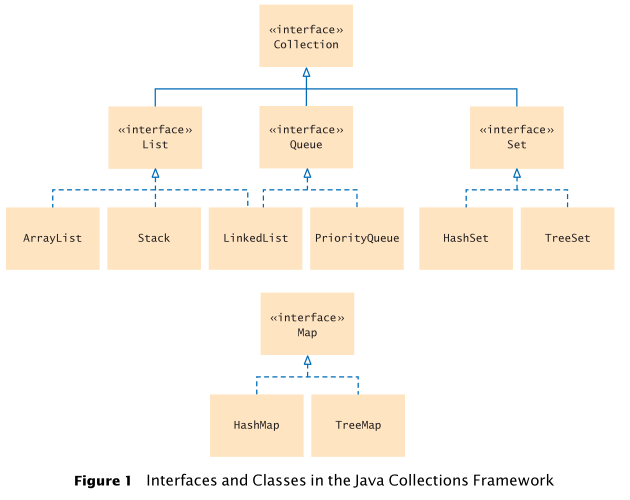
# 16. Java Collections Framework

Ha több dolgot együtt szeretnénk kezelni egy programban, akkor azokat egy gyűjteménybe, más szóval kollekcióba (collection) tesszük. Például az ArrayList osztály egy kollekció, amelyet a Java biztosít számunkra.

A Java ezen kívül még sok más interface-t és osztályt is biztosít a különböző kollekciókhoz. Ezeket együttesen Java Collections Framework-nek nevezzük.

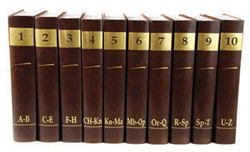
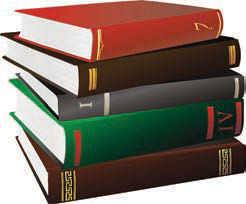
Az alábbi ábra mutatja a JCF főbb elemeit:  
(folytonos vonal: öröklődés, szaggatott vonal implementáció)



Az egyik ág a Collection interface-ből indul. Az ebben lévő absztrakt metódusokat a lejjebb lévő osztályok implementálják. Így elérhető, hogy ugyanolyan metódusokat lehessen használni a különböző osztályokban.

Néhány példa közös metódusokra: size(), add(), remove(), contains().

A kollekciók három fő csoportját a következő interface-ek írják le:

* List (lista): olyan kollekció, amelyben fontos az elemek sorrendje.  
  Ezt valósítja meg a már megismert ArrayList osztály. Egy ArrayList valójában egy tömb, amelynek méretét szükség szerint változtatja a program.   
    
  Ha a lista belsejébe kell beszúrni egy elemet, vagy törölni onnan egyet, akkor az ArrayList nem túl hatékony, mert a lista egy részét léptetni kell a tömbben. Ezért vezették be a LinkedList osztályt, amely ezeket a műveleteket hatékonyan tudja elvégezni.  
  A Stack osztály olyan listát definiál, amelynek csak a tetejére lehet új elemet tenni, és csak onnan lehet elvenni is.  
  
* A Queue (sor) olyan lista, amelynek cak a végéhez lehet hozzáadni új elemet, és az elejéről lehet elvenni. Olyan, mint egy sor a pénztárnál.  
  A Priority Queue ezzel szemben nem a sorrendet tárolja, hanem az egyes elemek prioritását (fontosságát), és nagyon hatékonyan tudja eltávolítani a legnagyobb prioritású elemet.  
  Olyan, mint a amikor a váróból a következő beteg megy be az orvoshoz.
* A Set (halmaz) nagyon hatékonyan tud hozzáadni, elvenni és megtalálni elemeket a kollekcióban, és nem engedi, hogy egy elem kétszer szerepeljen.

A Map (térkép) kulcsok és értékek közötti kapcsolatokat tárol. Bármelyik kulcshoz nagyon gyorsan meg tudja adni a hozzárendelt értéket. Ha fontos az elemek sorrendje, akkor a TreMap, egyébként a HashMap osztályt használjuk. Erre is láttunk már példákat.



## Csatolt listák

Az új adatszerkezetek közül először a csatolt listával ismerkedünk meg. (Láncolt listának is szokták hívni.) Erre azért van szükség, mert az ArrayList szerkezet bizonyos esetekben nem elég hatékony.

