



Generikusok, Collections Framework

Java PROGRAMOZÁS

6. GYAKORLAT



Amiről ma szó lesz

- Generikusok
- A `java.util` csomag:
- Collections Framework
 - Collection interface
 - Map interface
- Collection implementációk
- Map implementációk

Generikusok <T>

- Programozás „beégetett” típusok nélkül: „Programming with concepts”.

- Mi a gond az alábbi kódrészlettel?

```
List myIntList = new LinkedList();           //  
1  
myIntList.add(new Integer(0));               //  
2  
Integer x = (Integer) myIntList.iterator().next();  
// 3
```

- Amikor kivesszük a listából, explicit típuskonverzióra van szükség, hogy visszacapjuk a berakott típust.
- Ebből következően a 3. sor veszélyt rejt magában, mert futásidejű típuskonverziót kell végezni. A fordító ebből adódóan nem tudja ellenőrizni a konvertálás helyességét.
- Célunk, hogy formális típusokkal tudjunk dolgozni, ugyanakkor fordítási időben ellenőrizhessük a helyes típushasználatot.



Generikusok

- Szintaxis hasonlít a C++-ban megszokotthoz.
 - Nem kell szóközt hagyni a > között: `List<List<Integer>>` (ahogyan C++11-ben sem)
 - Amikor egy formális típussal megírt kódot használunk, a Java fordító ellenőrzi, hogy a későbbiekben az adott generikus objektumot aktuális típusának megfelelően használjuk-e.
-
- `List<Integer> myIntList = new LinkedList<Integer>();` // 1'
 - `myIntList.add(new Integer(0));` // 2'
 - `Integer x = myIntList.iterator().next();` // 3'
-
- A 3' sornál nem `java.lang.Object`-et ad vissza, hanem `Integer`! A fordító ellenőrizni tudja a helyes típushasználatot!



Java vs C++

- Generikusok vs. Template
 - Szintaxis és szemantika:
 - a használat szintaxisa megegyezik: `List<List<String>>`
 - de a szemantika különbözik!
 - C++ template: kiterjesztés (expansion), minden használt típushoz létrehoz a fordító egy változatot a template-tel megírt kódhoz.
 - Java generikus: Type Erasure, fordítási időben ellenőrzi a típushasználatot, majd eltávolítja az összes generikust a kódból.
 - Ezzel a módszerrel a fordított kód futtatása nem igényel plusz funkcionalitást a virtuális géptől, így Java SE 5.0 előtti környezettel is fut, tehát teljesen visszafele kompatibilis.
 - A generikus paraméter csak típus lehet.
 - Ebből következően: lehet-e a generikus paraméter primitív típus?
 - Valamint létre lehet-e hozni új objektumot, amelynek ez a típusa?



- `List<String> words = new ArrayList<String>();`
- `words.add("Hello ");`
- `words.add("world!");`
- `String s = words.get(0) + words.get(1);` `// "Hello world!"`
- `List words = new ArrayList();`
- `words.add("Hello ");`
- `words.add("world!");`
- `String s = ((String) words.get(0)) + ((String) words.get(1));`
- `// "Hello world!"`

- `List words = new ArrayList();`
- `words.add("Hello ");`
- `words.add("world!");`
- `String s = ((String) words.get(0)) + ((String) words.get(1));`
- `// "Hello world!"`



Generikus osztály szintaxisa

- Az osztály neve után <> zárójelek között adjuk meg a formális típusparamétert.
- Ezután a formális paraméterrel deklarálhatunk tagváltozót, függvényparaméter típusát, lokális változót vagy visszatérési értéket:

```
class Info<T> {  
    T obj;  
  
    public Info(T obj){  
        this.obj = obj;  
    }  
  
    public String getInfo() {  
        return obj.getClass().toString();  
    }  
    public T getObj(){  
        return obj;  
    }  
}
```

- Ezután létrehozhatunk az osztályból új példányt:
 - `Info<Integer> info = new Info<Integer>(1);`



Generikus metódus szintaxisa

- A metódus szignatúrájában, a visszatérési érték előtt, < > zárójelek között adjuk meg a formális típusparamétert.
- Ezután a formális paraméterrel deklarálhatunk függvényparamétert, lokális változót vagy visszatérési értéket.

```
public <T> Info<T> getInfoArray(T a, T b){  
    return new Info<T>[] {  
        new Info<T>(a),  
        new Info<T>(b)  
    };  
}
```

- Ezután a következőképpen hívhatjuk meg:
 - `AdditionResult info = new Info<Integer>(1);`



Formális típus névkonvenciók

- Néhány tipikus formális típussal gyakran találkozunk a Java dokumentációban.
- Ezek jelentése következő:
 - E - Elemek (Collections Framework használja)
 - K - Kulcs
 - N - Szám
 - T - Típus
 - V - Érték
 - S, U stb. - További típusok



A "diamond" operátor <>

- Java SE 7-től lehetőségünk van arra, hogy nem írjuk ki a generikus paramétereket olyan esetekben, amikor a kontextusból egyértelműen következik.
- Példa:
 - `List<Integer> integerList = new ArrayList<>();`



Nézzük meg az első példát!

