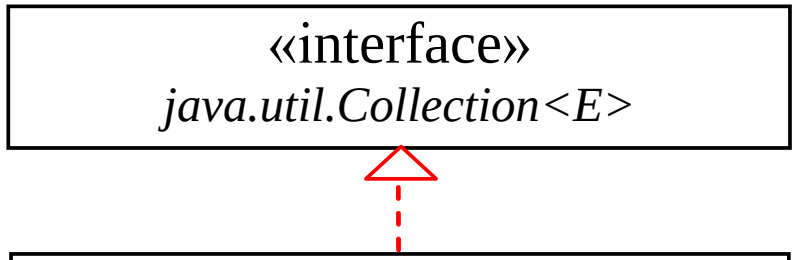
A `Set` interfész, tömeges (bulk) műveletek

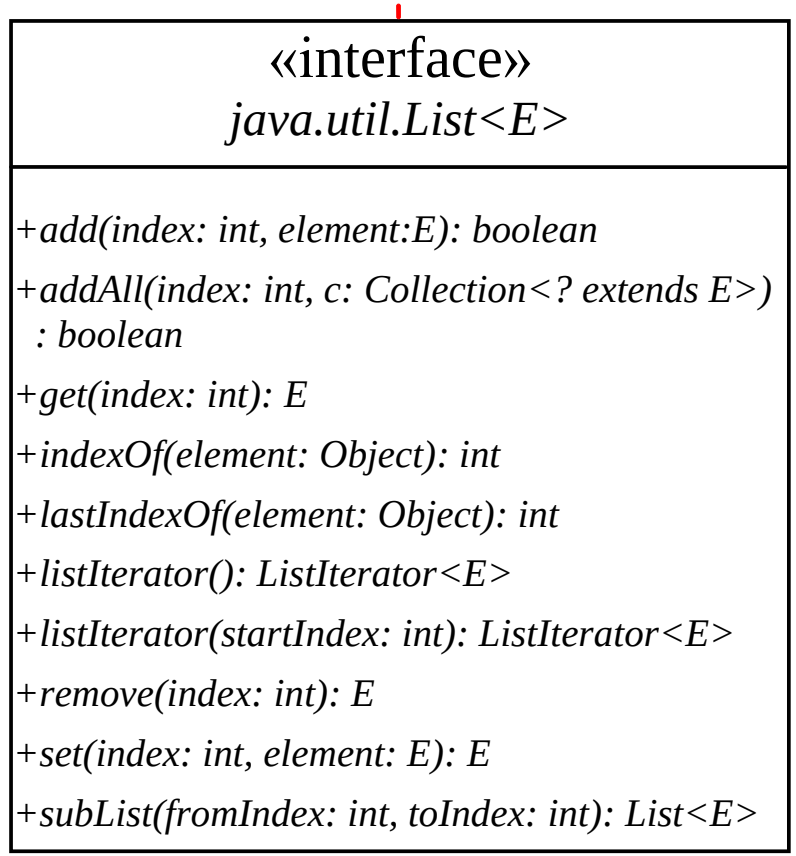
- Egyszerűen megvalósíthatók az algebrából ismert halmazműveletek.
- `s1.containsAll(s2)` – Igaz logikai értékkel tér vissza, ha `s2` részhalmaza `s1`-nek
- `s1.addAll(s2)` – Az `s1`-be `s1` és `s2` uniója kerül.
- `s1.removeAll(s2)` – Az `s1`-be `s1` és `s2` különbség kerül.
- `s1.retainAll(s2)` – Az `s1`-be `s1` és `s2` metszete kerül.
- `clear` – eltávolítja a kollekció összes elemét

A `List` interfész

- Sorszámozott kollekció
- A `Collection` osztálytól örökölt műveletek mellett újabbakat is tartalmaz:
  - Pozíció szerinti elérés: Az elemek a listában betöltött helyük alapján is elérhetők.
  - Keresés: A megadott elemet kikeresi a listából, és visszaadja a pozícióját.
  - Bejárás: Kibővíti az `Iterator` lehetőségeit
  - Részlista: Lehetővé tesz részlista műveleteket.
- A Java platform két általános célú `List` implementációt tartalmaz: a `ArrayList` (mely általában hatékonyabb), és a `LinkedList` (mely bizonyos esetekben hatékonyabb), valamint hozzáigazította a régebbi `Vector` osztályt az új interfészhez.

A `List` interfész





Új elem beszúrása adott indexnél.

c minden elemének beszúrása egy adott indexnél ezen listába.

Adott indexű listaelem visszaadása.

Első illeszkedő elem indexének visszaadása.

Utolsó illeszkedő elem indexének visszaadása.

A lista elemeinek iterátora.

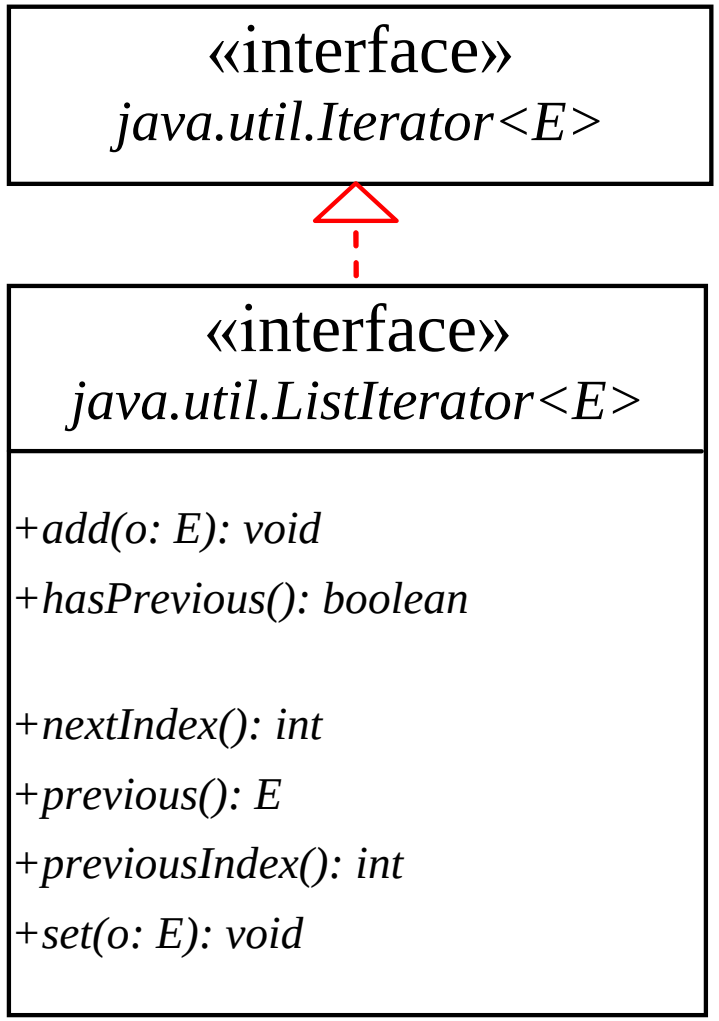
A lista elemeinek iterátora a startIndex-től.

Adott indexű listaelem törlése.

