Objektumorientált programozás

Segédosztályok

Goldschmidt Balázs

balage@iit.bme.hu

Interfészek

Geometria-példa

- `Shape` – alaposztály
  - `double getArea()`
  - `double getPerimeter()`
  - mindkettő absztrakt
- `Circle, Rectangle, Triangle` – leszármazottak
  - felüldefiniálnak minden függvényt
  - van saját belső szerkezetük

Alakzat és kör

Csak absztrakt metódusok!
Nincs adattag!

```
abstract public class Shape {
  abstract public double getArea(); // terület
  abstract public double getPerimeter(); // kerület
}
```

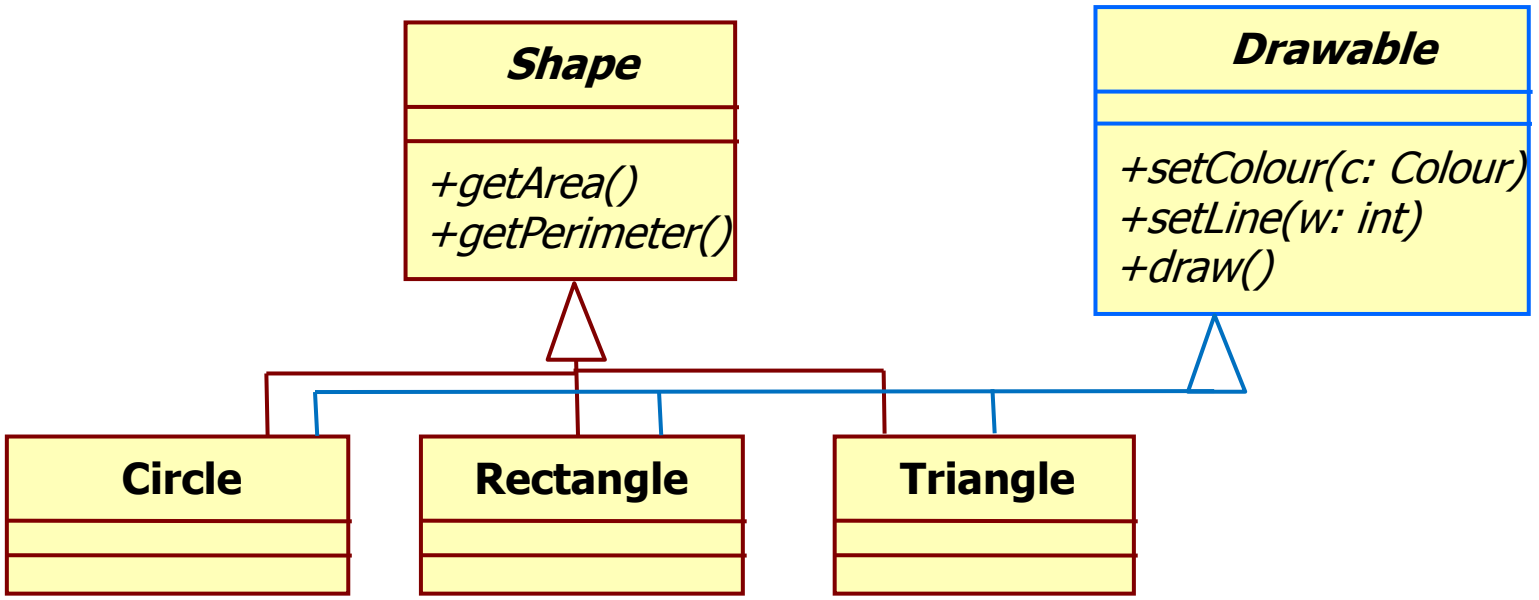
```
public class Circle extends Shape {
  private double r; // sugár
  public Circle(double r) { this.r = r; }
  public double getArea() { return r*r*Math.PI; }
  public double getPerimeter() { return 2*r*Math.PI; }
}
```

```
Circle c = new Circle(2);
System.out.println(c.getArea());

Shape s = c;
System.out.println(s.getArea());
```

Rajzolást hogyan?