- Nyissuk meg a **generics.example1** csomagban található **WrongContainer.java** fájlt!
 - A **WrongContainer** osztály egy darab objektumot tud eltárolni.
 - Ismert számot teszünk bele, vagy konzolról olvasunk.
 - Amit látunk a tárolóból, hogy **Object**-et lehet beletenni.
 - Tegyük bele egy **String**-et (hiszen annak őse az **Object**).
- Futtassuk le a programot! Mit tapasztalunk?
 - A tároló szerkezet nem oldja meg a konverziót **Integer** és **String** között! (**java.lang.ClassCastException**-t kapunk)
 - Generikusokkal előírhatjuk a típusalmazt.
- Most nyissuk meg a helyes megoldást adó **CorrectContainer.java**-t!



Generikus metódusok és konstruktorok

- Lehetőség van arra, hogy az osztály formális típusparaméterétől eltérő típusparaméterű metódusokat hozzunk létre.
 - Nézzük meg a `generics.example2` csomagot!
- A `Box` típusparamétere `T`, de az `inspect()` metódusé `U`.
 - Ha a `Box Integer` aktuális típusparaméterű, ez alatt az `inspect()` metódusé lehet `String`, `Double`, stb...
- Sőt, olyan osztályban is deklarálhatunk generikus metódust, amelynek nincsen típusparamétere.
 - Nézzük meg a `Printer` osztályt, amely ezt demonstrálja!



Típushalmaz korlátozása

- Felhasználhatjuk az osztályok közötti öröklődést arra, hogy korlátozzuk a behelyettesíthető típusok halmazát.
- **Felső korlátot szabhatunk**
 - **public class** Box <U **extends** Number>
Ebben az esetben csak a Number osztály és azok leszármazottai lehetnek a behelyettesíthető típusok.
- **Előírhatunk interface-t is**
 - **public class** Box <U **extends** Number & VectorProduct>
Ebben az esetben a behelyettesíthető típusok lehetnek a Number osztály és azok leszármazottai, **HA** implementálták VectorProduct interfészt!



A wildcard <?>

- A generikusok esetén a wildcard karakter <?> egy ismeretlen típusparamétert reprezentál. Használhatjuk függvényparaméter, mező, lokális változó, visszatérési érték típusparamétereként.
 - Például:
 - `Class<?> c;`
 - `public void printList(List<?> list){}`
 - Használhatjuk, ha
 - a típusparaméterre írt kód csak az `Object` osztály függvényeit hívja meg,
 - a generikus kód nem függ a típusparamétertől (pl. `List.size()`, `List.clear()`).
 - Miért érdemes használni?
 - az `Integer` hiába altípusa az `Object`-nek, a `List<Integer>` nem lesz altípusa a `List<Object>`-nek, de a `List<?>`-nek igen.
- ```
•public void processObjectList(List<Object> list){}
•public void processWildcardList(List<?> list){}
//...
•ArrayList<Integer> intList = new ArrayList<>();
•processObjectList(intList); //fordítási hiba
•processWildcardList(intList);
```



# A wildcard típusának korlátozása

- Az **extends** és a **super** kulcsszavakkal lehet korlátozni az ismeretlen típusparamétert.
- Felső korlát (**extends**)
  - Pl.: olyan listát szeretnénk átvenni, amelyben szám tárolódik (`List<Integer>`, `List<Double>`, stb).
  - **public void** `printList(List<Number> l)` { } // nem jó
  - **public void** `printList(List<? extends Number> l)` { }
- Alsó korlát (**super**)
  - Pl.: bármilyen listát átveszünk, amelyben `Integer` tárolódhat, akár `List<Number>`, vagy `List<Object>`.
  - **public void** `printList(List<Integer> l)` { } // nem jó
  - **public void** `printList(List<? super Integer> l)` { }
- Tehát a `List<? super Integer>` vagy a `List<? extends Integer>` a `List<Integer>` lehetséges típusparaméterének bővítését jelenti.
- Nézzük meg az előzőleg már ismert **Printer** osztály wildcard-os metódusait!
- További információért látogassátok meg az alábbi két linket
  - <http://docs.oracle.com/javase/tutorial/java/generics/wildcards.html>
  - <http://ted-gao.blogspot.hu/2012/01/type-wildcard-in-java-generics.html>



# Collections Framework



# Collections Framework

- Egy keretrendszer, amely a Java mérnökei által megírt adatszerkezeteket és algoritmusokat tartalmazza.
- Továbbá lehetőséget biztosít saját adatszerkezetek és algoritmusok beillesztésére.
- Ezen saját implementációk teljesen beilleszthetők a már meglévő szolgáltatások rendszerébe.



