

Advanced Programming Methods

Lecture 3 – Java Generics and Collections

Announcement

The seminars from Monday, 21 October, 8-14
are moved to:

Thursday, 17 October, 8-10, L001

Thursday, 24 October, 8-10, L001

Thursday, 31 October, 8-10, L001

Therefore the groups 922,926,927 must come
either on 17 October, or 22October, or
24October for the Seminar4.

Lecture Overview

1. Enhanced For
2. Generics
3. Collections
4. IO

Enhanced FOR (EACH) statement

Syntax:

```
for (Type elemName : tableName)
    instr;
```

```
int[] x={1, 4, 6, 9, 10};
for (int el:x)
    System.out.println(el);
```

```
for (int i=0;i<x.length;i++)
    System.out.println(x[i]);
```

JAVA GENERICS

Generics

- Parameterized types
- Started with Java 1.5
- Different than C++ templates
 - It does not generate a new class for each parameterized type
 - The constraints can be imposed on the type variables of the parameterized types.

■ Motivation:

```
Stiva s=new Stiva();      //stack of Object
s.push("Ana");
s.push(new Persoana("Ana", 23));
Persoana p1 =(Persoana)s.pop();
Persoana p2 =(Persoana)s.pop();
//correct at compile-time, error at execution time
```

Generic Class declaration

```
[access_mode] class ClassName <TypeVar1[, TypeVar2[, ...]] >{  
    TypeVar1 field1;  
    [declarations of fields]  
    [declarations and definitions of methods]  
}
```

Obs:

Type variables must be upper letters(for example E for element, K for key, V for value, T, U, S ...).

```
public class Stiva<E>{  
    private class Nod<T>{  
        T info;  
        Nod<T> next;  
        Nod(){info=null; next=null;}  
        Nod(T info, Nod next){  
            this.info=info;  
            this.next=next;  
        }  
    }  
} //class Nod  
Nod<E> top;  
//...  
}
```

Generic Classes Usage

```
public class Test{  
    public static void main(String[] args){  
        Stiva<String> ss=new Stiva<String>();  
        ss.push("Ana");  
        ss.push("Maria");  
        ss.push(new Persoana("Ana", 23));    //error at compile-time  
        String elem=ss.pop();                //NO CAST  
  
        Stiva<Persoana> sp=new Stiva<Persoana>();  
        sp.push(new Persoana("Ana", 23));  
        sp.push(new Persoana("Maria", 10));  
  
        Dictionar<String, String> dic=new Dictionar<String, String>();  
        dic.add("abc", "ABC");  
        dic.add(23, "acc"); //error at compile-time  
        dic.add("acc", 23); //error la compile-time  
    }  
}
```


Autoboxing

Type variables can be instantiated only with reference types. Primitive types: int, byte, char, float, double,... are not allowed. Therefore the corresponding reference types are used.

```
Stiva<int> si=new Stiva<int>();  
//error at compile-time
```

```
Stiva<Integer> si=new Stiva<Integer>();
```

primitive types	Corresponding reference types
boolean	Boolean
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Autoboxing

- Autoboxing: automatic conversion of a value of a primitive type to an object instance of a corresponding reference type when an object is expected, and vice-versa when a primitive value is expected.

```
Stiva<Integer> si=new Stiva<Integer>();
```

```
si.push(23);           //autoboxing
```

```
si.push(new Integer(23));
```

```
int val=si.pop();
```

```
Character ch = 'x';
```

```
char c = ch;
```

Generic methods

■ Methods with type variables

```
class ClassName[<TypeVar ...>]{  
  [access_mod] <TypeVar1[, TypeVar2[,...]]> TypeR nameMethod([list_param]){  
    }  
    //...  
}
```

Obs:

- Static methods cannot use the type variables of the class.
- A generic method can contain type variables different than those used by the generic class.
- A generic method can be defined in a non-generic class.

Generic methods

```
public class GenericMethods {  
    public <T> void f(T x) {  
        System.out.println(x.toString());  
    }  
  
    public static <T> void copy(T[] elems, Stiva<T> st) {  
        for(T e:elems)  
            st.push(e);  
    }  
}
```

Calling a generic method

- The compiler automatically infers the types which instantiate the type variables when a generic method is called.

```
public class A {  
    public <T> void print(T x) {  
        System.out.println(x);  
    }  
  
    public static void main(String[] args) {  
        A a=new A();  
        a.print(23);  
        a.print("ana");  
        a.print(new Persoana("ana",23));  
    }  
}
```

Calling a generic method

- The instantiations of the type vars are explicitly given:

- Instance method:

```
a.<Integer>print(3);  
a.<Persoana>print(new Persoana("Ana",23));
```

- Static method :

```
NameClass.<Typ>nameMethod([parameters]);  
//...  
Integer[] ielem={2,3,4};  
Stiva<Integer> st=new Stiva<Integer>();  
GenericMethods.<Integer>copy(ielem, st);  
//
```

- Non-static method in a class:

```
this.<Typ>nameMethod([parameters]);  
class A{  
    public <T> void print(T x){...}  
    public void g(Complex x){  
        this.<Complex>print(x);  
    }  
}
```

Generic arrays

- Cannot be created using new:

```
T[] elem=new T[dim]; //error at compile time
```

but we can use:

```
T[] elem=(T[])new Object[dim]; //warning at compile-time
```

- Alternatives:

- Using `Array.newInstance`

```
import java.lang.reflect.Array;
public class Stiva <E>{
    private E[] elems;
    private int top;
    @SuppressWarnings("unchecked")
    public Stiva(Class<E> tip) {
        elems= (E[])Array.newInstance(tip, 10);
        top=0;
    }
    //...
}
Stiva<Integer> si=new Stiva<Integer>(Integer.class);
```

- Using `ArrayList` instead of array.

Generic arrays

- Use an array of Object, but read operation requires an explicit cast:

```
public class Stiva <E>{
    private Object[] elems;
    private int top;
    public Stiva() {
        elems=new Object[10];
        top=0;
    }
    public void push(E elem){
        elems[top++]=elem;
    }
    @SuppressWarnings("unchecked")
    public E pop(){
        if (top>0)
            return (E)elems[--top];
        return null;
    }
    //...
}
```


Erasure

- Java does not create a new class for each new instantiation of the type variables in case of the generic classes.
- The compiler erases all type variables and replaces them with their upper bounds (usually Object) and explicit casts are inserted when it is necessary

```
public class A {  
    public String f (Integer ix){  
        Stiva<String> st=new Stiva<String>();  
        Stiva sts=st;  
        sts.push(ix);  
        return st.top();  
    }  
}
```

```
public class A {  
    public String f (Integer ix){  
        Stiva st=new Stiva();  
        Stiva sts=st;  
        sts.push(ix);  
        return (String)st.top();  
    }  
}
```

compilation



- Reason: backward compatibility with the non-generic Java versions
- The generic class is not recompiled for each new instantiation of the type variables like in C++.

Bounds

```
public class ListOrd<E> {
    private class Nod<E>{
        E info;
        Nod<E> nxt;
        public Nod(){ info=null; nxt=null; }
        private Nod(E info, Nod<E> nxt) { this.info = info; this.nxt = nxt; }
        private Nod(E info) { this.info = info; nxt=null; }
    }
    private Nod<E> head;
    public ListOrd(){ head=null;}
    public void add(E elem){
        if (head==null){
            head=new Nod<E>(elem);
            return;
        }
        if (/*compare elem to head.info*/){
            head=new Nod<E>(elem,head);
        }else {...}
    }
}
```

Bounds

- Type variables can have constraints (namely bounds) using **extends**.

T extends E //T is the type E or is a subtype of E.

- General form of the constraint:

T extends [C &] I₁ [& I₂ &...& I_n]

T inherits the class C and implements the interfaces I₁, ... I_n.

- If T has constraints then through T we can call any method from the class and interfaces specified as bounds.