Adott indexű listaelem beállítása.

fromIndex és toIndex közötti részlista.

A lista iterátor



Megadott objektum listához adása.

true, ha a lista iterátornak van még eleme visszafele bejáráskor.

A következő elem indexe.

Az előző elem a lista iterátorban.

Az előző elem indexe.

A previous vagy next metódussal visszaadott elem cseréje az adott elemre.

A `List` interfész

- A Collection-tól örökölt metódusokat az elvárásainknak megfelelően használhatjuk.
  - `remove` metódus mindig az első előforduló elemet törli a listából.
  - Az `add` és `addAll` metódusnál az elem a lista végére kerül. Például a `list1` végére másolja a `list2` elemeit:

```
list1.addAll(list2);
```

- A listaelemek összehasonlítása azok equals metódusa alapján történik
- A pozíció szerinti elérés és keresés metódusainak működése értelemszerű

A `List` interfész - példa

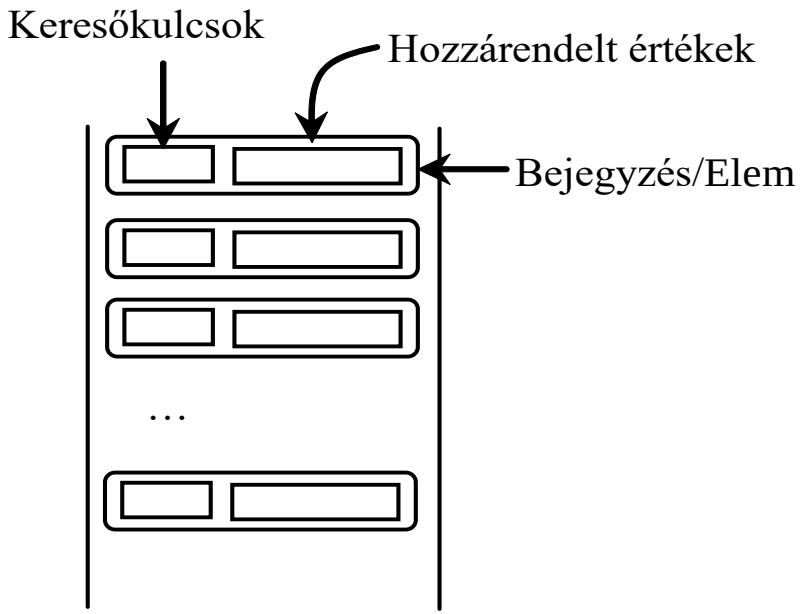
```
public class TestList {
  public static void main(String[] args) {
    // Create a list
    MyList<String> list = new MyArrayList<String>();
    // Add elements to the list
    list.add("America"); // Add it to the list
    System.out.println("(1) " + list);
    list.add(0, "Canada"); // Add it to the beginning of the list
    System.out.println("(2) " + list);
    list.add("Russia"); // Add it to the end of the list
    System.out.println("(3) " + list);
    list.add("France"); // Add it to the end of the list
    System.out.println("(4) " + list);
    list.add(2, "Germany"); // Add it to the list at index 2
    System.out.println("(5) " + list);
    list.add(5, "Norway"); // Add it to the list at index 5
    System.out.println("(6) " + list);
    // Remove elements from the list
    list.remove("Canada"); // Same as list.remove(0) in this case
    System.out.println("(7) " + list);
    list.remove(2); // Remove the element at index 2
    System.out.println("(8) " + list);
    list.remove(list.size() - 1); // Remove the last element
    System.out.println("(9) " + list);
  }
}
```

Lista algoritmusok

- A `Collections` osztály nagyon hatékonyan használható algoritmusokat nyújt listák kezelésére.
- A következő lista csak felsorolja a fontosabbakat:
  - `sort`: Rendezi a listát.
  - `shuffle`: Véletlenszerűen felcserél elemeket a listában. (Permutál.)
  - `reverse`: Megfordítja az elemek sorrendjét a listában.
  - `rotate`: Egy adott távolsággal rotálja az elemeket.
  - `swap`: Felcserél két meghatározott pozícióban levő elemet.
  - `replaceAll`: Az összes előforduló elemet kicseréli egy másikra.
  - `fill`: Felülírja az összes elemet egy meghatározott értékkel.
  - `copy`: Átmásolja a forráslistát egy céllistába.
  - `binarySearch`: Egy elemet keres a bináris keresési algoritmust használva.
  - `indexOfSubList`: Visszatér az első olyan indexszel, amelynél kezdődő részlista egyenlő a másik listával.
  - `lastIndexOfSubList`: Visszatér az utolsó olyan indexszel, amelynél kezdődő részlista egyenlő a másik listával.

A `Map` interfész

- A Map interfész kulcsokat kapcsol elemekhez. A kulcsok megfelelnek az indexnek.
- A listaindexek egészek, a Map kulcsok tetszőleges objektumok lehetnek.