# Collection Framework

- A keretrendszer tartalmaz:
  - kész adatszerkezeteket: általános célú implementációk
  - kész algoritmusokat: min, max keresés, rendezések
  - egységes API-t (Application Programming Interface): saját implementációk egyszerű integrálására
- Saját implementációinkat nem kell nulláról megírnunk, különböző szinteken tudunk becsatlakozni a keretrendszer hierarchiájába (lásd később).
- Az órán megírt saját megvalósításoknak nem kell minden függvényt definiálni: amire nincs szükség, annak a törzsében egyszerűen `UnsupportedOperationException` dobódik –
  - azaz a saját collection-ünk (pl. `MyMap`) nem fogja ezt a metódust támogatni.
  - Komoly programokban természetesen ezt nem így oldják meg...

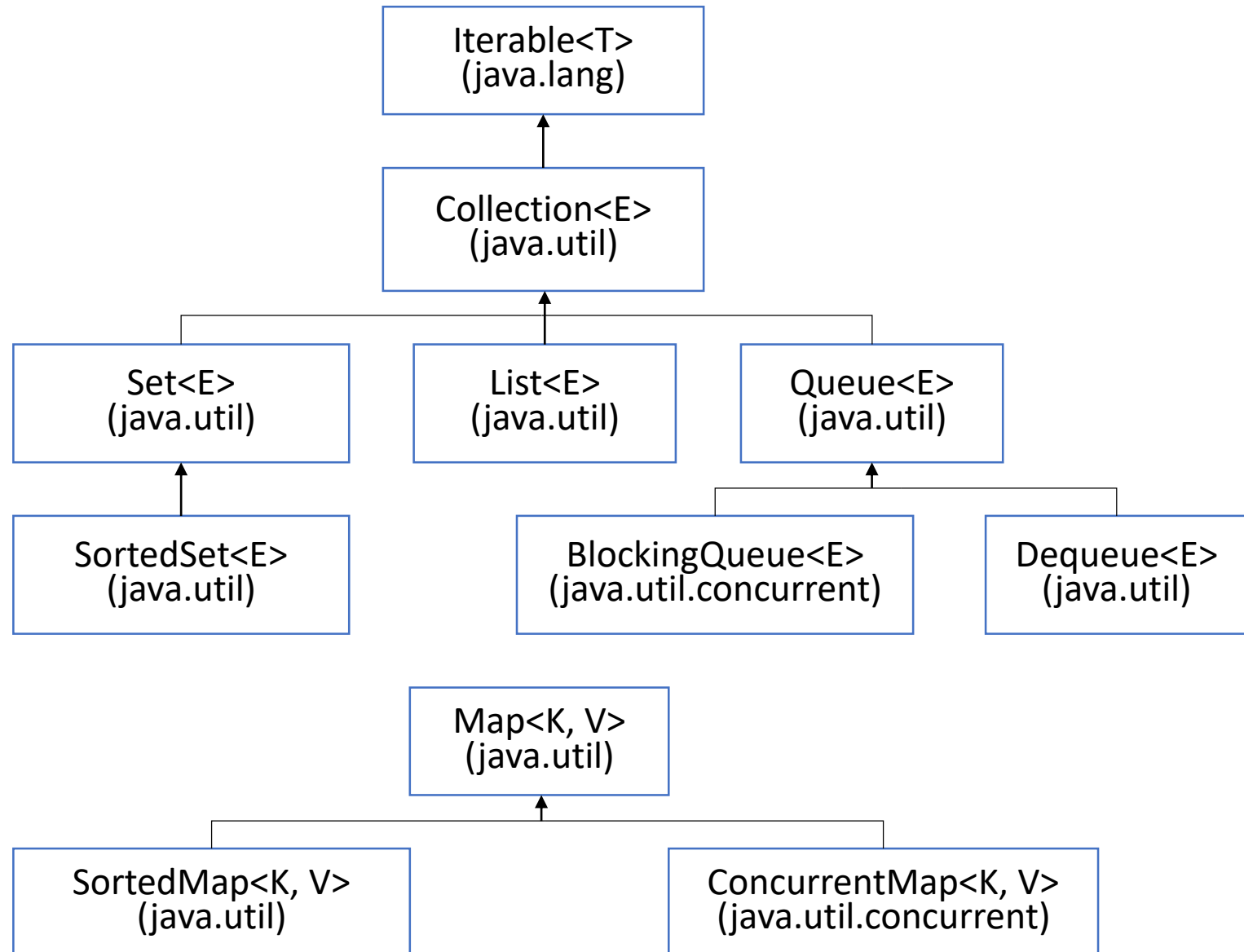


# Collection Framework

- Két fontos építőelem:

| Collection Interface     | Map Interface                                       |
|--------------------------|-----------------------------------------------------|
| Objektumok csoportja.    | Az objektumok kulcs-érték párokként vannak tárolva. |
| Halmazok, listák, sorok. | Kulcs-érték pár tömbök, asszociatív tömbök.         |