- Legyen egy rajzolható (*Drawable*) osztály
  - ezt ki lehet rajzolni (*draw*)
  - lehet a színét állítani (*setColour*)
  - lehet a vonalvastagságot állítani (*setLine*)
- Cél: *Circle, Rectangle, Triangle* tudja ezt is
  - *megoldás1*: *Shape* a *Drawable* leszármazottja
    - mi van, ha valaki csak *Shape* akar lenni?
  - *megoldás2*: *Drawable* a *Shape* leszármazottja
    - meglevő *Shape*-eket át kell írni

Drawable osztály



Alakzat és kör: interfésszel

```
public interface Shape {
  double getArea(); // terület
  double getPerimeter(); // kerület
}
```

Csak absztrakt metódusok!
Nincs adattag!

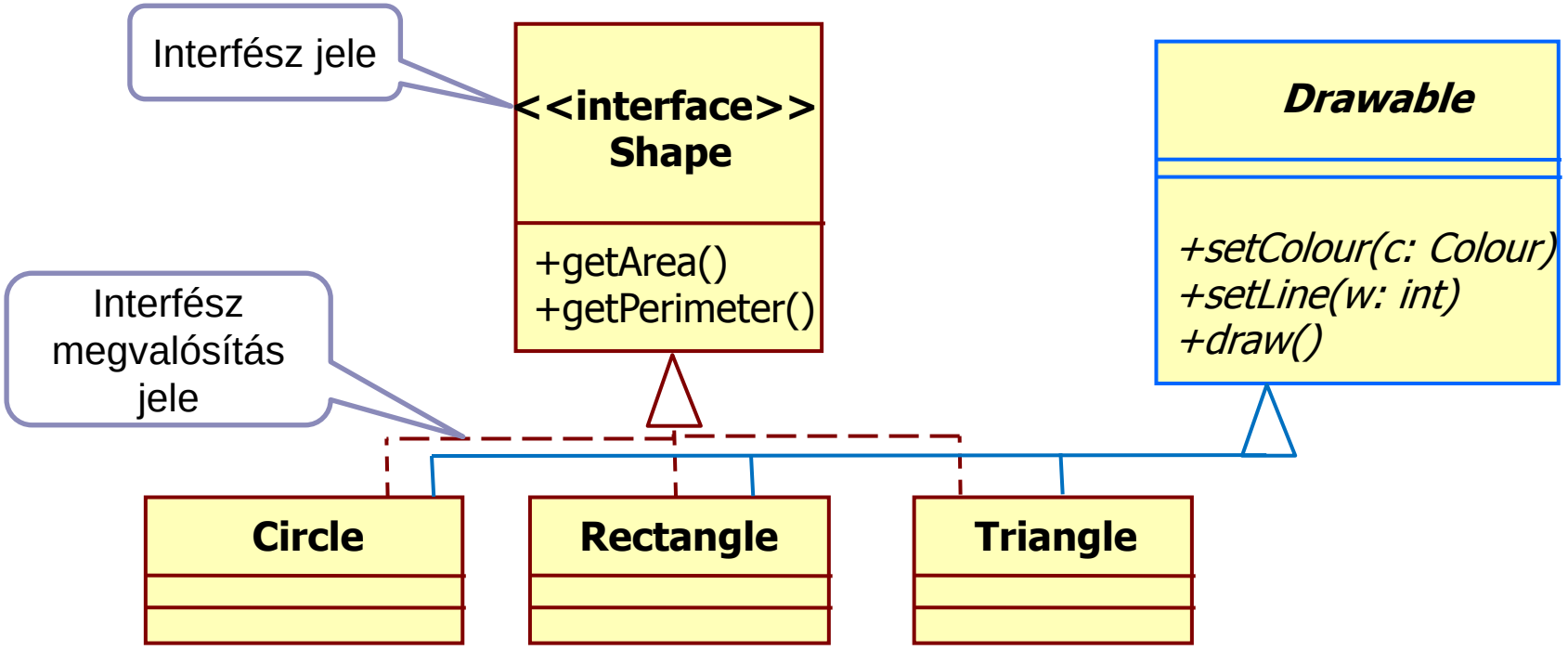
```
public class Circle implements Shape {
  private double r; // sugár
  public Circle(double r) { this.r = r; }
  public double getArea() { return r*r*Math.PI; }
  public double getPerimeter() { return 2*r*Math.PI; }
}
```