Bounds

```
public interface Comparable<E>{
    int compareTo(E e);
}

public class ListOrd<E extends Comparable<E>> {
    private class Nod<E>{...}
    private Nod<E> head;
    public ListOrd(){ head=null;}
    public void add(E elem){
        if (head==null){
            head=new Nod<E>(elem);
            return;
        }
        if (elem.compareTo(head.info)<0){
            head=new Nod<E>(elem,head);
        }else {...}
    }
    public E retElemPoz(int poz){
        //...
    }
}
```

Wildcards

```
ListOrd<String> ls=new ListOrd<String>();  
ListOrd<Object> lo=ls; //ASSUME this is CORRECT  
lo.add(23);  
String s=ls.retElemPoz(0); //ERROR
```

Obs:

If **SB** is a subtype of **T** and **G** is a generic container class then **G<SB>** is not a subtype of **G<T>**.

```
void printLista(ListOrd<Object> lo){  
    for(Object o:lo)  
        System.out.println(o);  
}  
...  
ListOrd<String> ls=new ListOrd<String>();  
ls.add("mere");  
ls.add("pere");  
printLista(ls);           //error at compile-time
```

Wildcards

We use ? to denote any type (or unknown type)

```
void printLista(ListOrd<?> lo) {  
    for(Object o:lo)  
        System.out.println(o);  
}
```

Obs:

1. When we use ?, the elements can be considered to be of type `Object` (upper bound).
2. When we use ? to declare an instance, the instance elements cannot be read or write, the only allowed operations are to read `Object` and to write `null`.

```
ListOrd<String> ls=new ListOrd<String>();  
ls.add("mere");  
ls.add("pere");  
ListOrd<?> ll=ls;  
ll.add("portocale"); //error  
ll.update(1, "struguri");//error  
Object el=ll.retElemPoz(0);
```

Bounded Wildcards

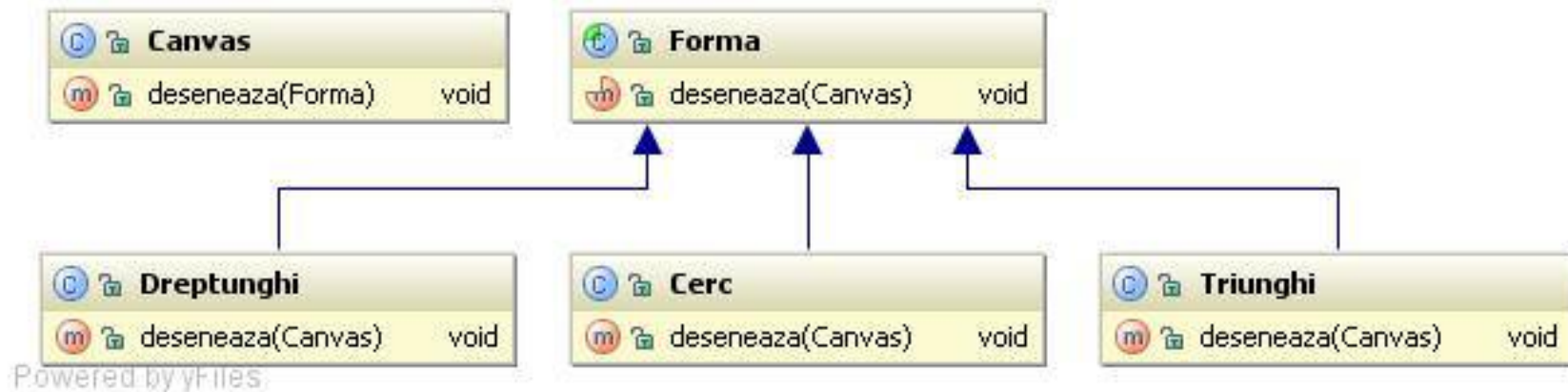
We can specify bounds for ?:

- Upper bound by `extends: ? extends C` or `? extends I`
- Lower bound by `super: ? super C` (any superclass of C)

1. Upper bound means that we can read elements of the type (or of superclass of the type) given by the upper bound.
2. Lower bound means that we can write elements of type (or of subclasses of the type) given by the lower bound.

```
ListOrd<Angajat> la=new ListOrd<Angajat>();  
la.add(new Angajat(...));  
ListOrd<? extends Persoana> lp=la;  
lp.add(new Angajat(...)); //error at compile time  
Persoana p=lp.retElemPoz(0);  
lp.retElemPoz(0).getNume();  
ListOrd<? super Angajat> linf=la;  
linf.add(new Angajat(...)); //correct
```

Bounded Wildcards



```
public class Canvas {  
    public void deseneaza(Forma f){ f.deseneaza(this); }  
    public void deseneaza(ListOrd<Forma> lf){  
        for(Forma f: lf)  
            f.deseneaza(this);  
    }  
}  
//...  
Canvas c=new Canvas();  
ListOrd<Cerc> lc=new ListOrd<Cerc>();  
c.deseneaza(lc);    //error at compile time
```


Bounded Wildcards

```
public class Canvas {  
    public void deseneaza(Forma f){ f.deseneaza(this); }  
    public void deseneaza(ListOrd<? extends Forma> lf){  
        for(Forma f: lf)  
            f.deseneaza(this);  
    }  
}  
//...  
Canvas c=new Canvas();  
ListOrd<Cerc> lc=new ListOrd<Cerc>();  
c.deseneaza(lc); //correct  
ListOrd<? extends Forma> ll=lc;  
ll.add(new Cerc()); //error at compile time;
```

JAVA COLLECTIONS

Java Collections Framework (JCF)

A *collection* is an object that maintains references to other objects
JCF is part of the `java.util` package and provides:

Interfaces

- Each defines the operations and contracts for a particular type of collection (List, Set, Queue, etc)
- Idea: when using a collection object, it's sufficient to know its interface

Implementations

- Reusable classes that implement above interfaces (e.g. LinkedList, HashSet)

Algorithms

- Useful polymorphic methods for manipulating and creating objects whose classes implement collection interfaces
- Sorting, index searching, reversing, replacing etc.

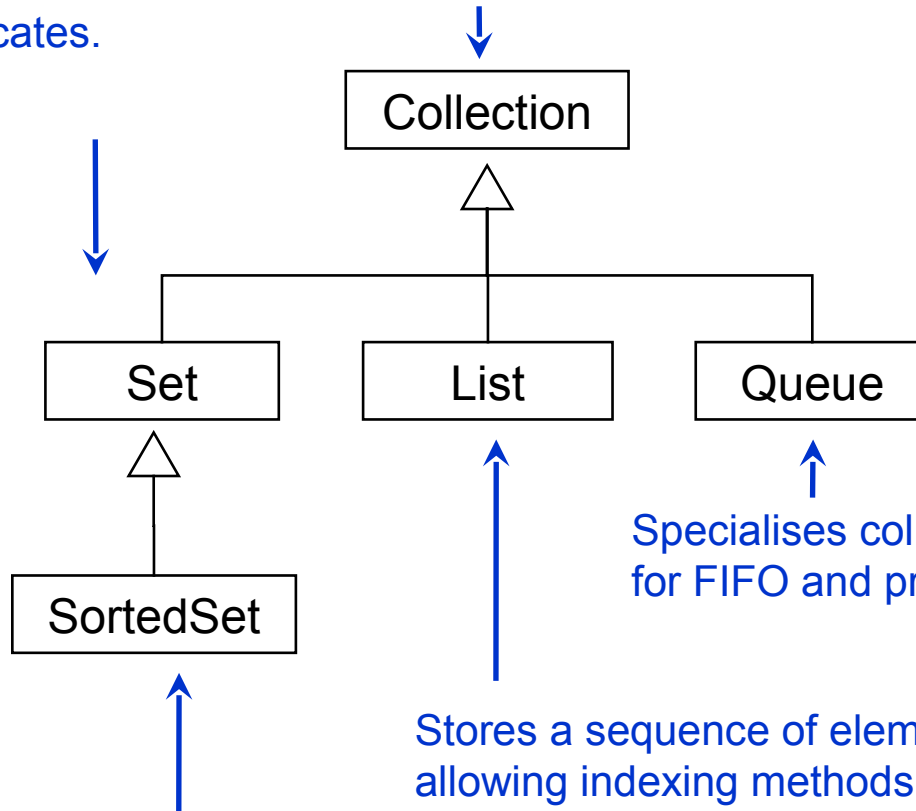
Interfaces

Generalisation

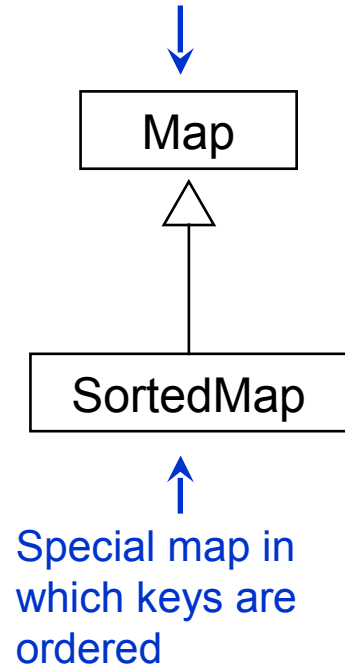
A special Collection that cannot contain duplicates.

Root interface for operations common to all types of collections

Stores mappings from keys to values



Specialises collection with operations for FIFO and priority queues.