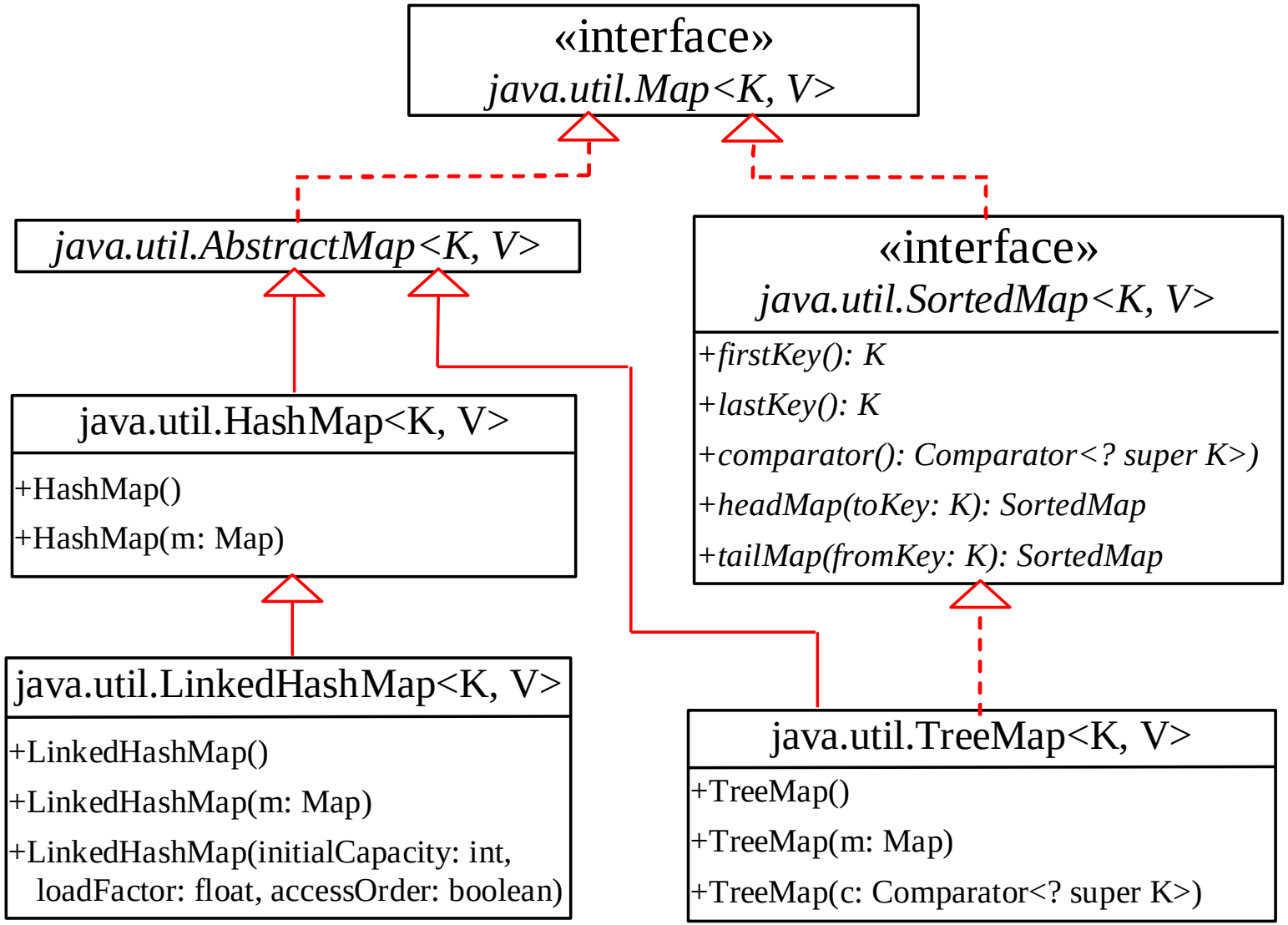
A `Map` interfész

java.util.Map<K, V>	
+clear(): void	Minden hozzárendelés törlése.
+containsKey(key: Object): boolean	true, ha ez a tábla tartalmaz értéket (bejegyzést) a megadott kulcshoz.
+containsValue(value: Object): boolean	true, ha ez a tábla tartalmaz kulcsot a megadott értékhez.
+entrySet(): Set	A tábla bejegyzéseinek halmaza.
+get(key: Object): V	A megadott kulcshoz tartozó bejegyzés.
+isEmpty(): boolean	true, ha a tábla nem tartalmaz bejegyzéseket.
+keySet(): Set<K>	A tábla kulcsainak halmaza.
+put(key: K, value: V): V	Bejegyzés felvétele a táblába.
+putAll(m: Map): void	m minden hozzárendelésének felvétele ebbe a táblába.
+remove(key: Object): V	Megadott kulcshoz tartozó bejegyzés törlése.
+size(): int	A tábla bejegyzéseinek száma.
+values(): Collection<V>	A tábla kulcsokhoz tartozó értékeinek kollekciója.

A `Map` interfész

- Kulcs-érték párokat tartalmaz
- Az egyedi kulcs alapján megkereshetjük a kulcshoz tartozó értéket (sokkal kényelmesebb, mintha össze kellene állítanunk egy keresőobjektumot)
- A Java platform három általános célú `Map` implementációt tartalmaz: `HashMap`, `TreeMap` és `LinkedHashMap` (melyek hasonló tárolási elvűek és működésűek, mint a Set implementációk), valamint hozzáigazította a régebbi `Hashtable` osztályt az új interfészhez.

A `Map` interfész hierarchia



HashMap és TreeMap

- A Map interfészt implementáló konkrét osztályok:
  - HashMap,
  - TreeMap.
- A HashMap hatékony egy érték megkeresésekor, egy bejegyzés beszúrásakor vagy törlésekor.
- A TreeMap, ami a SortedMap-et implementálja, hatékony a kulcsok egy rendezés szerinti bejárásakor.

LinkedHashMap

- A LinkedHashMap kiterjeszti a HashMap osztályt egy láncolt lista implementációval, ami támogatja az elemek rendezését.
- A HashMap bejegyzések nem rendezettek, de egy LinkedHashMap bejegyzései lekérhetők:
  - Beszúrási sorrendben – abban a sorrendben, ahogyan beszúrásra kerültek,
  - Elérési sorrendben – abban a sorrendben, ahogyan az elérésük utoljára történt, kezdve a legutoljára elért elemmel.
- Alapértelmezés: a no-arg konstruktor beszúrási sorrendben hozza létre a LinkedHashMap objektumot.
- Egy LinkedHashMap elérési sorrendű példányának létrehozásához a LinkedHashMap(initialCapacity, loadFactor, true) kódot használjuk.

A `Map` interfész - példa

```
import java.util.*;

public class Freq {
 public static void main(String[] args) {
  String[] words={"i","came","i","saw","i","left"};
  Map<String, Integer> m=new HashMap<String, Integer>();

  for (String w: words) {
   Integer freq=m.get(w);
   m.put(w, (freq==null)?1:freq+1);
  }

  System.out.println(m.size() + " distinct words:");
  System.out.println(m);
 }
}
```

A `Map` interfész

- A `Collection view` metódusokkal háromféle nézetét kapjuk a Map-nek
  - `keySet`: kulcsok halmaza (`Set`)
  - `values`: értékek kollekciója (`Collection`)
  - `entrySet`: kulcs-érték párok halmaza (`Set`). A `Map` interfész tartalmaz egy kis beágyazott interfészt – `Map.Entry` – ezen halmaz elemei típusának meghatározására.
- Csak a kollekció nézetekkel lehet iterációt végezni a Map-en.
- Használhatók `remove, removeAll, retainAll, clear` kollekció műveletek

Rendezés

- `Collections.sort(l);`
  Alapértelmezett rendezés, a lista elemeinek osztálya implementálja `Comparable` interfészt, azaz tartalmaz egy `compareTo` metódust.
  - Korlátok:
    - Csak egyféle rendezettség
    - Be van égetve az objektumba, nem lehet változtatni rajta
- Rendezés egy Comparator objektummal

```
public interface Comparator<T> {
  int compare(T o1, T o2);
}
```

A `SortedSet` interfész

```
public interface SortedSet<E> extends Set<E> {
    // Részlista
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);
    // Végpontok
    E first();
    E last();
    // Comparator elérés
    Comparator<? super E> comparator();
}
```

Rendezettség:

- Az elemek természetes rendezettsége szerint
- A létrehozásakor megadott Comparator alapján

Általános implementáció: TreeSet

A `SortedMap` interfész

```
public interface SortedMap<K,V> extends Map<K,V>{
    Comparator<? super K> comparator();
    SortedMap<K, V> subMap(K fromKey, K toKey);
    SortedMap<K, V> headMap(K toKey);
    SortedMap<K, V> tailMap(K fromKey);
    K firstKey();
    K lastKey();
}
```

Rendezettség:

- Az kulcsok természetes rendezettsége szerint
- A létrehozásakor megadott Comparator alapján

Általános implementáció: TreeSet

Általános célú implementációk

Interfaces	Implementations				
	Hash table	Resizable array	Tree	Linked list	Hash table + Linked list
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList (Vector, Stack)		LinkedList	
Queue					
Map	HashMap (HashTable)		TreeMap		LinkedHashMap

- Az általános célú implementációk mind biztosítják az összes opcionális műveletet, amit az interfészek tartalmaznak.