Használat ugyanaz

```
Circle c = new Circle(2);
System.out.println(c.getArea());

Shape s = c;
System.out.println(s.getArea());
```

Használat ugyanaz

Drawable osztály UML diagram



Interface (metódushalmaz)

- Célja
  - annak biztosítása, hogy egy osztály több, különböző kliens számára is az elvárt viselkedést nyújthassa
- Definiálása
  - class helyett interface
  - metódusok törzs nélkül
  - attribútum statikus, ha lehet, kerüljük
- Használata
  - statikus típusként bárhol referálható
  - megvalósító osztály *extends* helyett *implements*-cel
    - akárhány interface-t

Interface példa: Person+

```
public interface Greeter {
  String greetings(); // alapból public, nem kell jelölni
}
public interface Eater {
  String eat(String s);
}
```

```
abstract public class Person implements Greeter, Eater {
  ... // greetings és eat abstract, majd leszármazottakban
}
```

```
Student s1 = new Student("Gipsz Jakab", "1A2B3C", 1996);
Teacher t1 = new Teacher("Rend Elek", "Q1W2E3", 1973);

Greeter g1 = s1;
Greeter g2 = t1;
System.out.println(g1.greetings());
```

g1 és *g2* csak *Greeter*-ként: csak *greetings* metódus hívható!

Objektumok másolása

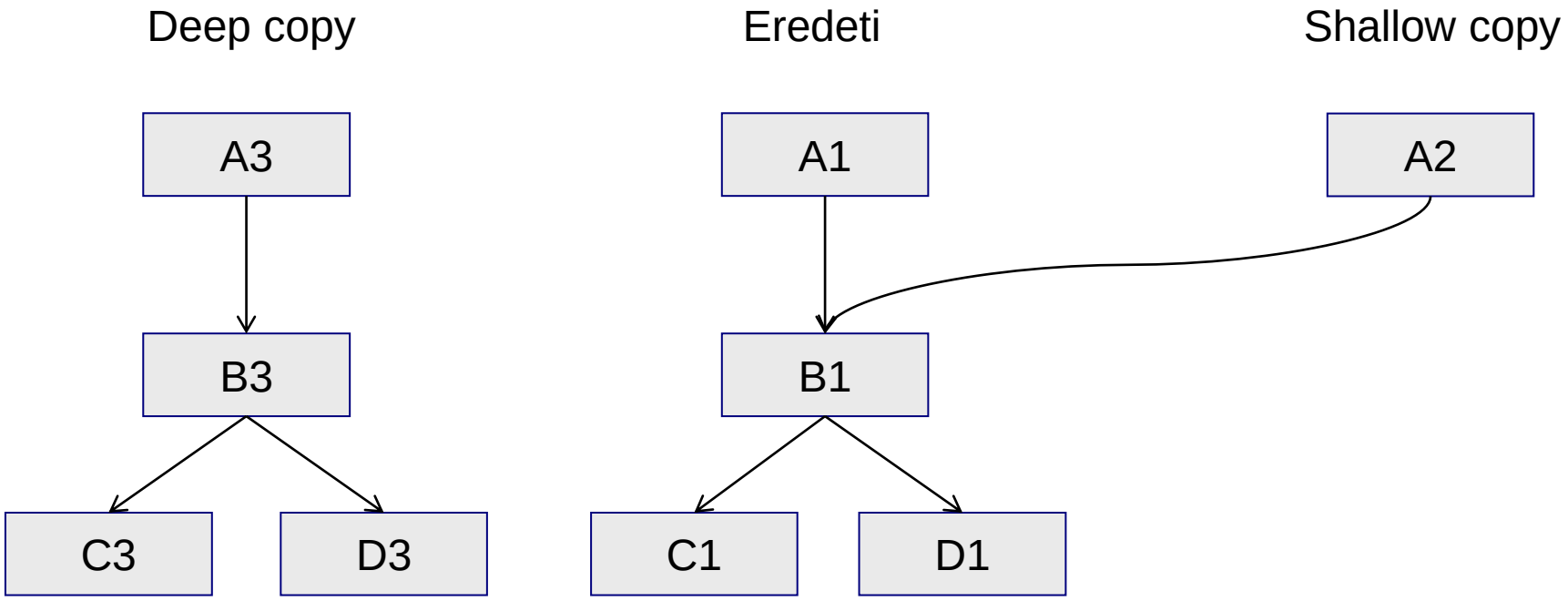
Object ősosztályban

- *Object* osztály minden osztály őse
  - definiál alapmetódusokat
  - így minden objektumon meg lehet hívni
  - az egységes működés garantált
- Metódusai
  - `public String toString()`
  - `public boolean equals(Object o)`
  - `protected Object clone()`
  - `public int hashCode()`
  - `...`