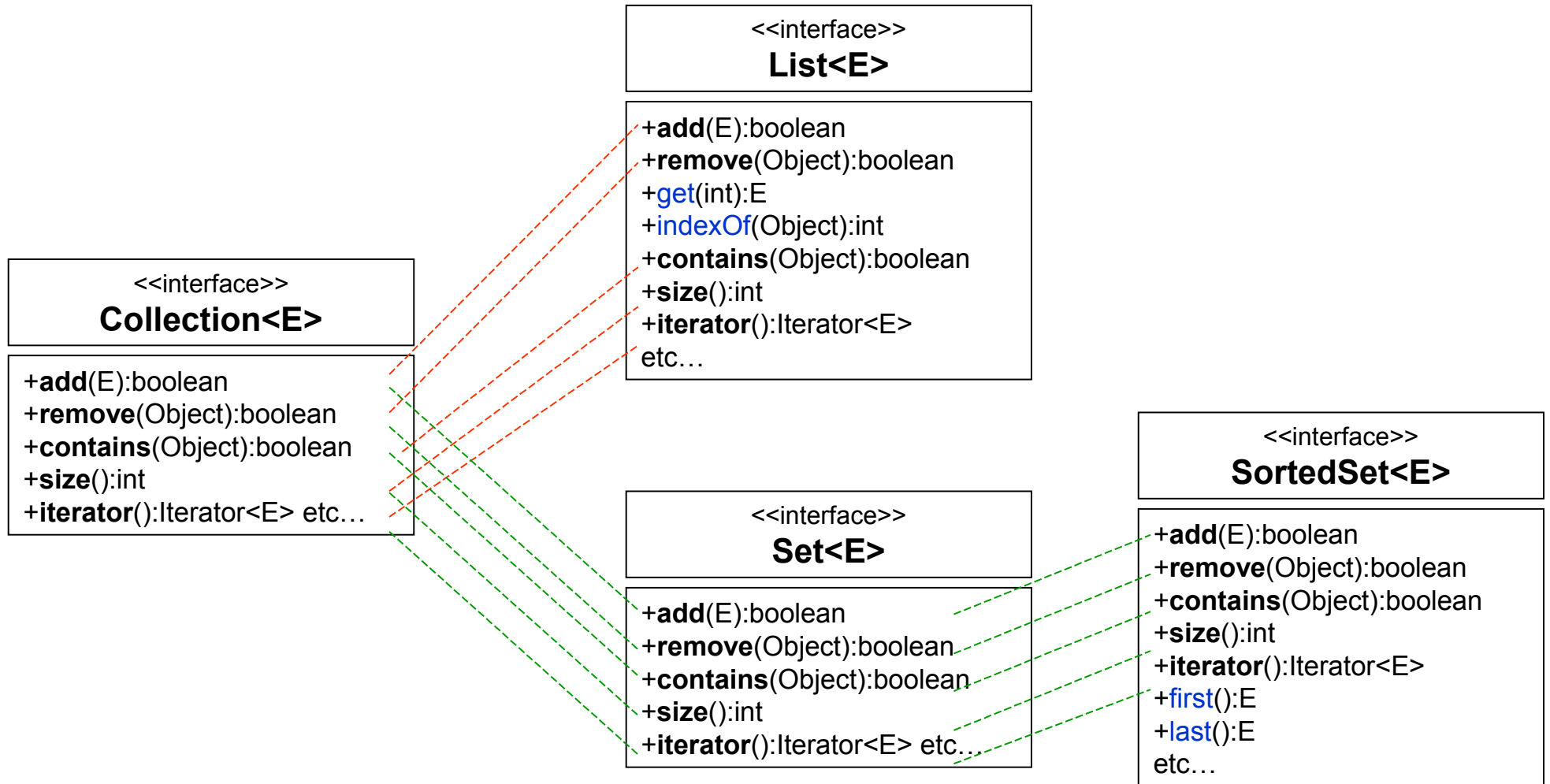
Special Set that retains ordering of elements.

Stores a sequence of elements, allowing indexing methods

Special map in which keys are ordered

Specialisation

Expansion of contracts



The Collection Interface

- The Collection interface provides the basis for List-like collections in Java. The interface includes:

```
boolean add(Object)
boolean addAll(Collection)
void clear()
boolean contains(Object)
boolean containsAll(Collection)
boolean equals(Object)
boolean isEmpty()
Iterator iterator()
boolean remove(Object)
boolean removeAll(Collection)
boolean retainAll(Collection)
int size()
Object[] toArray()
Object[] toArray(Object[])
```

List Interface

- Lists allow duplicate entries within the collection
- Lists are an ordered collection much like an array
 - Lists grow automatically when needed
 - The list interface provides accessor methods based on index
- The List interface extends the Collections interface and add the following method definitions:

```
void add(int index, Object)
boolean addAll(int index, Collection)
Object get(int index)
int indexOf(Object)
int lastIndexOf(Object)
ListIterator listIterator()
ListIterator listIterator(int index)
Object remove(int index)
Object set(int index, Object)
List subList(int fromIndex, int toIndex)
```

Set Interface

- The Set interface also extends the Collection interface but does not add any methods to it.
- Collection classes which implement the Set interface have the add stipulation that Sets CANNOT contain duplicate elements
- Elements are compared using the equals method
- NOTE: exercise caution when placing mutable objects within a set. Objects are tested for equality upon addition to the set. If the object is changed after being added to the set, the rules of duplication may be violated.

SortedSet Interface

- SortedSet provides the same mechanisms as the Set interface, except that SortedSets maintain the elements in ascending order.
- Ordering is based on natural ordering (Comparable) or by using a Comparator.

java.util.Iterator<E>

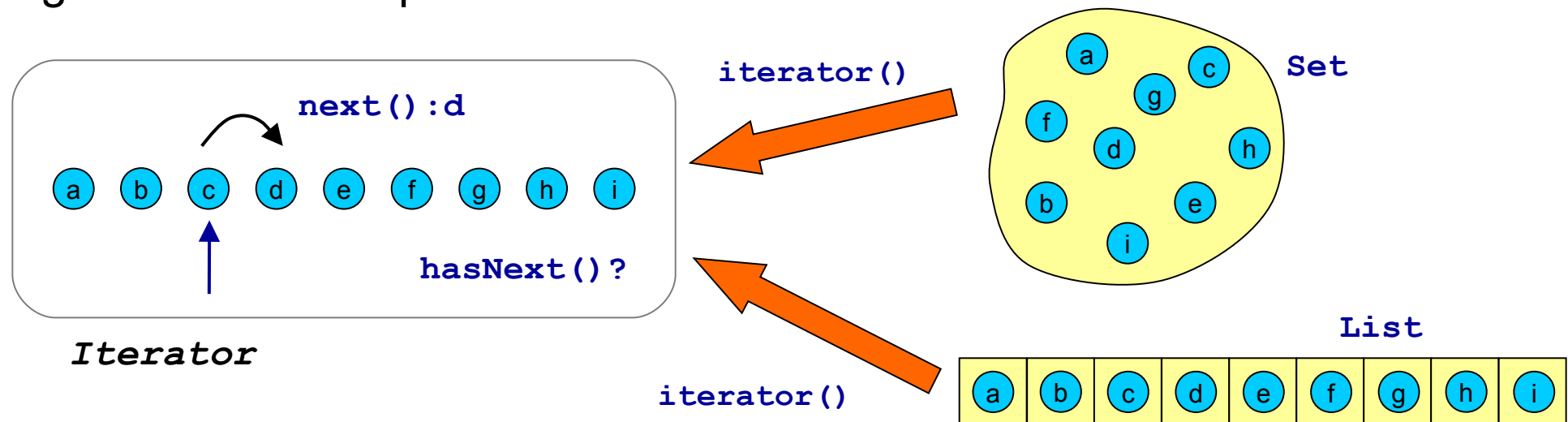
Think about typical usage scenarios for Collections

Retrieve the list of all patients

Search for the lowest priced item

More often than not you would have to traverse every element in the collection – be it a List, Set, or your own datastructure

Iterators provide a generic way to traverse through a collection regardless of its implementation



Using an Iterator

Quintessential code snippet for collection iteration:

```
public void list(Collection<T> items) {  
    Iterator<T> it = items.iterator();  
    while(it.hasNext()) {  
        T item = it.next();  
        System.out.println(item.toString());  
    }  
}
```

<<interface>>

Iterator<E>

+hasNext():boolean

+next():E

+remove():void

Design notes:

- Above method takes in an object whose class implements Collection
 - List, ArrayList, LinkedList, Set, HashSet, TreeSet, Queue, MyOwnCollection, etc
- We know any such object can return an Iterator through method iterator()
- We don't know the exact implementation of Iterator we are getting, but **we don't care**, as long as it provides the methods next() and hasNext()
- Good practice: **Program to an interface!**

java.lang.Iterable<T>

```
for (Item item : items) {  
    System.out.println(item);  
}
```

=

```
Iterator<Item> it = items.iterator();  
while(it.hasNext()) {  
    Item item = it.next();  
    System.out.println(item);  
}
```