# Collections Framework interfész hierarchiája





# Collection Framework interfész hierarchiája

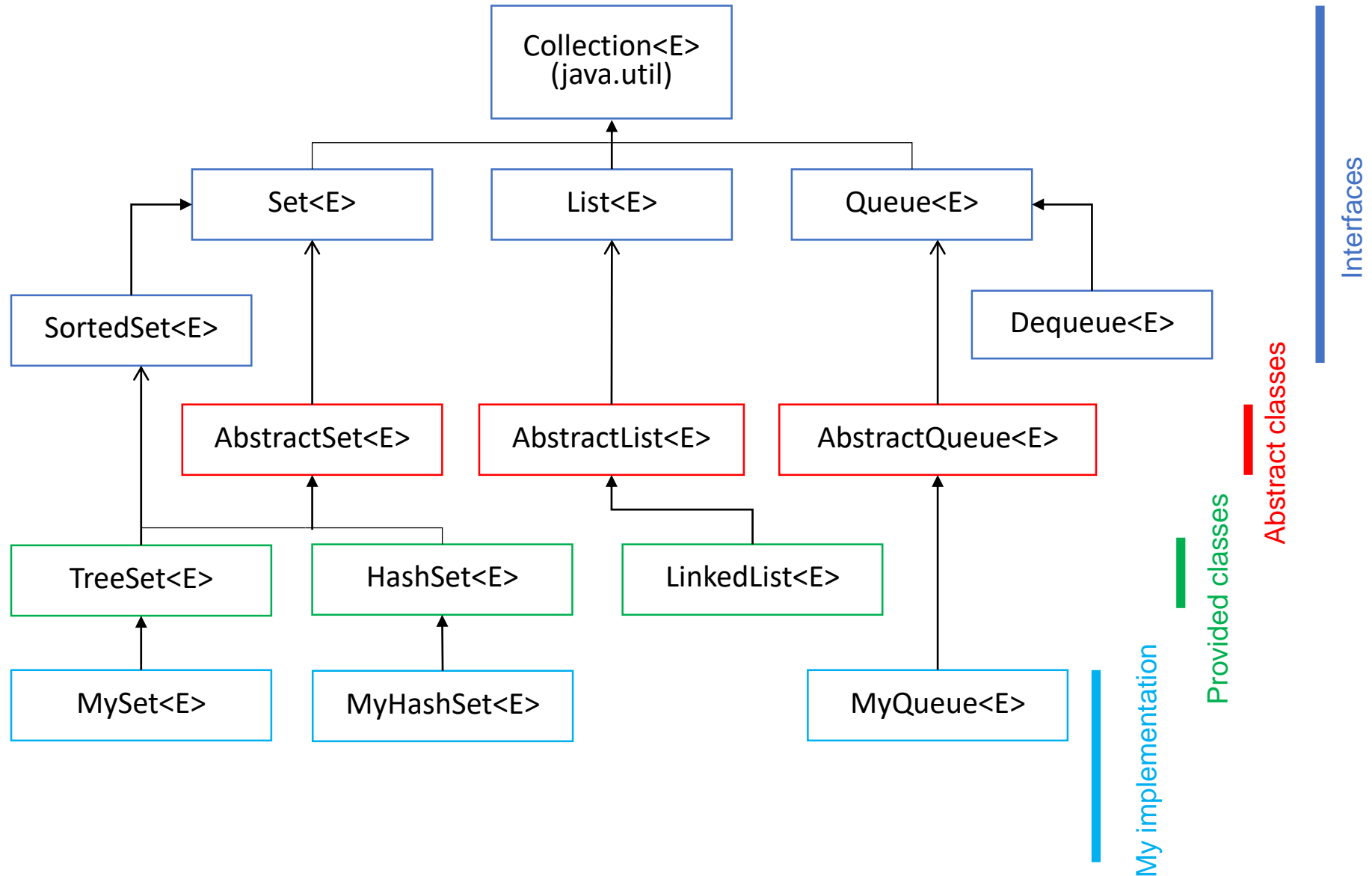
- Az alábbi három linken meg lehet nézni teljes valójában a hierarchiába tartozó interfészeket és osztályokat.
  - <http://www.holub.com/goodies/java.collections.html>
  - <http://docs.oracle.com/javase/8/docs/api/java/util/package-tree.html>
  - <http://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html>



# Collection interface

- Objektumok csoportját tárolja, egy tárolt objektumot **Element**nek hívunk.
- A `Collection` a hierarchia legfelső eleme (**alapvető metódusokat** tartalmaz egy adatszerkezethez való hozzáféréshez).
  - A megvalósítandó adatszerkezeteknek kötelező metódusait írja elő:  
`add()`, `remove()`, `isEmpty()`, `size()`, `toArray()`, stb.
  - A `toArray()` metódus lehetővé teszi a kapcsolatot olyan API-kkal melyek tömbökön alapulnak (a `collection` elemeit leképezi egy tömbbe).
  - <http://docs.oracle.com/javase/8/docs/api/java/util/Collection.html#toArray-->
- A `Collection` az `Iterable` interfészből származik, tehát a megvalósított adatszerkezetek iterátorral bejárhatóak.

# Collections Framework: Collection hierarchiája







# Hierarchia szintek jelentősége

- Az osztályok kétféleképpen valósíthatják meg az egyes interfészeket:
  - **Absztrakt módon:** csak az alapmetódusok megvalósítását tartalmazzák. Ezeket az osztályokat örököltethetjük például saját collection osztályba, ahol megvalósítjuk a további szükséges metódusokat. (**piros** színnel jelölve)
    - Nyissátok meg az ArrayList közvetlen ősének, az AbstractList-nek az implementációját, és keressetek rá az abstract metódusokra. Ezeket a metódusokat kell a leszármazottakban implementálni.
  - **Közvetlen példányosíthatóak:** Az adatszerkezethez tartozó minden metódus megvalósítását tartalmazzák. (ábrán **kékkel**, illetve **zölddel** jelölve).