Objektumok másolása

- `java.util.Cloneable` interfész megvalósításával
- `Object.clone()` felüldefiniálásával
  - mindig hívjuk meg a **`super.clone()`** metódust
  - *Object.clone()* trükkös: dinamikus típust példányosít
    - attribútumok megkapják a hívott objektumban tárolt értéket
- *Shallow copy (sekély másolás)*
  - Csak referenciákat másolunk (ez az alapértelmezett)
    - eredményül az attribútumok ugyanoda mutatnak
- *Deep copy (mély másolás)*
  - rekurzív másolás
    - minden attribútumon meghívjuk a saját *clone()*-ját

Deep és shallow copy



Másolás: nincs ősosztály, naív

```
// naív implementáció!
public class A implements Cloneable {
	B b;
	// shallow, clone() alapból ezt csinálja
	public Object clone() {
		A a2 = new A();
		a2.b = b;
		return a2;
	}
	...
}
```

Másolás: nincs ősosztály, naív

```
// naív implementáció!
public class A implements Cloneable {
	B b;
	// deep, drágább!
	public Object clone() {
		A a3 = new A();
		a3.b = (B)b.clone();
		return a3;
	}
	...
}
```

Másolás: van ősosztály

```
public class A extends E {
        B b;
         // shallow
        public Object clone() {
                A a2 = (A)super.clone(); // !!!
                a2.b = b;
                return a2;
        }

        ...
}
```

Az *E clone()* metódusát hívja, de *A* osztályú objektumot készít!

Másolás: van ősosztály

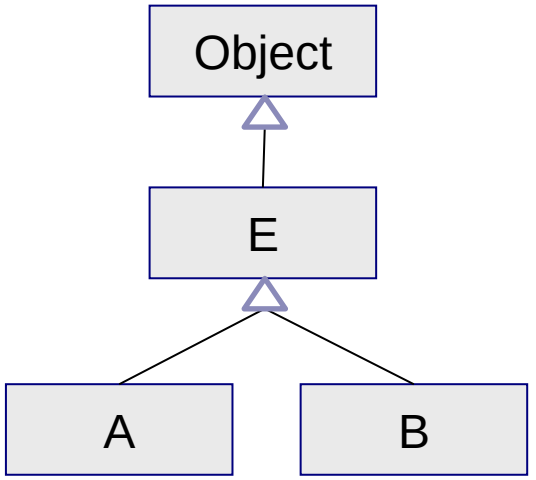
```
public class A extends E {
        B b;
        // deep
        public Object clone() {
                A a3 = (A)super.clone();
                a3.b = (B)b.clone();
                return a3;
        }
        ...
}
```

Az *E clone()* metódusát hívja, de *A* osztályú objektumot készít!

Clone az öröklésben

```
public class E implements Cloneable {
  public Object clone() {
    try { return super.clone(); // Object.clone(), legyen!
    } catch (CloneNotSupportedException e) {}
    return null;
  }
}
```

```
public class A extends E {
  B b;
  public Object clone() {
    A a3 = (A)super.clone();
    a3.b = (B)b.clone(); // deep
    return a3;
  }
  ...
}
public class B extends E { ... }
```



Gyors azonosítás: *hash*

- *public int hashCode()*
  - egyedi, objektumra jellemző int visszaadása
    - *a.equals(b)==true → a.hashCode() == b.hashCode()*
    - célja az objektumok gyors és hatékony tárolása
      - e.g. HashMap, HashSet
  - egy lehetséges alap implementáció *Object*-ben: memóriacím
- Jó hash függvényt nehéz
  - eclipse segít (*generate hashCode*)

Kollekció keretrendszer (bevezetés)

Legegyszerűbb kollekció: tömb

- Beépített típus
- Változatlan méret
  - ha 10 elemű, az is marad…
- Viszonylag kényelmes használat

Dinamikus adatszerkezetek

- Láncolt lista
  - egyszeres vagy többszörös, strázsa, fésű, stb.
- Bináris fa
  - egyensúly, vörös-fekete, AVL, stb.
  - szó-fa
- Asszociatív tároló
  - kulcs-érték párok tárolása
  - hash függvény alapján pl.