This is called a “**for-each**” statement

For each `item` in `items`

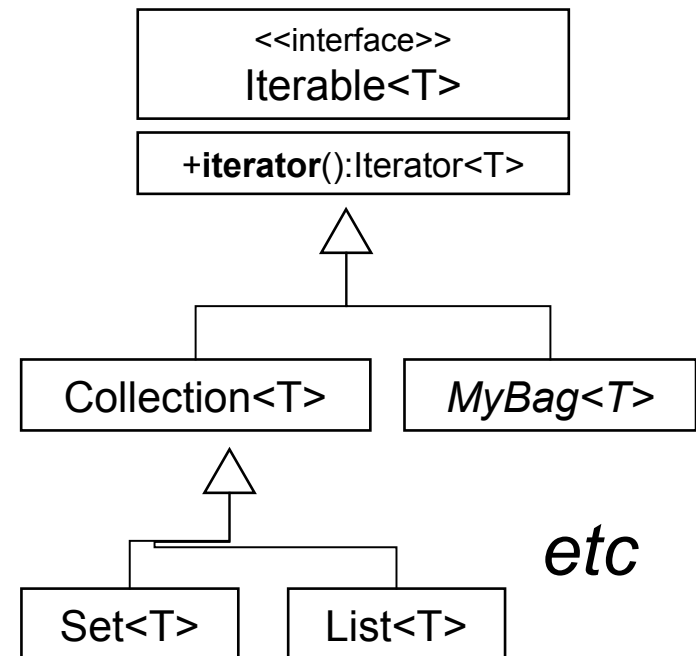
This is possible as long as `items` is of type **Iterable**

-Defines single method `iterator()`

Collection (and hence all its subinterfaces) implements **Iterable**

You can do this to your own implementation of **Iterable** too!

To do this you may need to return your own implementation of **Iterator**



java.util.Collections

Offers many very useful utilities and algorithms for manipulating and creating collections

Sorting lists

Index searching

Finding min/max

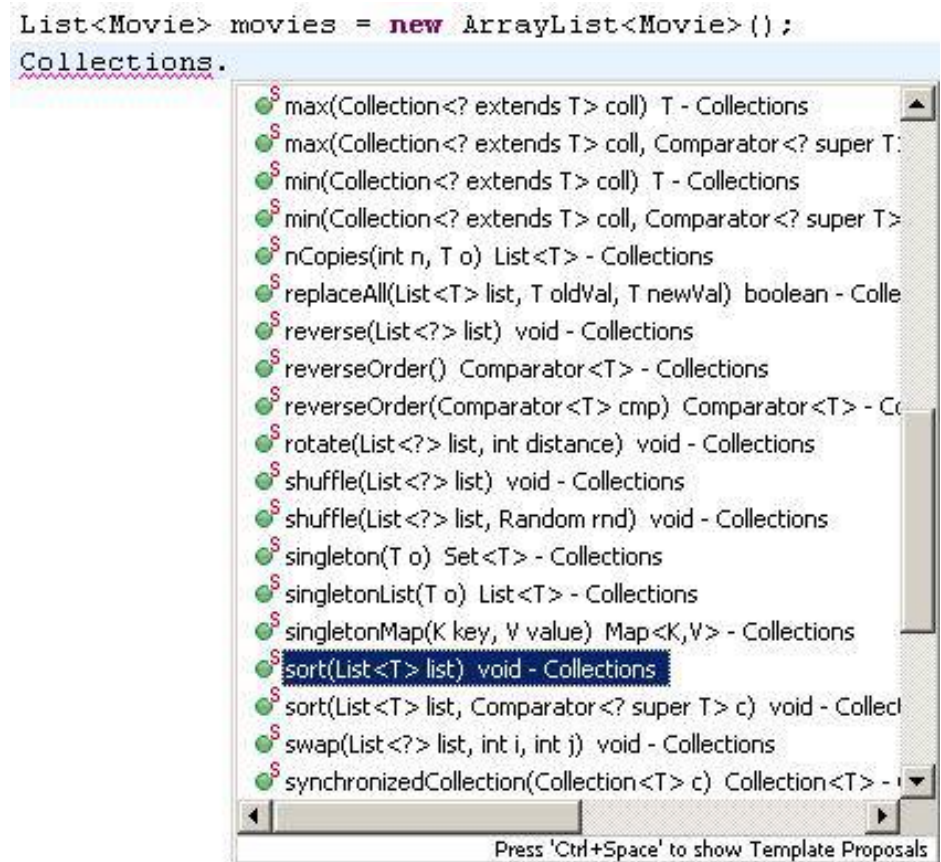
Reversing elements of a list

Swapping elements of a list

Replacing elements in a list

Other nifty tricks

Saves you having to implement them yourself → **reuse**

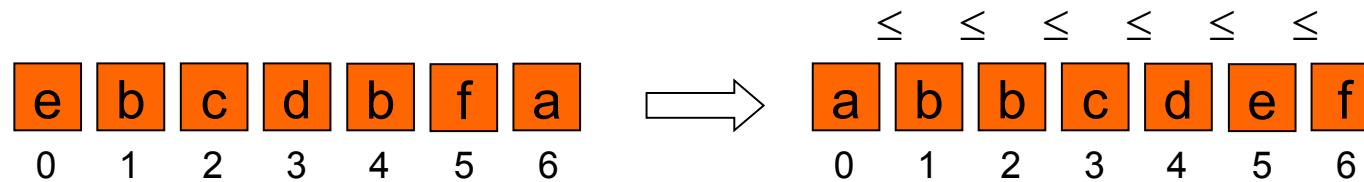


Comparable and Comparators

- You will have noted that some classes provide the ability to sort elements.
 - How is this possible when the collection is supposed to be de-coupled from the data?
- Java defines two ways of comparing objects:
 - The objects implement the Comparable interface
 - A Comparator object is used to compare the two objects
- If the objects in question are Comparable, they are said to be sorted by their "natural" order.
- Comparable object can only offer one form of sorting. To provide multiple forms of sorting, Comparators must be used.

Collections.sort()

Java's implementation of merge sort – ascending order



- What types of objects can you sort? Anything that has an **ordering**
- Two sort() methods: sort a given List according to either 1) *natural ordering* of elements or an 2) externally defined ordering.

1) `public static <T extends Comparable<? super T>> void sort(List<T> list)`

2) `public static <T> void sort(List<T> list, Comparator<? super T> c)`

■ Translation:

1. Only accepts a List parameterised with type implementing `Comparable`
2. Accepts a List parameterised with any type as long as you also give it a `Comparator` implementation that defines the ordering for that type

java.lang.Comparable<T>

A **generic interface** with a single method: `int compareTo(T)`

Return 0 if this = other

Return **any +’ve integer** if this > other

Return **any –’ve integer** if this < other

Implement this interface to define **natural ordering** on objects of type T

```
public class Money implements Comparable<Money> {  
    ...  
    public int compareTo( Money other ) {  
        if( this.cents == other.cents ) {  
            return 0;  
        }  
        else if( this.cents < other.cents ) {  
            return -1;  
        }  
        else {  
            return 1;  
        }  
    }  
}
```

```
m1 = new Money(100,0);  
m2 = new Money(50,0);  
m1.compareTo(m2) returns 1;
```

A more concise way of doing this? (hint: 1 line)

```
return this.cents - other.cents;
```


Natural-order sorting

```
List<Money> funds = new ArrayList<Money>();  
funds.add(new Money(100,0));  
funds.add(new Money(5,50));  
funds.add(new Money(-40,0));  
funds.add(new Money(5,50));  
funds.add(new Money(30,0));
```

```
Collections.sort(funds);  
System.out.println(funds);
```

```
List<CD> albums = new ArrayList<CD>();  
albums.add(new CD("Street Signs","Ozomatli",2.80));  
//etc...  
Collections.sort(albums);
```

What's the output?
[-40.0,
5.50,
5.50,
30.0,
100.0]

CD does not implement a
Comparable interface

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

java.util.Comparator<T>

Useful if the type of elements to be sorted is not Comparable, or you want to define an alternative ordering

Also a generic interface that defines methods **compare(T, T)** and **equals(Object)**

Usually only need to define **compare(T, T)**

Define ordering by CD's getPrice() → **Money**

Note: PriceComparator implements a Comparator parameterised with CD → T “becomes” CD

<<interface>> Comparator<T>
+compare(T o1, T o2):int +equals(Object other):boolean

CD
+getTitle():String +getArtist():String +getPrice():Money

```
public class PriceComparator
implements Comparator<CD> {
    public int compare(CD c1, CD c2) {
        return c1.getPrice().compareTo(c2.getPrice());
    }
}
```

Comparator and Comparable
going hand in hand 😊

Comparator sorting

```
List<CD> albums = new ArrayList<CD>();  
albums.add(new CD("Street Signs", "Ozomatli", new Money(3, 50)));  
albums.add(new CD("Jazzinho", "Jazzinho", new Money(2, 80)));  
albums.add(new CD("Space Cowboy", "Jamiroquai", new Money(5, 00)));  
albums.add(new CD("Maiden Voyage", "Herbie Hancock", new Money(4, 00)));  
albums.add(new CD("Here's the Deal", "Liquid Soul", new Money(1, 00)));  
  
Collections.sort(albums, new PriceComparator());  
System.out.println(albums);
```