Interfészek - összegzés

- Általános szabály, hogy programíráskor az interfészeken, és nem az implementációkon kell gondolkodni.
- Legtöbb esetben az implementáció megválasztása csak a teljesítményt befolyásolja.
- Az előnyben részesített programozói stílus az, ha egy interfész típusú változóhoz választunk egy implementációt.
- Így a program nem függ majd egy adott implementáció esetén az ahhoz hozzáadott új metódusoktól, ezáltal a programozó bármikor szabadon változtathatja az implementációt, mikor jobb teljesítményt szeretne elérni, vagy a működési részleteket szeretné módosítani.

Magasszintű programozási nyelvek 2

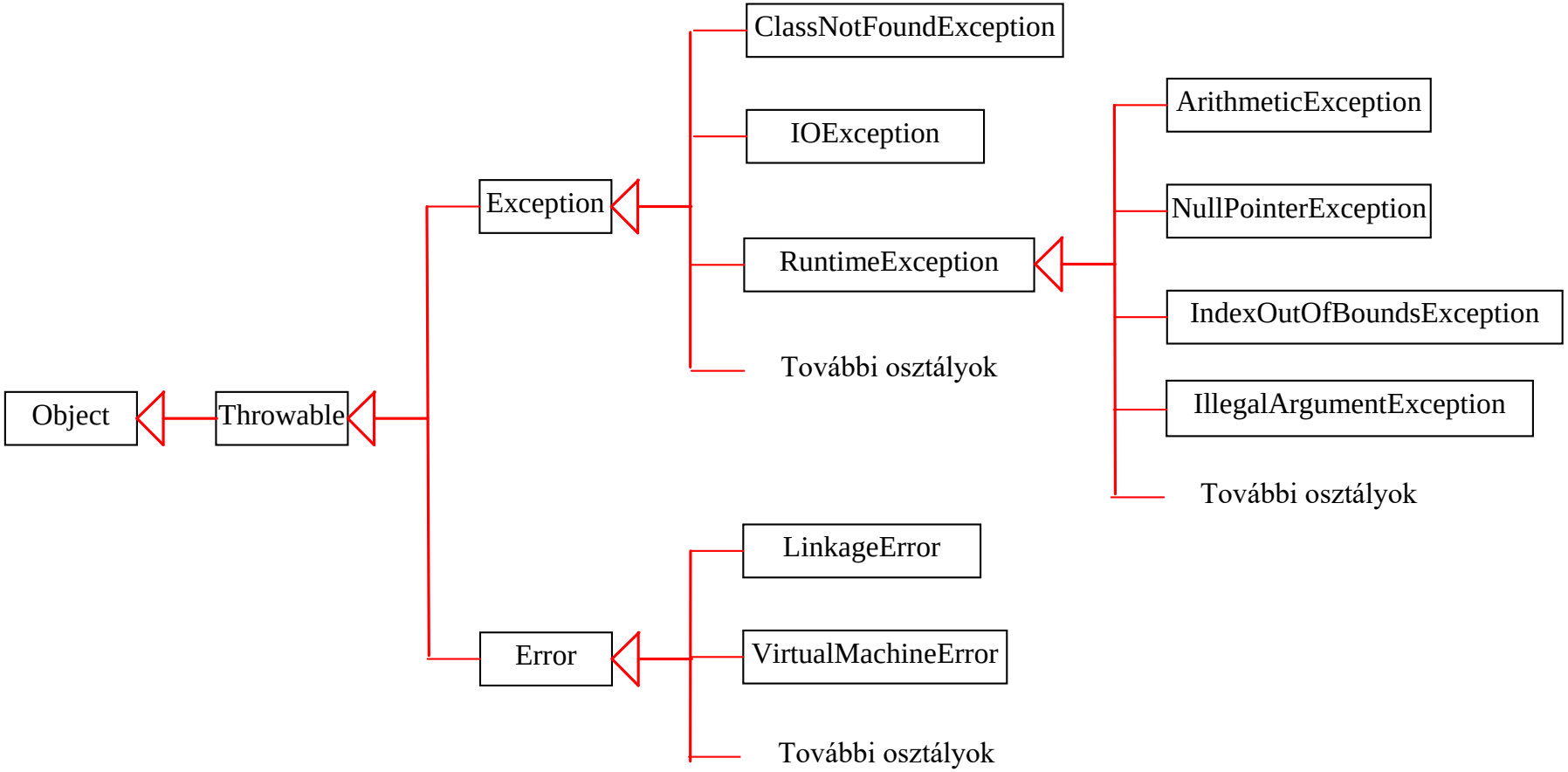
Kivételkezelés

Dr. Tiba Attila

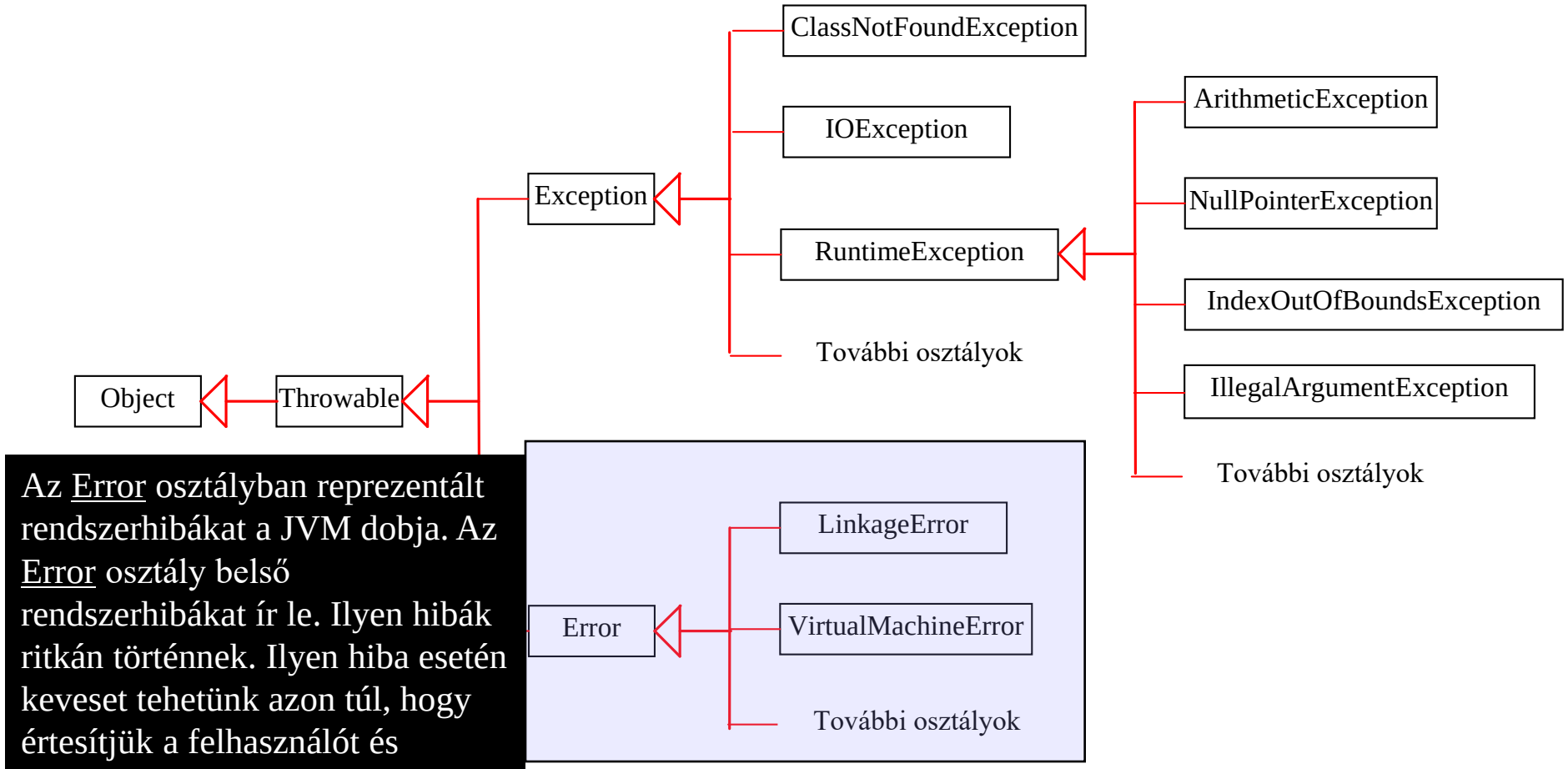
Motiváció

Ha egy programban futás közben hiba történik, abnormálisan megszakítja a működését. Hogyan tudjuk kezelni ezekez a hibákat, hogy a program továbbhaladjon és szabályosan fejeződjön be? A kivételkezelés erre a problémára adja meg a választ.