Kollekciók közös jellemzői

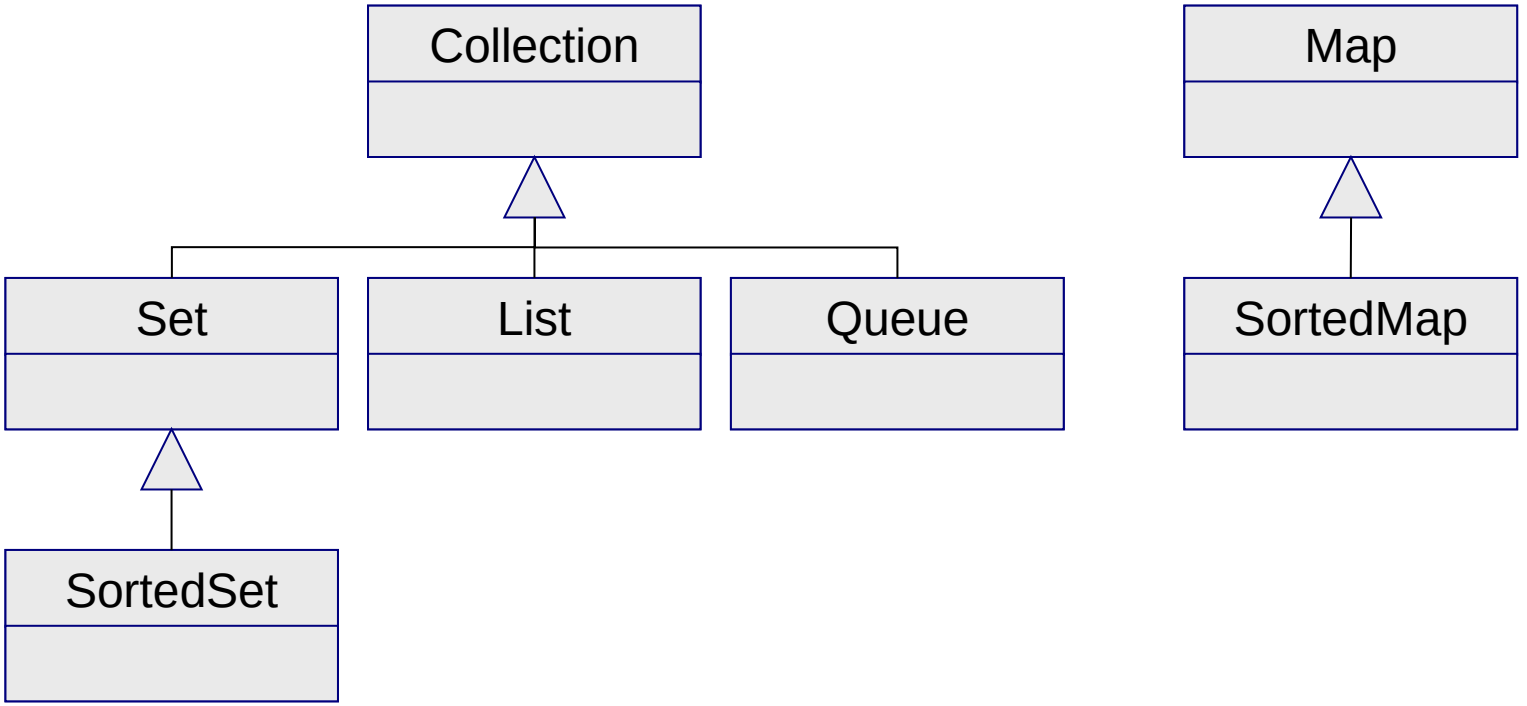
- Alapfunkciók
  - beszúrás, keresés, felülírás, törlés
    - CRUD: create, read, update, delete
  - iterálás
  - az elemek referenciáját tároljuk, kezeljük!
- Különböző megvalósítások
  - rendezett
  - halmaz vagy zsák
  - deep/shallow copy
  - különféle optimalizálások (pl. beszúrás vagy keresés)