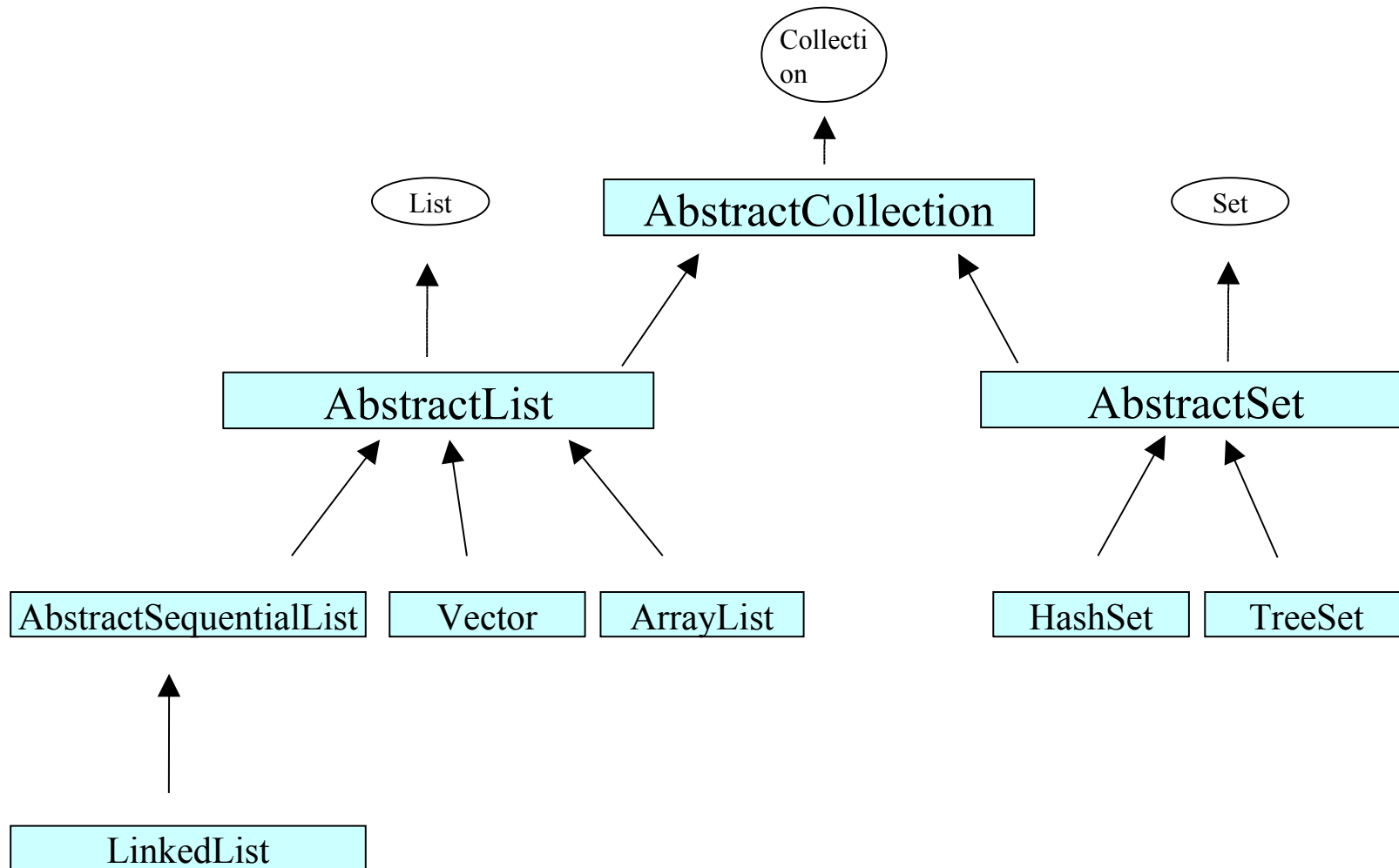
implements `Comparator<CD>`

Note, in `sort()`, `Comparator` overrides natural ordering
i.e. Even if we define natural ordering for `CD`, the given
comparator is still going to be used instead
(On the other hand, if you give `null` as `Comparator`, then
natural ordering is used)

```
public static <T> void sort(List<T> list, Comparator<? super T> c)
```

The Class Structure

- The Collection interface is implemented by a class called AbstractCollection. Most collections inherit from this class.



Lists

- Java provides 3 concrete classes which implement the list interface
 - Vector
 - ArrayList
 - LinkedList
- Vectors try to optimize storage requirements by growing and shrinking as required
 - Methods are synchronized (used for Multi threading)
- ArrayList is roughly equivalent to Vector except that its methods are not synchronized
- LinkedList implements a doubly linked list of elements
 - Methods are not synchronized

Sets

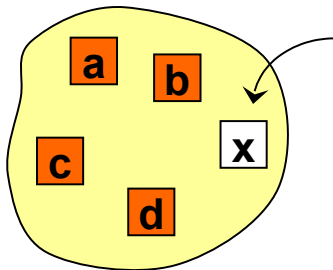
- Java provides 2 concrete classes which implement the Set interface
 - HashSet
 - TreeSet
- HashSet behaves like a HashMap except that the elements cannot be duplicated.
- TreeSet behaves like TreeMap except that the elements cannot be duplicated.
- Note: Sets are not as commonly used as Lists

Set<E>

Mathematical Set abstraction – contains **no duplicate** elements
i.e. no two elements e_1 and e_2 such that $e_1.equals(e_2)$

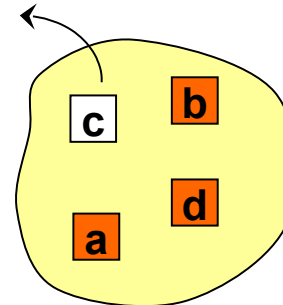
add(x)
→ *true*

add(b)
→ *false*



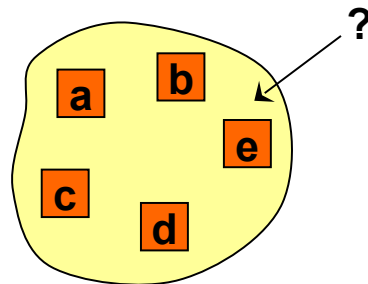
remove(c)
→ *true*

remove(x)
→ *false*



contains(e)
→ *true*

contains(x)
→ *false*

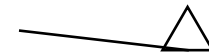


isEmpty()
→ *false*

size()
→ 5

<<interface>>
Set<E>

+**add**(E):boolean
+**remove**(Object):boolean
+**contains**(Object):boolean
+**isEmpty**():boolean
+**size**():int
+**iterator**():Iterator<E> etc...



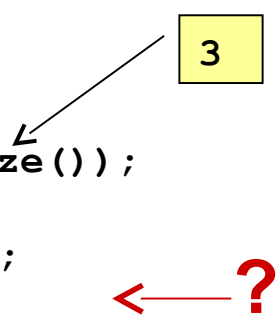
<<interface>>
SortedSet<E>

+**first**():E
+**last**():E
etc...

HashSet<E>

- Typically used implementation of Set.
- Parameterise Sets just as you parameterise Lists
- Efficient (constant time) insert, removal and contains check – all done through hashing
- x and y are duplicates if x.equals(y)
- How are elements ordered? Quiz:

```
Set<String> words = new HashSet<String>();  
words.add("Bats");  
words.add("Ants");  
words.add("Crabs");  
words.add("Ants");  
System.out.println(words.size());  
for (String word : words) {  
    System.out.println(word);  
}
```



<<interface>>
Set<E>

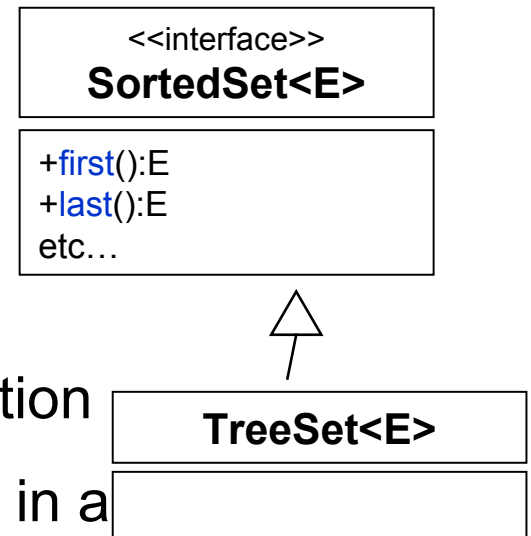
+**add**(E):boolean
+**remove**(Object):boolean
+**contains**(Object):boolean
+**size**():int
+**iterator**():Iterator<E> etc...