# Collection interface leszármazottai

- **Set**: klasszikus halmaz, nem tartalmazhat két azonos elemet, equals() alapján.
- **SortedSet**: rendezve tartalmazza az elemeket.
- **List**: sorban lévő elemek, index létezik.
- **Queue**: feldolgozás előtti tárolásra, a peek() metódussal belenézhetünk
  - (Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.)
- **Deque**: Double Ended Queue (mindkét végére tehetünk bele elemeket).
- **BlockingQueue**: több szál által használható Queue
- Ezekben az interfészekben leírt metódusok írják elő az adott adatszerkezetre alkalmazható műveleteket.



# Collection példa

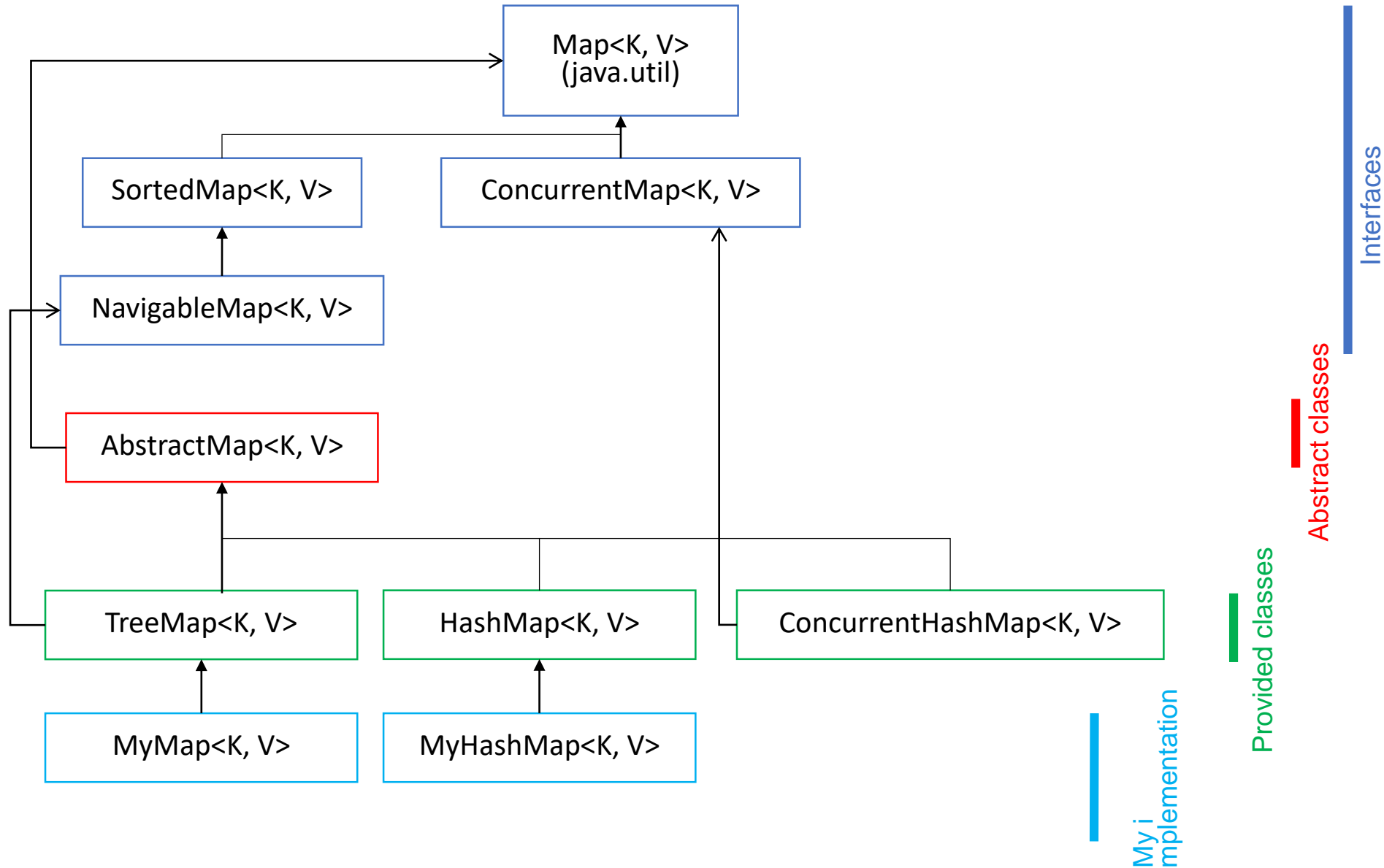
- Nyissuk meg a **`collections.collectiontest`** csomagot!
  - Implementáljunk egy osztályt, mely megvalósítja a `List` interfészt!
  - Teszteljük le a `LinkedList`, `ArrayList` és a saját listánk sebességét
    - feltöltéskor
    - index alapján történő eléréskor
    - bejárásnál
  - A futási sebességek mérésénél használhatjátok a `System` osztály `currentTimeMillis()` metódusát!