Használatuk

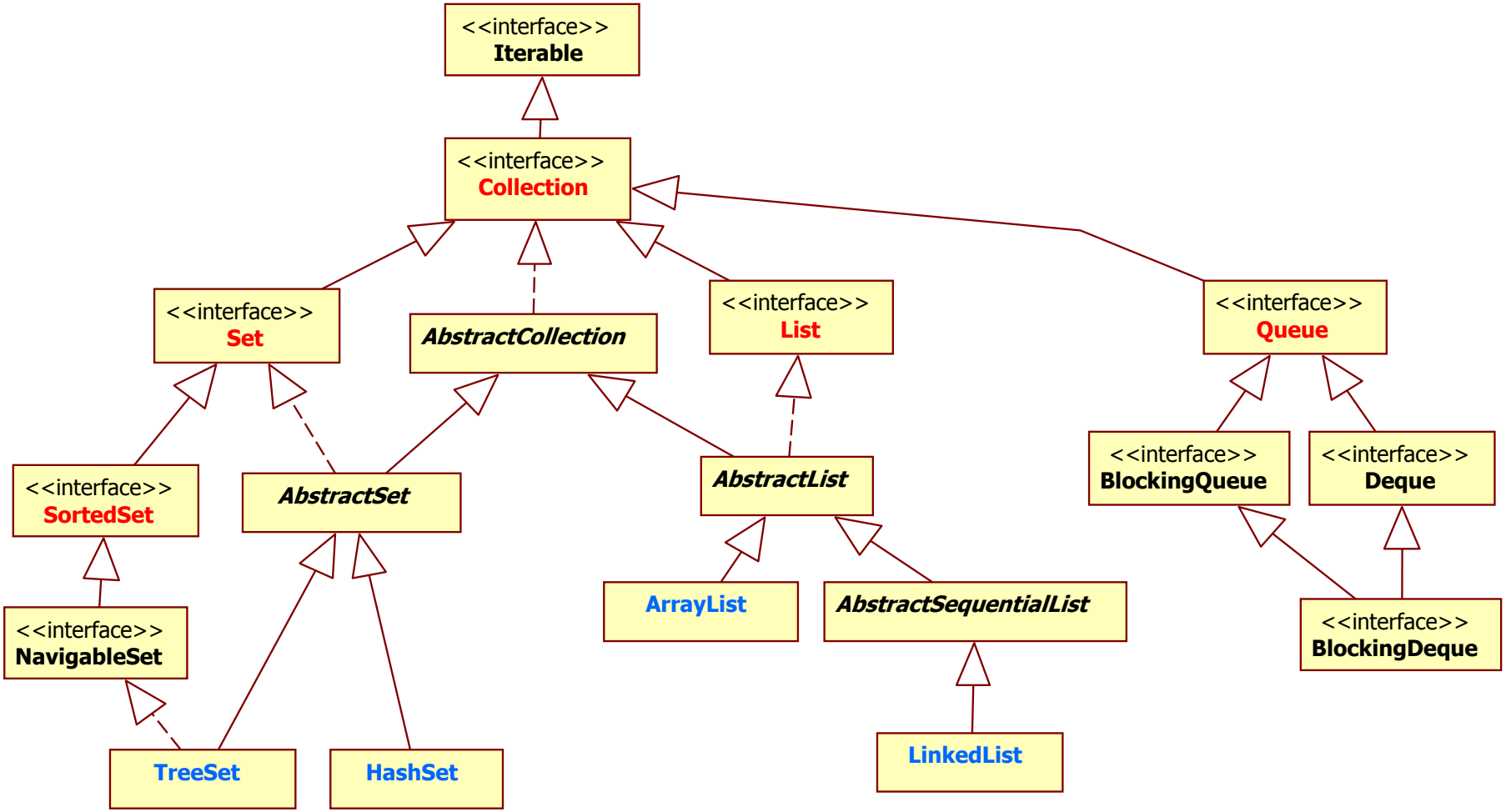
```
ArrayList<Integer> l = new ArrayList<Integer>();

// since J2SE 7
// List<Integer> l = new ArrayList<>();

for (int i = 0; i < args.length; i++) {
      l.add(Integer.parseInt(args[i]));
}

for (int i = 0; i < l.size(); i++) {
      System.out.println(l.get(i)+10);
}
```