Kivételtípusok



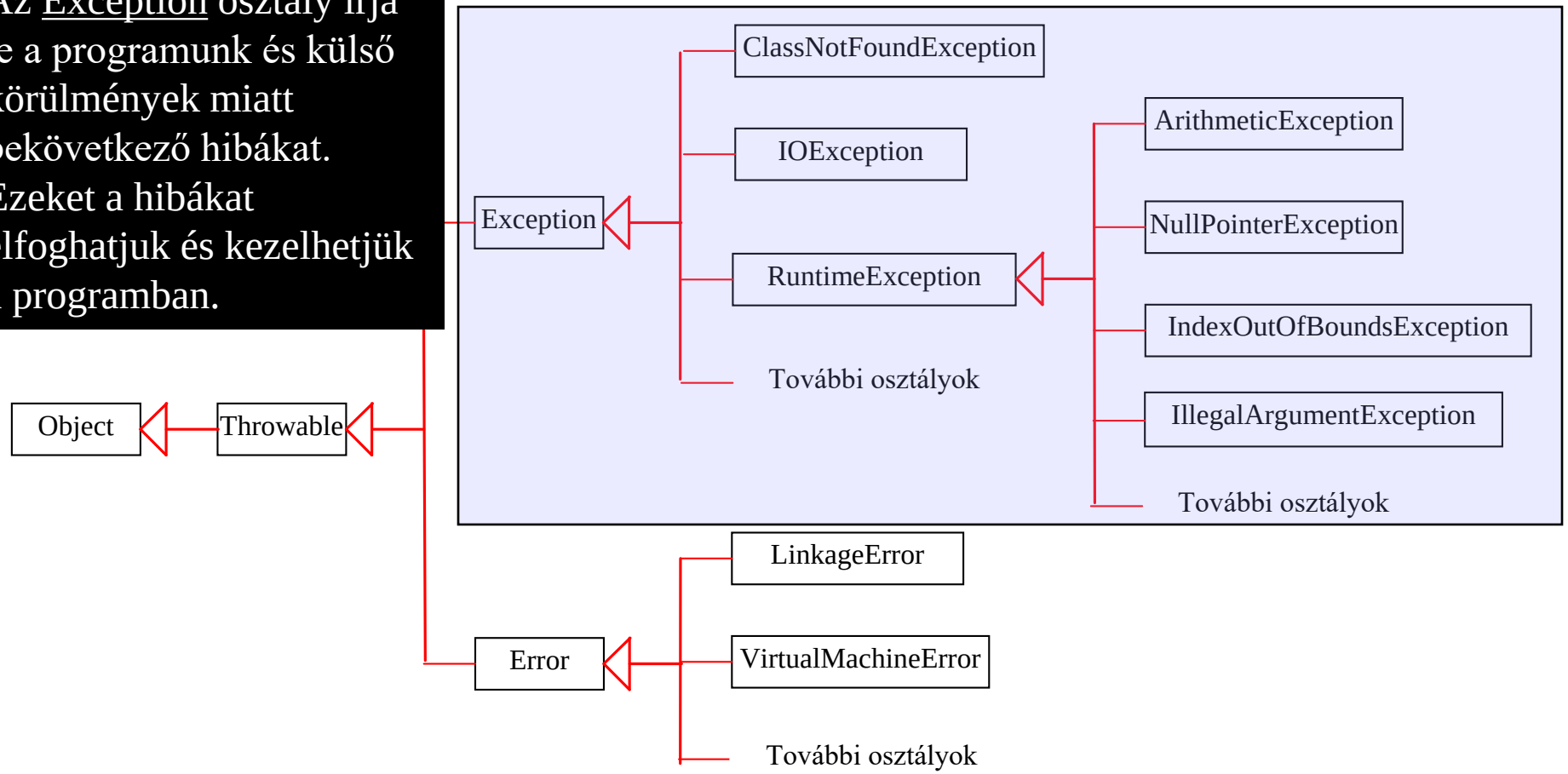
Rendszerhibák



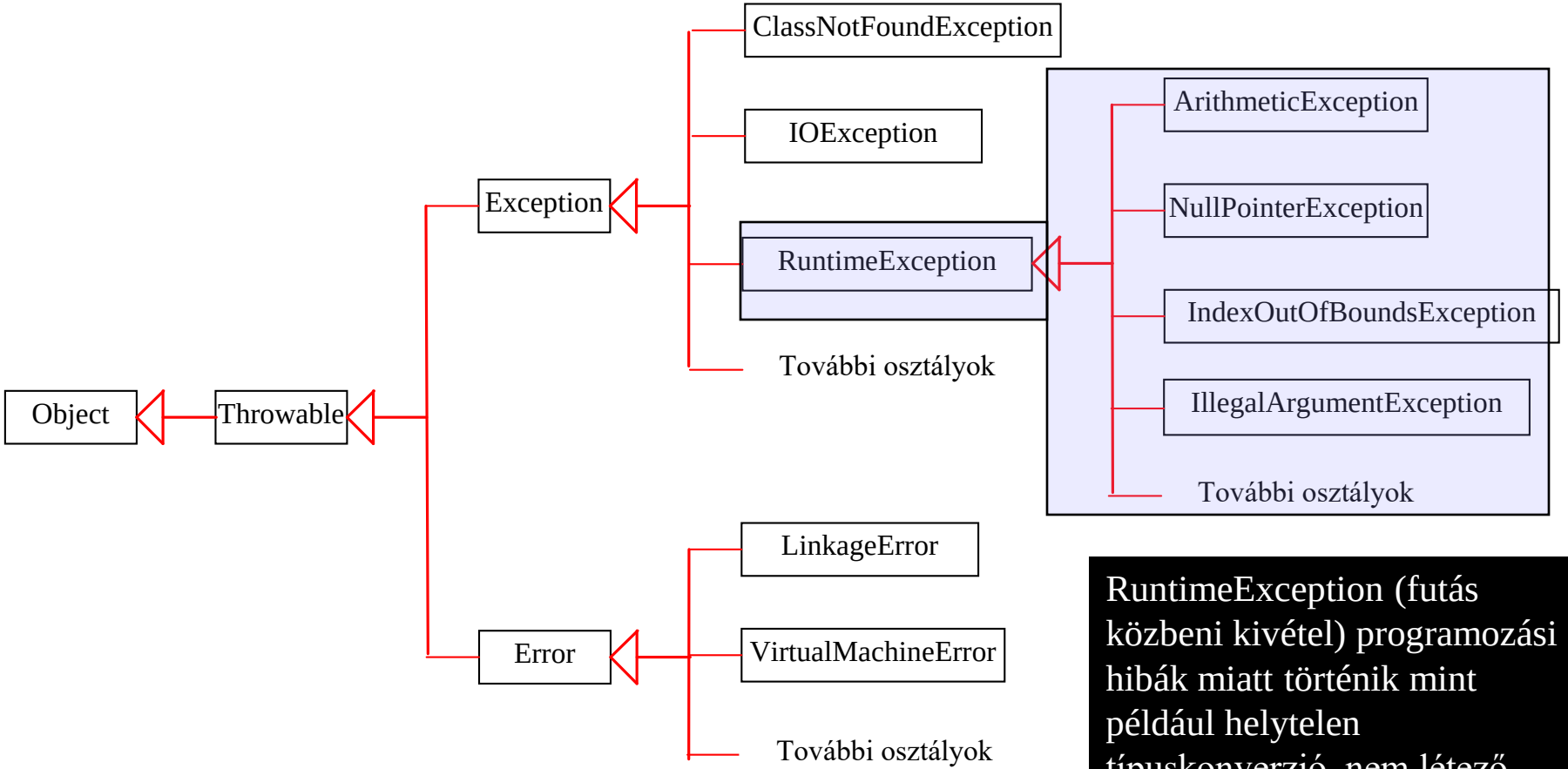
Az Error osztályban reprezentált rendszerhibákat a JVM dobja. Az Error osztály belső rendszerhibákat ír le. Ilyen hibák ritkán történnek. Ilyen hiba esetén keveset tehetünk azon túl, hogy értesítjük a felhasználót és próbáljuk a programot a legszabályosabban befejezni.

Kivételek

Az Exception osztály írja le a programunk és külső körülmények miatt bekövetkező hibákat. Ezeket a hibákat elfoghatjuk és kezelhetjük a programban.