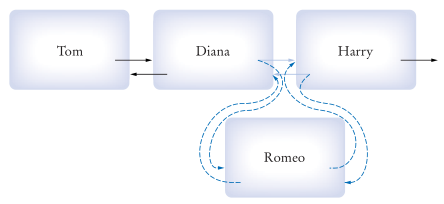
Képzelj el egy listát, amelyben egy csoport tagjai vannak. Amikor valaki kilép a csoportból, törölni kell a listából. Mivel az ArrayList valójában egy tömb, a törölt elem utániakat eggyel előrébb kell vinni a listában. nagy listák esetén ez egy időigényes művelet. Hasonló a helyzet egy új tag beszúrásakar: az utána következő elemeket eggyel hátrább kell léptetni.

A csatolt listák kiküszöbölik ezeket a léptetéseket.

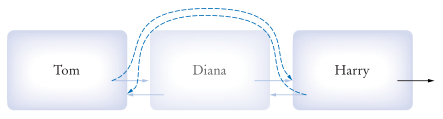
A csatolt listák csomópontokból (nodes) állnak. Egy csomópont egy elemet tárol és hivatkozásokat a szomszédos elemekre.



Amikor egy új csomópontot szúrunk be a listába, akkor csak a szomszédok hivatkozásait kell módosítani:



Egy csomópont eltávolításánál hasonló a helyzet:



Van azonban hátránya is a csatolt listáknak: sokáig tart egy adott elemre állni. Ha például a lista ötödik elemét szeretnénk kiolvasni, akkor az elejétől egyesével kell lépkedni az elemeken az ötödikig.

## A LinkedList osztály

A JCF-ben a LinkedList osztály segítségével készíthetünk csatolt listákat. Ez is egy generikus osztály, vagyis meg kell adnunk a lista elemeinek típusát. Például:

LinkedList<String> nevek = new LinkedList<>();

A JCF közös metódusain kívül van néhány metódus, amely a lista elejét és végét kezeli:

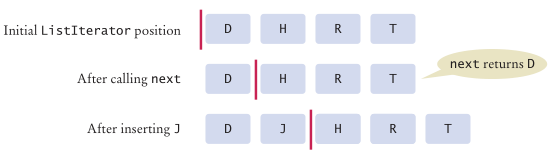
|  |  |
| --- | --- |
| nevek.addFirst("Sally"); | új elem hozzáadása a lista elejéhez |
| nevek.addLast("Harry"); | új elem hozzáadása a lista végéhez |
| nevek.getFirst(); | megadja a lista első elemét |
| nevek.getLast(); | megadja a lista utolsó elemét |
| nevek.removeFirst(); | törli a lista első elemét |
| nevek.removeLast(); | törli a lista utolsó elemét |

A csatolt lista elemeinek eléréséhez szükség van egy iterátorra, amely jelzi, hogy hol tartunk a listában. Az iterátor két elem közé mutat, ahogyan a kurzor a szövegszerkesztőben a betűk közé mutat.

Iterátor létrehozása:

ListIterator<String> it = csl.listIterator();

Létrehozása után az iterátor a lista első eleme elé mutat. A következő pozícióba mozgatni az it.next() utasítással lehet. A next metódus a következő elemet adja vissza. Új elem beszúrása után az elem utáni helyre ugrik.



Ha a lista végén hívjuk meg a next metódust, akkor egy NoSuchElementException kivételt kapunk.

Ezt úgy lehet megelőzni, hogy a léptetés előtt a hasNext() metódussal megvizsgáljuk, hogy van-e következő elem.

A csatolt lista összes elemén a következő ciklussal lehet végigmenni:

while (it.hasNext()) {  
 String nev = it.next();  
 // nev feldolgozása  
}

Ha a lista összes elemével szeretnénk dolgozni, akkor használhatjuk a for‑each ciklust is:

for (String nev : csl) {  
 // név feldolgozása  
}

Így nem kell az iterátorral fogalkoznunk, de a háttérben a ciklus azt fogja használni.

Mivel a csomópontok az előző elem címét is tárolják, visszafelé is lehet haladni a listában a previous() metódussal. Az előző elem létezését a hasPrevious() metódussal lehet megvizsgálni.

Az iterátor helyére az add() metódussal lehet hozzáadni egy új elemet, a legutóbbi next() vagy previous() metódus által visszaadott elemet pedig a remove() metódussal lehet törölni.

A remove() metódust a léptetés után csak egyszer lehet meghívni, és nem lehet közvetlenül egy elem hozáadása után!

## Mintaprogram

Kezdj egy új projektet a NetBeans-ben *csatoltlista* néven!

Írd be és róbáld ki az alábbi programot! A sorok végén lévő megjegyzések az elemeket és az iterátor helyét mutatják.