Kollekciók (alapok)

- Interfészek



Kollekciók (bővebben)



Interface `Collection`

- Kollekciók alapja
- Közös műveleteket definiálja
- Implementáció változhat
  - néhány művelet lehet, hogy nincs implementálva
- Sablondefiníció:

```
Collection<E>
```

Interface `Collection` 2

- **`void add(E e)`** *opcionális*
  - új elem hozzáadása
- **`void addAll(Collection<? extends E> c)`** *opcionális*
  - ***`c`*** kollekció minden elemét hozzáadjuk
  - csak a referenciákat! (shallow copy)
- **`boolean remove(E e)`** *opcionális*
  - ha **`e`** benne van, törli
- **`boolean removeAll(Col<? ext E> c)`** *opcionális*
  - a **`c`** minden elemét törli a kollekcióból
  - referencia alapú! Maga az elem megmarad

UnsupportedOperationException

Interface `Collection` 3

- **`boolean contains(E e)`** *opcionális*
  - igaz, ha **`e`** a kollekcióban van
- **`boolean containsAll(Col<? ext E> c)`** *opcionális*
  - igaz, ha **`c`** minden eleme a kollekcióban van
- **`int size()`**
  - tárolt elemek száma
- **`boolean isEmpty()`**
  - igaz, ha a kollekció üres
- **`void clear()`**
  - minden tárolt referenciát eldob

Interface `Collection` 4

- **`boolean retainAll(`*`Col`*`<? `*`ext`*` E> c)`** *opcionális*
  - csak azokat az elemeket őrzi meg, amik c-ben is megvannak
- **`boolean equals(Object o)`**
  - kollekciók azonossága
  - szimmetrikus implementáció
- **`Object[] toArray()`**
  - tömbbé konvertálás
- **`<T> T[] toArray(T[] ta)`**
  - adott típusú tömbbé konvertálás
- **`Iterator<E> iterator()`**
  - iterátor visszaadása

Kollekció példa

Statikus típus: Collection

```
Collection<Integer> l2 = new ArrayList<Integer>();
Collection<Integer> l3 = new ArrayList<Integer>();

// since J2SE 7
// Collection<Integer> l2 = new ArrayList<>();

for (int i = 0; i < 100; i++) {
      l2.add(new Integer(i*2));
      l3.add(new Integer(i*3));
}
Collection<Integer> l6 = new ArrayList<Integer>();
l6.addAll(l2); // l2 tartalma l6-ba
l6.retainAll(l3); // csak a 3-mal oszthatók maradnak
```

Interface `Iterator`

- Biztosítja a kollekció elemeinek végigjárását
- Definíció: **`Iterator<E>`**
  - **`E`**-ket tartalmaz a kollekció
- **`boolean hasNext()`**
  - igaz, ha van még elem
- **`E next()`**
  - visszaadja a következő elemet
- **`void remove()`**
  - törli az utoljára visszaadott elemet a kollekcióból

Interface `Iterator` 2

- Tipikus példa: töröljük a negatív számokat!

```
Collection<Integer> c = getNumbers();

Iterator<Integer> i = c.iterator();

while (i.hasNext()) {
        int a = i.next(); // outboxing
        if (a < 0) {
                i.remove();
        }
}
```

Interface `Iterator` 3

- Többszörös hozzáférés veszélyes
  - iterálás közben nem jó, ha módosul a kollekció
  - **`ConcurrentModificationException`** dobódhat

```
Collection c = ...;
...
Iterator i1 = c.iterator();
Iterator i2 = c.iterator();
i1.next();
i2.next();
i2.remove();
i1.next();    // kivétel keletkezik
```

Interface `Set`

- Set: halmaz, minden elem csak egyszer
- Sorrend nem definiált
- Iterator tetszőleges sorrendben jár be
- Nincs új metódusa
  - csak a Collection-ben definiáltak
- Tipikus megvalósítás: **`HashSet`**
  - jó hash függvény kell a hatékony működéshez