Futás közbeni kivételek



RuntimeException (futás közbeni kivétel) programozási hibák miatt történik mint például helytelen típuskonverzió, nem létező indexű tömbelem elérése vagy numerikus hibák.

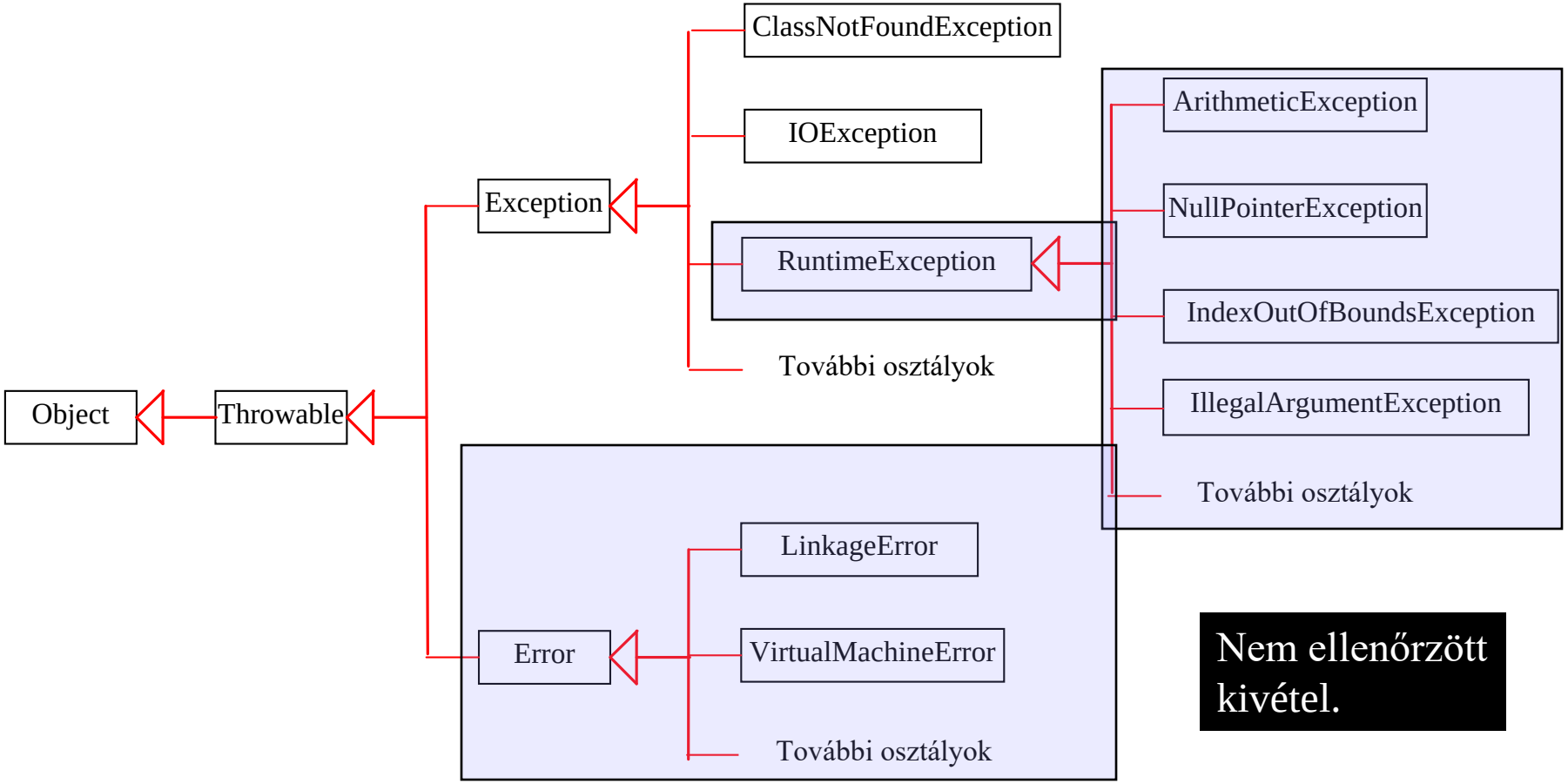
Ellenőrzött kivételek vs. Nem ellenőrzött kivételek

A RuntimeException, Error és alosztályaik *nem ellenőrzött kivételekként* ismertek. Minden más kivétel *ellenőrzött kivétel*, azaz esetükben a fordító kényszeríti a programozót ezek ellenőrzésére és kezelésére.