HashSet<E>

- a) Bats, Ants, Crabs
b) Ants, Bats, Crabs
c) Crabs, Bats, Ants
d) Nondeterministic

TreeSet<E> (SortedSet<E>)

- If you want an ordered set, use an implementation of a SortedSet: TreeSet
- What's up with "Tree"? Red-black tree
- Guarantees that all elements are ordered (sorted) at all times
 - » **add()** and **remove()** preserve this condition
 - » **iterator()** always returns the elements in a specified order
- Two ways of specifying ordering
 - » Ensuring elements have natural ordering (**Comparable**)
 - » Giving a **Comparator<E>** to the constructor
- **Caution:** TreeSet considers x and y are duplicates if `x.compareTo(y) == 0` (or `compare(x,y) == 0`)



TreeSet construction

```
Set<String> words = new TreeSet<String>();  
words.add("Bats");  
words.add("Ants");  
words.add("Crabs");  
for (String word : words) {  
    System.out.println(word);  
}
```

String has a **natural ordering**, so empty constructor

What's the output?

Ants; Bats; Crabs

- But CD doesn't, so you must pass in a Comparator to the constructor

```
Set<CD> albums = new TreeSet<CD>(new PriceComparator());  
albums.add(new CD("Street Signs", "O", new Money(3, 50)));  
albums.add(new CD("Jazzinho", "J", new Money(2, 80)));  
albums.add(new CD("Space Cowboy", "J", new Money(5, 00)));  
albums.add(new CD("Maiden Voyage", "HH", new Money(4, 00)));  
albums.add(new CD("Here's the Deal", "LS", new Money(2, 80)));  
System.out.println(albums.size());  
for (CD album : albums) {  
    System.out.println(album);  
}
```

What's the output?

4

Jazzinho; Street; Maiden; Space

The Map Interface

- The Map interface provides the basis for dictionary or key-based collections in Java. The interface includes:

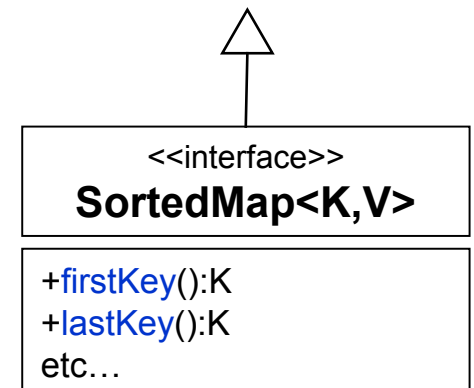
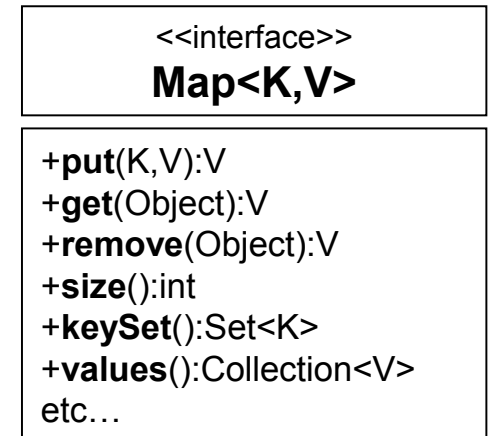
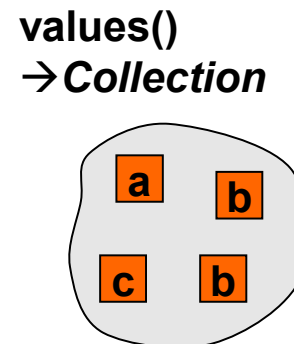
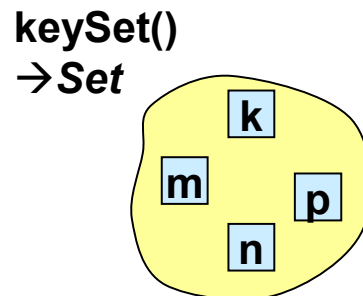
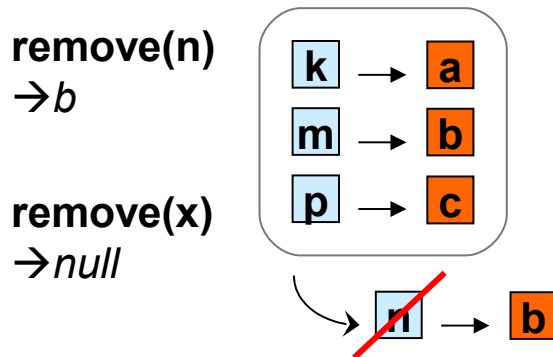
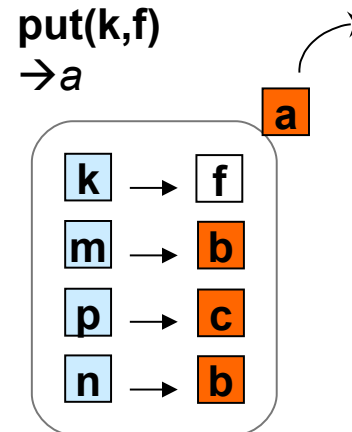
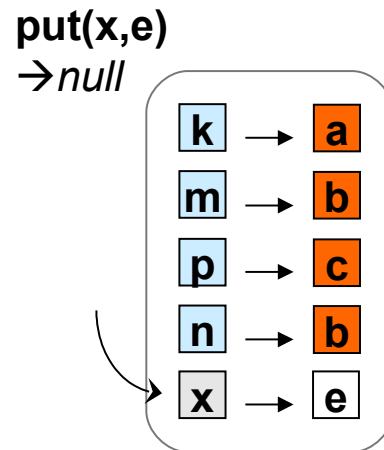
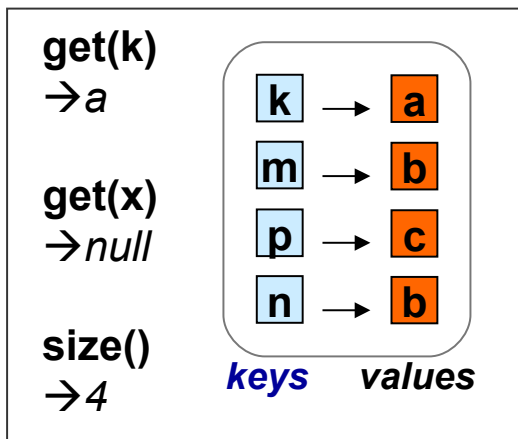
```
void clear()
boolean containsKey(Object)
boolean containsValue(Object)
Set entrySet()
boolean equals(Object)
Object get(Object)
boolean isEmpty()
Set keySet()
Object put(Object key, Object value)
void putAll(Map)
boolean remove(Object key)
int size()
Collection values()
```

Maps

- Java provides 3 concrete classes which implement the map interface
 - HashMap
 - WeakHashMap
 - TreeMap
- HashMap is the most commonly used Map.
 - Provides access to elements through a key.
 - The keys can be iterated if they are not known.
- WeakHashMap provides the same functionality as Map except that if the key object is no longer used, the key and its value will be removed from the Map.
- A Red-Black implementation of the Map interface

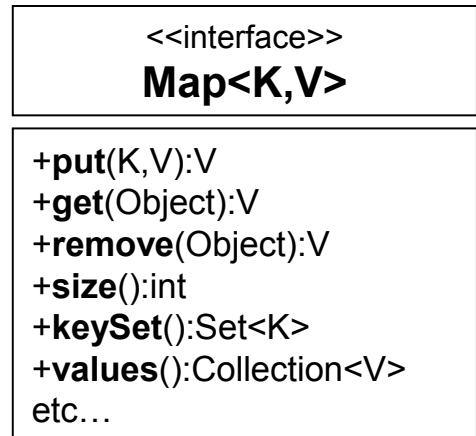
Map<K, V>

- Stores mappings from (unique) keys (type \mathbb{K}) to values (type \mathbb{V})
 - » See, you can have more than one type parameters!
- Think of them as “arrays” but with objects (keys) as indexes
 - » Or as “directories”: e.g. "Bob" → 021999887



HashMap<K, V>

- keys are hashed using `Object.hashCode()`
 - » i.e. no guaranteed ordering of keys
- `keySet()` returns a `HashSet`
- `values()` returns a `Collection`



```
Map<String, Integer> directory
    = new HashMap<String, Integer>();
directory.put("Mum", new Integer(9998888));
directory.put("Dad", 9998888);
directory.put("Bob", 12345678);
directory.put("Edward", 5553535);
directory.put("Bob", 1000000);
System.out.println(directory.size());
for (String key : directory.keySet()) {
    System.out.print(key+"'s number: ");
    System.out.println(directory.get(key));
}
System.out.println(directory.values());
```

"autoboxing"

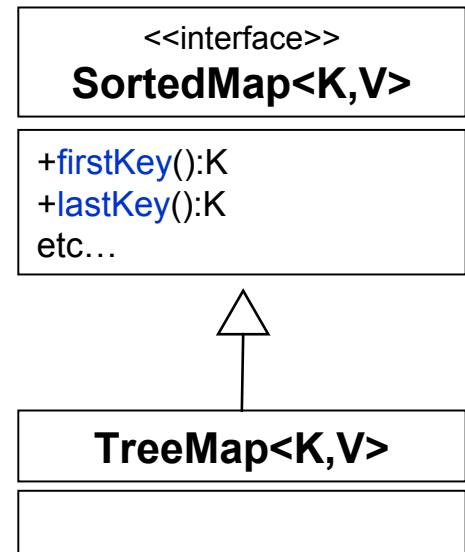
4 or 5?