# Map Interface

- Kulcs-érték párokat tartalmaz (**Key-Value**)
- **Nem a Collection interface leszármazottja**
- Elemek elérése:
  - `V get(Object key)` – Kulcs alapú lekérés
  - `Set keySet()` – Kulcs halmaz lekérése
  - `Collection values()` – Értékek Collection-ként
  - `Set entrySet()` – Kulcs-érték párok halmazként

# A Map hierarchiája





# Java adatszerkezetek összefoglalása

- Teljes funkcionalitású adatszerkezetek.
- Optimalizált megoldásokat tartalmaznak.
- A Java forráskód letölthető és ezek az implementációk megnézhetőek.
- Legfontosabb adatszerkezetek
  - Dinamikus tömb (`ArrayList`), Láncolt lista (`LinkedList`), Sorok (`Queue`), Halmaz (`Set`)
  - Kiegyensúlyozott (piros-fekete) fa: `TreeMap` és `TreeSet`
  - Hasítótábla: `HashMap` és `HashSet`
  - Hasító tábla + Láncolt lista: `LinkedHashMap`



# java.util.ArrayList osztály

- A `List` interfészt valósítja meg.
- Az elemelérés konstans idejű (a reprezentáció egy változtatható méretű tömb).
- `initialCapacity`: a belső tömb kezdeti mérete.
- Az Auto-boxingnak köszönhetően a primitív típusok külön konvertálás nélkül berakhatóak egy `ArrayList`-be.

```
List<Integer> l = new ArrayList<Integer>();
l.add(10);
```



# java.util.ArrayList

- Kivételkezelés

- `List<Integer> l = new ArrayList<Integer>();`  
`l.add(5, 4);`
  - `java.lang.IndexOutOfBoundsException`
- `List<Integer> l = new ArrayList<Integer>();`  
`l.get(1);`
  - `java.lang.IndexOutOfBoundsException`





# Map példa

- Nyissuk meg a **`collections.maptest`** csomagot!
  - Teszteljük le a `HashMap`, `TreeMap` és a `LinkedHashMap` sebességét
    - feltöltéskor
    - kulcs alapján történő elemkeresésnél
    - és bejárásnál
  - A bejárás esetében figyeljük meg a fenti implementációknál észlelhető sebességkülönbségeket!



# Adatszerkezetek bejárása

- For-each segítségével

```
Iterable<T> list = new ArrayList<T>;
for (T t : list) {
 t.doSomething();
}
```

- Iterátor segítségével

- ```
Collection<T> collection = ...;  
Iterator<T> iterator = collection.iterator();  
while (iterator.hasNext()) {  
    T element = iterator.next();  
    if (removalCheck(element)) {  
        iterator.remove();  
    }  
}
```

<<Interface>>

Iterator<E>

+hasNext() : boolean

+next() : E

+remove() : void



Elemek összehasonlítása

- Általános elvárás a Collections Framework által tárolható elemekkel szemben, hogy összehasonlíthatóak legyenek.
- Három lehetőség adott Java-ban
 - Comparable interfész megvalósítása
 - Comparable interfészt megvalósító osztály leszármazottjában a `compareTo()` metódus felüldefiniálása
 - Comparator osztály készítése



Comparable interfész

- A Comparable interfészt megvalósító osztály rendelkezik egy `compareTo()` tagfüggvénnyel.
- Segítségével az adott objektum össze tudja magát hasonlítani egy másik azonos típusú objektummal.
 - Negatív integer: ez az objektum (`this`) kisebb, mint a paraméterben szereplő
 - Nulla: a két objektum egyenlő
 - Pozitív integer: ez az objektum (`this`) nagyobb, mint a paraméterben szereplő

<<Interface>>

Comparable<T>

+compareTo(o : T) : int




Comparable interfész

- A Java legtöbb típusa megvalósítja a Comparable interfészt

- `Double dd = 5d;`
- `boolean b = dd.compareTo(4d) > 0; // true`

- Tipp: a kifejezés „megfejtése” ugyanaz, mintha a `compareTo` utasítást kicserélnénk a relációs jellel:

- `dd.compareTo(4d) > 0;`  `dd > 4d;`

- ezért a Collections Framework `sort()` statikus függvénye működik ezekre a típusokra.

```
Integer[] ai = { 3, 2, 1, 4 };  
Integer[] sorted = { 1, 2, 3, 4 };  
Arrays.sort(ai);
```

- `Arrays.equals(ai, sorted); // true`



Comparator osztály

- Két azonos típusú objektum összehasonlítására szolgál.
- Ahhoz, hogy egy osztály komparátor legyen, a Comparator interfészt kell implementálnia.
- A `compare()` metódus viselkedése azonos a korábban bemutatott `compareTo()` metódussal.

```
<<interface>>
```

```
Comparator<T>
```

```
+compare( o1 : T, o2 : T ) : int
```

```
+equals( o : Object ) : boolean
```



Összehasonlítás példa

- Nyissuk meg a **collections.compare** csomagot!
 - Nézzük meg ComparableTable osztályt!
 - Majd futtassuk le a Main-t!
 - Próbáljuk kikommentezni a 17. sort!
 - Írjuk meg a Comparator osztályt a Table-hez!
 - Írjuk át a Main-t, hogy a 17. sor a Comparator-t használja!
 - (A Table és ComparableTable osztályok teljesen redundánsak. Hogy lehetett volna ezt kódismétlés nélkül megoldani?)