Nem ellenőrzött kivételek

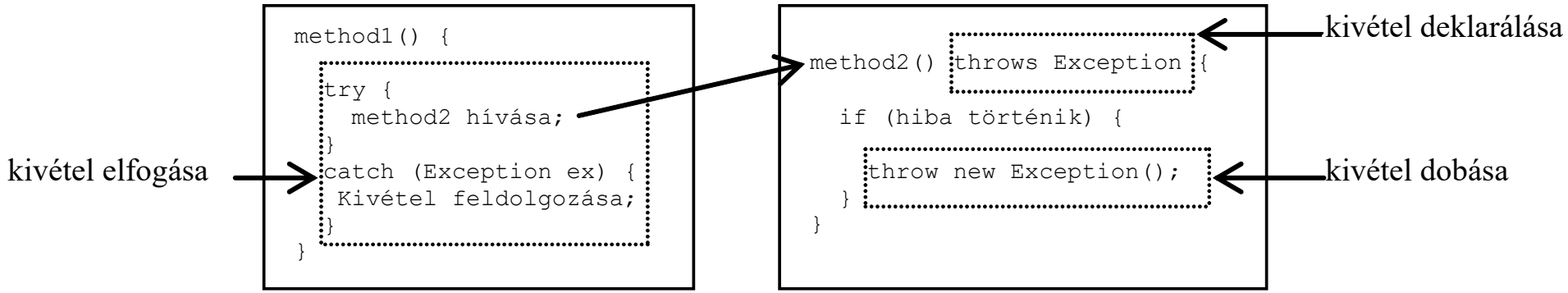
A legtöbb esetben a nem ellenőrzött kivételek logikai programhibákat jeleznek, amik automatikusan nem javíthatók, Például NullPointerException kivétel történik, ha egy objektumot referenciaváltozón keresztül érünk el anélkül, hogy a változóhoz már objektumot rendeltünk volna; IndexOutOfBoundsException történik, ha a tömb méreténél nagyobb indexű elemet próbálunk elérni. Ezeket a logikai hibákat a programban kell korrigálnunk. Nem ellenőrzött kivételek bárhol bekövetkezhetnek a programban. A Java nem kötelez minket arra, hogy kódot írjunk ezen kivételek ellenőrzésére.

Nem ellenőrzött kivételek



Nem ellenőrzött kivétel.

Kivételek deklarálása, dobása, és elkapása



Kivételek deklarálása

Minden metódusnak meg kell adnia azokat az ellenőrzött kivételeket, amiket dobhat. Ez az elvárás *kivételek deklarálásaként* ismert.

```
public void myMethod()
   throws IOException

public void myMethod()
   throws IOException, OtherException
```

Kivételek dobása

Ha a program hibát érzékel, létrehozhatja egy kivétel osztály egy példányát és azt visszaadhatja (dobhatja). Ez a művelet *kivétel dobásaként* ismert. Példa:

```
throw new TheException();

TheException ex = new TheException();
throw ex;
```

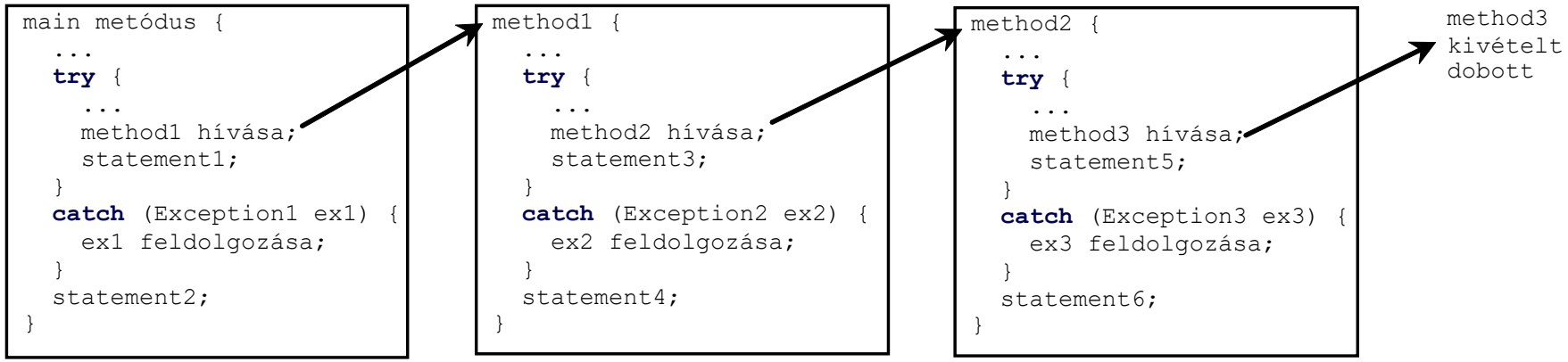
Kivételek dobása - példa

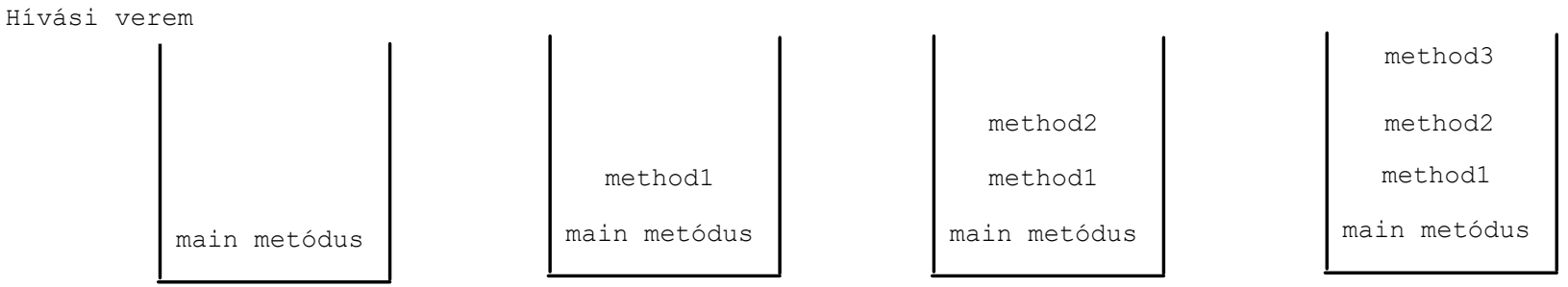
```
/** Új sugárérték beállítása */
public void setRadius(double newRadius)
    throws IllegalArgumentException {
  if (newRadius >= 0)
    radius =  newRadius;
  else
    throw new IllegalArgumentException(
      "A sugar nem lehet negativ");
}
```

Kivételek elkapása

```
try {
  utasítások;  // Kivételeket dobható utasítások
}
catch (Exception1 exVar1) {
  exception1 kezelése;
}
catch (Exception2 exVar2) {
  exception2 kezelése;
}
...
catch (ExceptionN exVar3) {
  exceptionN kezelése;
}
```