Set<String>

What's Bob's number?

TreeMap<K, V>

- Guaranteed ordering of keys (like TreeSet)
 - » In fact, TreeSet is implemented using TreeMap ☺
 - » Hence `keySet()` returns a `TreeSet`
- `values()` returns a Collection – ordering depends on ordering of keys



```
Map<String, Integer> directory
    = new TreeMap<String, Integer>();
directory.put("Mum", new Integer(9998888));
directory.put("Dad", 9998888);
directory.put("Bob", 12345678);
directory.put("Edward", 5553535);
directory.put("Bob", 1000000);
System.out.println(directory.size());
for (String key : directory.keySet()) {
    System.out.print(key+"'s #: ");
    System.out.println(directory.get(key));
}
System.out.println(directory.values());
```

Empty constructor
→ natural ordering

4

Loop output?

```
Bob's #: 1000000
Dad's #: 9998888
Edward's #: 5553535
Mum's #: 9998888
```

?

TreeMap with Comparator

As with TreeSet, another way of constructing TreeMap is to give a Comparator → necessary for non-Comparable keys

```
Map<CD, Double> ratings
    = new TreeMap<CD, Double>(new PriceComparator());
ratings.put(new CD("Street Signs", "O", new Money(3, 50)), 8.5);
ratings.put(new CD("Jazzinho", "J", new Money(2, 80)), 8.0);
ratings.put(new CD("Space Cowboy", "J", new Money(5, 00)), 9.0);
ratings.put(new CD("Maiden Voyage", "H", new Money(4, 00)), 9.5);
ratings.put(new CD("Here's the Deal", "LS", new Money(2, 80)), 9.0);

System.out.println(ratings.size());
for (CD key : ratings.keySet()) {
    System.out.print("Rating for "+key+": ");
    System.out.println(ratings.get(key));
}
System.out.println("Ratings: "+ratings.values());
```

4

Ordered by key's price

Depends on key ordering

Most Commonly Use Methods

- While it is a good idea to learn and understand all of the methods defined within this infrastructure, here are some of the most commonly used methods.
- For Lists:
 - `add(Object)`, `add(index, Object)`
 - `get(index)`
 - `set(index, Object)`
 - `remove(Object)`
- For Maps:
 - `put(Object key, Object value)`
 - `get(Object key)`
 - `remove(Object key)`
 - `keySet()`

Which class should I use?

- You'll notice that collection classes all provide the same or similar functionality. The difference between the different classes is how the structure is implemented.
 - This generally has an impact on performance.
- Use Vector
 - Fast access to elements using index
 - Optimized for storage space
 - Not optimized for inserts and deletes
- Use ArrayList
 - Same as Vector except the methods are not synchronized. Better performance
- Use linked list
 - Fast inserts and deletes
 - Stacks and Queues (accessing elements near the beginning or end)
 - Not optimized for random access

Which class should I use?

- Use Sets
 - When you need a collection which does not allow duplicate entries
- Use Maps
 - Very Fast access to elements using keys
 - Fast addition and removal of elements
 - No duplicate keys allowed
- When choosing a class, it is worthwhile to read the class's documentation in the Java API specification. There you will find notes about the implementation of the Collection class and within which contexts it is best to use.

Collections and Fundamental Data Types

- Note that collections can only hold Objects.
 - One cannot put a fundamental data type into a Collection
- Java has defined "wrapper" classes which hold fundamental data type values within an Object
 - These classes are defined in java.lang
 - Each fundamental data type is represented by a wrapper class
- The wrapper classes are:
 - Boolean
 - Byte
 - Character
 - Double
 - Float
 - Short
 - Integer
 - Long

Wrapper Classes

- The wrapper classes are usually used so that fundamental data values can be placed within a collection
- The wrapper classes have useful class variables.
 - `Integer.MAX_VALUE`, `Integer.MIN_VALUE`
 - `Double.MAX_VALUE`, `Double.MIN_VALUE`, `Double.NaN`, `Double.NEGATIVE_INFINITY`, `Double.POSITIVE_INFINITY`
- They also have useful class methods
 - `Double.parseDouble(String)` - converts a `String` to a `double`
 - `Integer.parseInt(String)` - converts a `String` to an `integer`

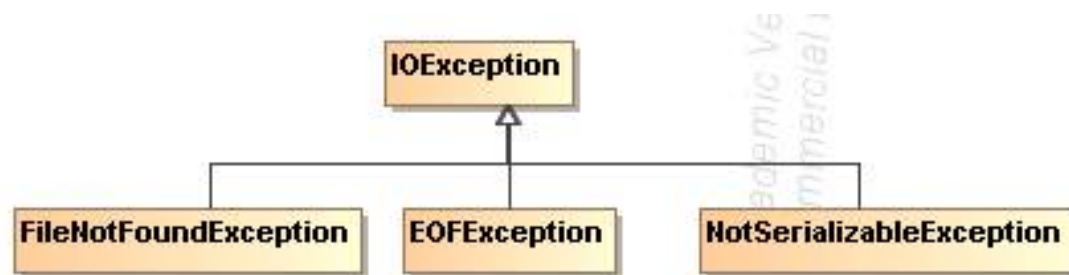
Java IO

Java.IO

■ Package java.io

- classes working on bytes (InputStream, OutputStream)
- Classes working on chars (Reader, Writer)
- Byte-char conversion (InputStreamReader, OutputStreamWriter)
- Random access (RandomAccessFile)
- Scanner

■ Exceptions:

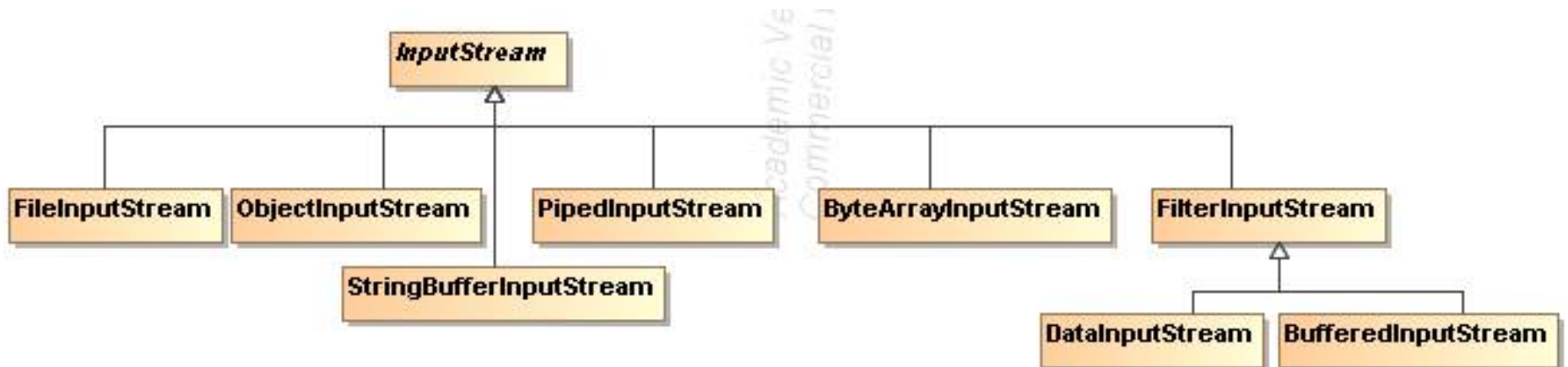


IO Stream

- represents an input source or an output destination
- is a sequence of data
- supports many different kinds of data, including simple bytes, primitive data types, localized characters, and objects.
- simply passes on data; others manipulate and transform the data in useful ways.