Aggregált műveletek

- Java 8 óta rendkívül egyszerű módon lehet Collection-ön műveleteket végezni.
- A `Collection.stream()` függvény egy `Stream<E>` típusú objektummal tér vissza, amin nagyon egyszerűen lehet bejárást, átlagszámítást, szűrést, leképezést stb. elvégezni for ciklusok nélkül.
- A műveletek „belsejét” legkényelmesebb lambda kifejezéseként átadni.

```
List<String> l = ...;  
l.stream().forEach(e -> System.out.println(e));
```

```
List<String> l = ...;  
l.stream()  
    .distinct()  
    .filter(e -> e.contains("Java"))  
    .count();
```




Gyakorló feladat G05F01

- A házi feladatban egy TaskManager alkalmazást kell írnod Java Generics és Collections Framework segítségével.
- Definiálj egy Task interfészt, mely a Comparable-ből származik és van egy void doTask(), illetve int getPriority() metódusa.
- Az interfészt valósítsa meg az AbstractTask osztály, amely tartalmaz name és priority mezőt, továbbá értelemszerűen kifejti a getPriority() és a compareTo() metódusokat.
- Az AbstractTask-nak két final leszármazottja legyen:
 - SimpleTask, mely a doTask() metódusban kiírja a task nevét
 - Illetve egy generikus ListTask, amely Task típusú elemekből álló listát képes nyilván tartani mezőjében, és ezt a listát kiírni a doTask() metódusban. Ezen kívül felüldefiniálja a getPriority() metódust, és a lista feladatainak átlag prioritásának egészre kerekített értékével tér vissza.
- Az előzőekben felsorolt paramétereket az osztályok konstruktoraikban vegyék át.



Gyakorló feladat G05F01 (folyt.)

- A feladatokat a TaskManager osztály kezelje.
- Ez egy asszociatív adatszerkezet tartalmazzon, amely a hét napjaihoz rendel Task-okból álló listát. A hét napjait egy Day enum típussal reprezentáld.
- A TaskManager-nek a következő két művelete legyen:
 - Az addTask(Day, Task) metódusával adható új feladat az adott naphoz. Egy naphoz maximum 10 feladat tartozhat, ha 11.-et akarunk hozzáadni, dobódjon CapacityAlreadyReached kivétel.
 - Az completeTask(Day) metódus a feladat elvégzéséért felel, azaz meghívja az adott Task doTask() metódusát, továbbá eltávolítja azt a napi listából. Amennyiben az adott napon nincs már feladat, dobódjon NoTaskException.
 - A metódusoknak ügyelniük kell arra, hogy a listák mindig prioritás szerint legyenek rendezve.
- A program rendelkezzen konzolos menüvel, amelyen keresztül a fenti két művelet végrehajtható.



Gyakorló feladat G05F02

- A feladat célja egy egyszerű Blackjack játék megírása, tétek nincsenek csak a lapok összértéke számít.
- Hozz létre egy `Player` osztály, a `Player`nek van neve, tárolja, hogy hányszor nyert, valamint tárolja, hogy aktuálisan milyen kártya lapok vannak nála.
- A `Player`től lelehet kérdezni, hogy mennyi a nála lévő kártyák összértéke, valamint ellehet dobálni vele a kártyákat (új kör), illetve kártyát lehet neki adni.
- Természetesen van `Card` osztály is, ami tárolja a kártya értékét és egy `Deck` adatszerkezetbe lehet őket eltárolni, majd a `Collection Framework` shuffle segítségével megkeverni. (a kártya színe legyen enum – Treff, Káró, Kőr, Pikk)
- Az osztó maga is játékos, ezért a `Player` osztály leszármazottja. Plusz funkciója, hogy lapot oszt a pakliból, ha még van benne lap.
- A `Player`eket (osztót is), egy olyan adat szerkezet tárolja, aminek maximum 4 elem lehet.



Gyakorló feladat G05F02 (folyt.)