Kivételek elkapása





Ellenőrzött kivételek deklarálása és elkapása

A Java kötelez minket rá, hogy az ellenőrzött kivételekkel foglalkozzunk. Ha egy metódus ellenőrzött kivételt deklarál (azaz nem Error vagy RuntimeException), meg kell azt hívnunk egy try-catch blokkban, vagy deklarálnunk, hogy a kivételt a hívó metódusban dobja. Például a p1 metódus hívja a p2 metódust és a p2 dobhasson ellenőrzött kivételt (pl. IOException-t); ekkor az alábbi (a) vagy (b) kódot kell használnunk.

```
void p1() {
  try {
    p2();
  }
  catch (IOException ex) {
    ...
  }
}
```

(a)

```
void p1() throws IOException {

  p2();

}
```

(b)

A `finally` használata

```
try {
   utasítások;
}
catch(TheException ex) {
   ex kezelése;
}
finally {
   záró utasítások;
}
```

Végrehajtás követése

Tegyük fel, hogy nincsenek kivételek az utasítások során.

```
try {
  utasítások;
}
catch(TheException ex) {
  ex kezelése;
}
finally {
  záró utasítások;
}

következő utasítás;
```

Végrehajtás követése

```
try {
   utasítások;
}
catch(TheException ex) {
   ex kezelése;
}
finally {
   záró utasítások;
}

következő utasítás;
```

A végső (final) blokk mindig végrehajtásra kerül.

Végrehajtás követése

```
try {
  utasítások;
}
catch(TheException ex) {
  ex kezelése;
}
finally {
  záró utasítások;
}

következő utasítás;
```

A metódus következő utasítása végrehajtásra kerül.

Végrehajtás követése

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  ex kezelése;
}
finally {
  záró utasítások;
}

következő utasítás;
```

Tegyük fel, hogy a statement2 utasítás Exception1 típusú kivételt dob.

Végrehajtás követése

```
try {
   statement1;
   statement2;
   statement3;
}
catch(Exception1 ex) {
   ex kezelése;
}
finally {
   záró utasítások;
}

következő utasítás;
```

A kivétel lekezelésre kerül.

Végrehajtás követése

```
try {
   statement1;
   statement2;
   statement3;
}
catch(Exception1 ex) {
   ex kezelése;
}
finally {
   záró utasítások;
}

következő utasítás;
```

A végső (final) blokk mindig végrehajtásra kerül.

Végrehajtás követése

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  ex kezelése;
}
finally {
  záró utasítások;
}

következő utasítás;
```

A metódus következő utasítása végrehajtásra kerül.

Végrehajtás követése

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  ex kezelése;
}
catch(Exception2 ex) {
  ex kezelése;
  throw ex;
}
finally {
  záró utasítások;
}

következő utasítás;
```

Tegyük fel, hogy a statement2 utasítás Exception2 típusú kivételt dob.

Végrehajtás követése

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  ex kezelése;
}
catch(Exception2 ex) {
  ex kezelése;
  throw ex;
}
finally {
  záró utasítások;
}

következő utasítás;
```

A kivétel lekezelésre kerül.

Végrehajtás követése

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  ex kezelése;
}
catch(Exception2 ex) {
  ex kezelése;
  throw ex;
}
finally {
  záró utasítások;
}

következő utasítás;
```

A végső blokk végrehajtása.

Végrehajtás követése

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  ex kezelése;
}
catch(Exception2 ex) {
  ex kezelése;
  throw ex;
}
finally {
  záró utasítások;
}

következő utasítás;
```

A kivétel újbóli dobása, és a vezérlés visszaadása a hívónak.

Figyelmeztetés: kivételek

- A kivételkezelés megkülönbözteti a hibakezelő kódot a normál programozási feladatoktól, így a programok könnyebben olvashatóbbá és módosíthatókká válnak. Ügyeljünk azonban arra, hogy a kivételkezelés meglehetősen erőforrásigényes a kivételobjektumok példányosítása, a hívási verem visszafejtése és a hibának a hívó metódushoz való visszajuttatása (propagálása) miatt.

Mikor dobjunk kivételt?

- Kivétel metódusban történik. Ha a kivételt a hívóval akarjuk feldolgoztatni, akkor készítenünk kell egy kivételobjektumot és dobni azt. Ha a kivételt le tudjuk kezelni a metódusban, ahol bekövetkezik, akkor nem kell dobnunk azt.

Mikor használjunk kivételt?

Mikor használjunk try-catch blokkot a kódban? Akkor, ha nem várt hibákat szeretnénk kezelni. Ne használjuk egyszerű, elvárt helyzetekhez, csak váratlan körülményekhez. Például az alábbi kód helyett (folyt.):

```
try {
  System.out.println(refVar.toString());
}
catch (NullPointerException ex) {
  System.out.println("refVar is null");
}
```

Mikor használjunk kivételt? (folyt.)

Használjuk ezt:

```
if (refVar != null)
  System.out.println(refVar.toString());
else
  System.out.println("refVar is null");
```

Saját kivételosztályok definiálása

- Használjuk az API kivételosztályait, amikor csak lehetséges.
- Csak akkor használjunk saját kivételosztályokat, ha az előre definiált osztályok nem elégségesek.
- A saját kivételosztályokat az Exception (vagy annak alosztályából) örökítsük.

Állítások (assertion)

Az állítás egy Java utasítás, ami lehetővé teszi, hogy feltételt fogalmazzunk meg a programra. Ez a feltétel egy logikai kifejezést tartalmaz, aminek a program futása során igaznak kell maradnia.

Az ilyen állítások segítenek a program helyességének biztosításában és a logikai hibák elkerülésében.

Állítások deklarálása

Egy állítás a JDK 1.4-től elérhető assert kulcsszavával definiálható:

assert *állítás*; vagy
assert *állítás*: *detailMessage*;

ahol az *állítás* egy logikai (Boolean) kifejezés, a *detailMessage* pedig egy egyszerű vagy objektum típusú érték.

Állítások végrehajtása

Egy állítás utasítás végrehajtásakor a Java kiértékeli az állítást. Ha az hamis, egy AssertionError kivételt dob. Az AssertionError osztálynak van egy no-arg (argumentum nélküli) konstruktora, továbbá hét túlterhelt egyargumentumú konstruktúra következő típusokkal: int, long, float, double, boolean, char, és Object.

Állítások végrehajtása - példa

```
public class AssertionDemo {
  public static void main(String[] args) {
    int i; int sum = 0;
    for (i = 0; i < 10; i++) {
      sum += i;
    }
    assert i == 10;
    assert sum > 10 && sum < 5 * 10 : "sum is " + sum;
  }
}
```

Az első, üzenetet nem tartalmazó assert utasításnál, az AssertionError osztály no-arg konstruktora lesz alkalmazva. Az üzenetet tartalmazó második assert utasításnál a megfelelő AssertionError konstruktor (int argumentum) kerül felhasználásra az üzenetben szereplő típushossz illeszkedve. Mivel az AssertionError az Error alosztály, ezért az állítás hamisra válásakor a program a konzolra küld hibaüzenetet, majd kilép.