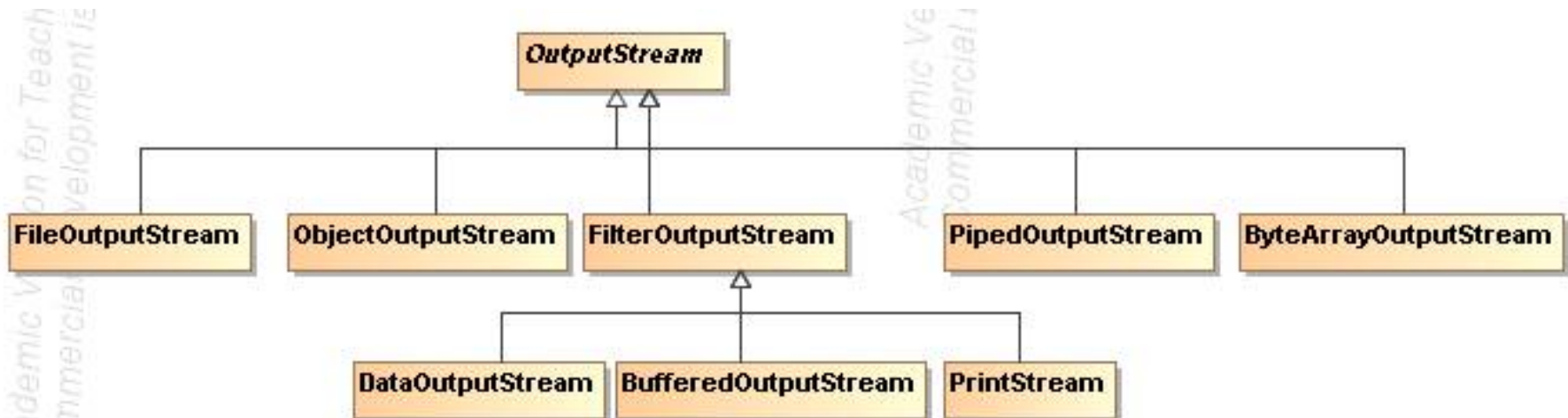
InputStream

- Abstract class that contains methods for reading bytes from a stream (file, memory, pipe, etc.)
 - `read():int` //read a byte, return -1 if no more bytes
 - `read(cbuff:byte[]): int` //read max `cbuff.length` bytes, return the nr of bytes that has been read, or -1
 - `read(buff:byte[], offset:int, length:int):int` //read max `length` bytes and write to `buff` starting with position `offset`, return the number of bytes that has been read, or -1
 - `available(): int` //number of bytes available for reading
 - `close()` //close the stream



OutputStream

- Abstract class that contains methods for writing bytes into a stream (file, memory, pipe, etc.)
 - `write(int)` //write a byte
 - `write(b:byte[])` //write `b.length` bytes from array `b` into the stream
 - `write(b:byte[], offset:int, len:int):int` //write `len` bytes from array `b` starting with the position `offset`
 - `flush()` // force the effective writing into the stream
 - `close()` //close the stream

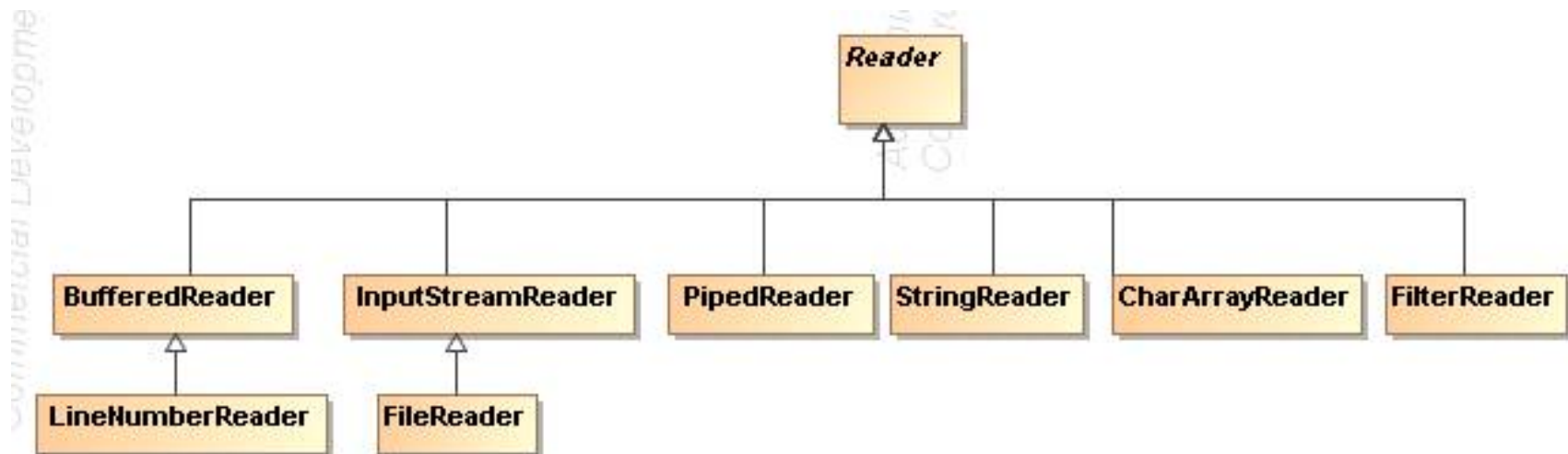


Example

```
FileInputStream in = null;
FileOutputStream out = null;
try {
    in = new FileInputStream("fisier.txt");
    out = new FileOutputStream("fisier2.txt");
    int c;
    while ((c = in.read()) != -1) {
        out.write(c);
    }
} catch(IOException e){
    System.err.println("Eroare "+e);
}finally {
    if (in != null)
        try {
            in.close();
        } catch (IOException e){ System.err.println("eroare "+e);}
    if (out != null)
        try {
            out.close();
        } catch (IOException e) { System.err.println("eroare "+e);}
}
```

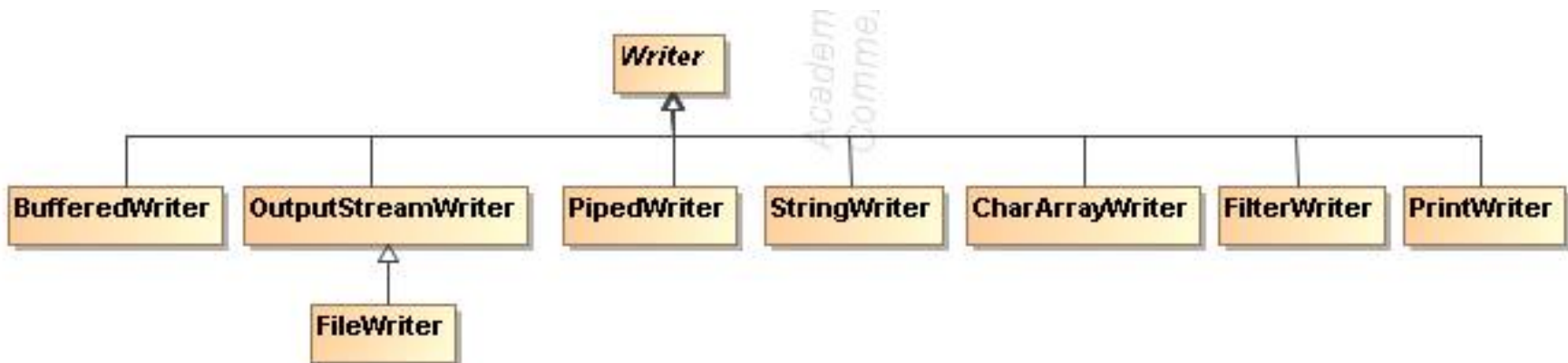
Reader

- Abstract class that contains methods for chars reading (1 char = 2 bytes) from a stream (file, memory, pipe, etc.)
 - `read():int` //read a char, return -1 for the end of the stream
 - `read(cbuff:char[]): int` //read max `cbuff.length` chars, return nr of read chars or -1
 - `read(buff:char[], offset:int, length:int):int` //read max `length` chars into array `buff` starting with offset `offset`, return the number of read chars or -1
 - `close()` //close the stream



Writer

- Abstract class that contains the methods for writing chars into a stream
 - `write(int)` //write a char
 - `write(b:char[])` //write `b.length` chars from array `b` into the stream
 - `write(b:char[], offset:int, len:int):int` //write `len` chars from array `b` starting with offset
 - `write(s:String)` //write a `String`
 - `write(s:String, off:int, len:int)` //write a part of a `String`
 - `flush()` // force the effective writing
 - `close()` //close the stream



Example

```
FileReader input = null;
FileWriter output = null;
try {
    input = new FileReader("Fisier.txt");
    output = new FileWriter("Fisier2out.txt");
    int c;
    while ((c = input.read()) != -1) output.write(c);
} catch (IOException e) {
    System.err.println("Eroare la citire/scriere"+e);
} finally {
    if (input != null)
        try {
            input.close();
        } catch (IOException e) {System.err.println("eroare "+e);}
    if (output != null)
        try {
            output.close();
        } catch (IOException e) {System.err.println("Eroare "+e);}
}
```

Classes

Operations	Byte	Char
Files	FileInputStream, FileOutputStream	FileReader, FileWriter
Memory	ByteArrayInputStream, ByteArrayOutputStream	CharArrayReader CharArrayWriter
Buffered Operations	BufferedInputStream BufferedOutputStream	BufferedReader BufferedWriter
Format	PrintStream	PrintWriter
Conversion Byte ↔ Char	InputStreamReader (byte -> char) OutputStreamWriter (char -> byte)	