Interface `List`

- Sorrendben levő elemek (indexelt)
- Ugyanaz az elem többször is előfordulhat
- Elemek pozíciója ismert
  - indexeléssel elérhetők
- Kereshető (objektum -> index)
- Saját iterátor (`ListIterator`)
  - Iterator-t bővíti extra funkciókkal
- Tipikus megvalósítás: `ArrayList`

Interface `List` 2

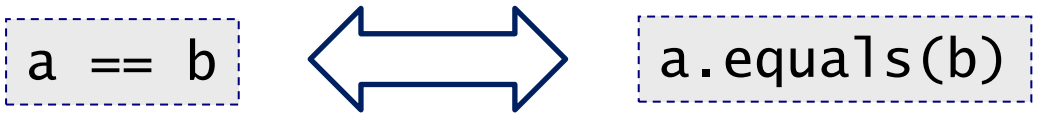
- Extra metódusok az indexelés miatt
  - `add(int index, E e)`
  - `E get(int index)`
  - `int indexOf(Object)`
  - `int lastIndexOf(Object)`
  - `E remove(int index)`
  - `boolean remove(Object o)`
  - `E set(int index, E e)`
  - `List<E> subList(int from, int to)`
  - `...`

`Collections` segédosztály

- Rendezés, min-max kiválasztás, stb.
  - `sort(List<T> l)`
  - `sort(List<T> l, Comparator<T> c)`
- Megfordítás
  - `reverse(List<T> l)`
- Rotálás
  - `rotate(List<T> l, int distance)`
- Megkeverés
  - `shuffle(List<T> l)`
  - `shuffle(List<T> l, Random r)`

Objektumok azonossága (ism.)

- `==` operátor
  - referencia-alapú azonosság
    - ugyanaz-e a két objektum?
- `boolean equals(Object o)`
  - tartalom-alapú azonosság
    - ugyanaz-e a tartalmuk?
  - rekurzió javasolt
    - ha a tartalom is equals, akkor az objektumok is
  - *alapértelmezett megvalósítás referencia-alapú*



Összehasonlítás

- Természetes rendezés
  - megvalósítandó interfész **`Comparable<T>`**
    - ahol **`T`** a rendezendő típus maga
  - **`int compareTo(T x)`**

    this < x ↔ this.compareTo(x) < 0

    this = x ↔ this.compareTo(x) = 0

    this > x ↔ this.compareTo(x) > 0
  - osztályonként egy-egy implementáció
  - fordítási időben eldől

Összehasonlítás: rendezéshez

```
public class Person
implements Comparable<Person> {
  private String name;
  private String neptun;
  //...
  // this < p, ha this.name < p.name, stb.
  public int compareTo(Person p){
    return name.compareTo(p.name);
  }
}
```

Person típushoz hasonlítható

név alapján hasonlít

```
ArrayList<Person> pp = new ArrayList<>();
//... megtöltjük

Collections.sort(pp);
```

compareTo alapján rendez

Összehasonlítás külső segítséggel

- Dedikált összehasonlító objektum (*Comparator*)
  - felelősséget kiemeljük külön osztályba
  - **`interface Comparator<T>`**
  - megvalósítandó metódusa

    `int compare(T o1, T o2)`
    - *T*-ket hasonlít

$$o1 < o2 \leftrightarrow compare(o1,o2) < 0$$
$$o1 = o2 \leftrightarrow compare(o1,o2) = 0$$
$$o1 > o2 \leftrightarrow compare(o1,o2) > 0$$

Összehasonlítás példa: rendezés

Person típusú objektumokat hasonlít össze

```
public class PersonByNeptun
implements Comparator<Person> {
  public int compare(Person p1, Person p2){
       String n1 = p1.getNeptun();
       String n2 = p2.getNeptun();
       return n1.compareTo(n2);
  }
}
```

neptunkód alapján hasonlít

```
ArrayList<Person> pp = new ArrayList<>();
//... megtöltjük


// comparator-alapú rendezés
Collections.sort(pp, new PersonByNeptun());
```

neptunkód alapján hasonlít (*PersonByNeptun.compare*)

Köszönöm a figyelmet