- Játék menete a következő. Az osztó megkapja a Playereket tartalmazó adatszerkezet referenciáját.
- A pakliból húzásra fogynak a kártyák. Minden kör végén a használt kártyákat egy külön pakliba gyűjtjük.
- Az osztó minden körben húz egy lapot a pakliból és oda adja a soron következő játékosnak
- Amennyiben a kör végén valakinek a lapjainak összértéke 21, az nyert.
- Ha valakinek több pontja van, mint 21 akkor ne kapjon tovább lapot, aktuális játék végével újra játszhatson. (Player osztály boolean változó)
- Ha elfogy a pakliból a kártya akkor egy kivétel váltódik ki és vége a játéknak
- Plusz pontért lehetséges extrák
 - Ha elfogyott a pakliból a kártya akkor kivétel váltódjon ki, de ez olyan szinten kerüljön kezelésre, hogy a játék folytatható legyen egy új paklival.
 - A pontjai mindenkinek megmaradnak, de a kezükben lévő kártyát visszavesszük és meg keverjük a használt paklit és ez lesz a új pakli kártya.
 - Ezzel együtt is max 1.1 pont szerezhető, és plusz pont csak akkor kerül beszámításba, ha anélkül maximális 1 pont teljesítve lett!!!



Gyakorló feladat G05F03

- A házi feladatban kézilabda bajnokságot kell szimulálni
- Szükség van egy `HandballPlayer` osztályra, minden játékosnak van neve és mezszáma (ami most speciálisan bajnokságszinten egyedi)
 - Illetve egy `playHandball()` metódusa, mely kiírja: „i am shooting”
- Kapusokat a `GoalKeeper` osztály reprezentálja
 - Ez definiálja felül a metódust, és írja ki: „i am saving”
- Ezek implementálják a `Comparable` interfészt, a mezszám alapján történjen az összehasonlítás
- `HandballTeam` osztály kézilabda csapatot reprezentál
 - Játékosok halmazát tartalmazza
 - Illetve egy `points` mezőt, amely a bajnokságbeli pontjainak számát jelenti
 - Továbbá a csapatnak neve is legyen
 - `transfer(Player who, HandballTeam from)` metódussal átigazolhat játékos ebbe a csapatba (ekkor természetesen az előző csapatból törölni kell, a mostanihoz hozzáadni)
 - Készíts egy `HandballTeamComparator` osztályt, ami két csapatot képes összehasonlítani, a pontszámuk alapján



Gyakorló feladat G05F03 (folyt.)

- HandballLeague osztály a bajnokságot reprezentálja
 - Listában a csapatokat tartalmazza, az elsőt a 0. indexen stb.
 - `match(HandballTeam, HandballTeam)` metódusa lejátszik egy mérkőzést, és az eredmény alapján 2 illetve 1 pontot ad (akár random módon) a megfelelő csapat(ok)hoz, ezután frissíti a lista állását
 - `getPlayer(int)` metódusa mezszám szerint le tud kérni egy játékost
 - Mivel ehhez minden csapat összes játékosán végig kéne iterálni, ennek gyorsítása érdekében a bajnokság kezdetekor (konstruktor) töltsön fel egy asszociatív adatszerkezetet `int → HandballPlayer` párokkal, amit ez a metódus felhasznál
- Szimuláld le a bajnokságot:
 - Hozz létre pár csapatot, töltsd fel játékosokkal azokat
 - Hajts végre egy-két átigazolást, `getPlayer()` metódussal kérd le a játékosokat
 - Hozz létre egy bajnokságot és add hozzá a csapatokat
 - Majd az összes lehetséges meccset játszd le egyszer a bajnokságban
 - Végül írd ki a bajnokság végeredményét, azaz a végső sorrendet
 - Illetve a csapatok összes játékosát, mezszám szerint sorba rendezve



Gyakorló feladat G05F04

- Egy osztály tanulmányi eredményeinek modellezése.
- A tanulókat egy absztrakt osztály reprezentálja (Student). Egy konstruktora van, amely beállítja a tanuló nevét, valamint két absztrakt metódusa: `getAverage(): double`, és `doExam(): void`.
- Háromfajta tanuló van, amelyek különböző tárgyakat tanulnak: nyelvi (angol, német), humán (irodalom, történelem) és tudományos (matek, fizika). Mindegyik tantárgy egy-egy integer példányváltozó. Mindhárom tanulótípust egy-egy osztály reprezentálja, amelyek a tanulókat reprezentáló absztrakt osztály leszármazottjai. A `doExam` metódust úgy írják felül, hogy az adott tanuló tantárgyaihoz egy-egy véletlenszerű osztályzatot rendelnek 1 és 5 között. A leszármazott tanuló osztályok konstruktorában hívjuk meg a `doExam` metódust. A `getAverage` értelemszerűen a jegyek átlagát adja vissza.
- A teljes csoportot a Group osztály reprezentálja. A háromféle tanuló tárolására három listát tart fent. Az `addStudent()` metódusa egy generikus típusparaméterrel rendelkezik, amellyel bármilyen tanulót hozzá tud adni a megfelelő collectionhoz.
- A Group osztály `listStudents()` függvénye kilistázza az összes tanulót ABC sorrendben. A `listResults()` függvény paraméterként egy listát vár, amely bármilyen típusú tanulókat tárolhat, és kilistázza az átadott Listben lévő tanulókat a tanulmányi átlaguk szerinti csökkenő sorrendben.
- Írjuk felül a tanuló `toString()` metódusát, hogy kiírja a tanuló nevét, átlagát, és az egyes tárgyakból elért érdemjegyet.
- A Group osztály működését egy `main` függvényben teszteljük.