# Advanced Programming Methods

## Lecture 3 – Java Generics and Collections

# Lecture Overview

1.  Enhanced For

2. Generics

3. Collections

# 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

# A non generic class

public class SingleBox {

    private **Object** first;

    public void setFirst(**Object** object) { this.first = object; }

    public **Object** getFirst() { return first; }}

**Class usage:**

SingleBox ob= new SingleBox();

Integer i = new Integer(10);

ob.setFirst(i);//information about Integer is lost


//we need a downcast to recover the specific information, namely the Integer

Integer a = **(Integer)** ob.getFirst();

//the cast may be wrong, it will pass the compiler but a runtime error will occur

String s = **(String)** ob.getFirst();

# First generic version

```
public class SingleBox<T> {

    // T stands for "Type"

    protected T first;

    public void setFirst(T t) { this.first = t; }

    public T getFirst() { return first; }}
```

**Class usage:**

```
SingleBox<Integer> ob= new SingleBox<Integer>();

Integer i = new Integer(10);

ob.setFirst(i);

Integer a = ob.getFirst();
```


**String s = ob.getFirst(); //an error will be generated at compile time**

# Inheritance Example

```
public class PairBox<T1, T2> extends SingleBox<T1> {

    private T2 second;

    public PairBox(T1 f, T2 s) {

        this.first = f;  this.second = s;

    }


    public T2 getSecond()     { return second; }

    public void setSecond(T2 t) { this.second=t; }

}
```

# 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.

# 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

```java
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

```java
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.

```java
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);
    }
}
```

# Raw Types

SingleBox<String> stringBox = new SingleBox<>();


Raw types is the non-generic class SingleBox

SingleBox rawBox = stringBox;                    // OK


But if you assign a raw type to a parameterized type, you get a warning:


SingleBox rawBox = new SingleBox();              // rawBox is a raw type of Box<T>

SingleBox<Integer> intBox = rawBox;      // warning: unchecked conversion

# Raw Types

You also get a warning if you use a raw type to invoke generic

methods defined in the corresponding generic type:


SingleBox<String> stringBox = new SingleBox<>();

SingleBox rawBox = stringBox;

rawBox.setFist(new Integer(8));  // warning: unchecked invocation to set(T)


**The warnings show that raw types bypass generic type checks, deferring the catch of unsafe code to runtime.**

**Therefore, you should avoid using raw types!!!!**

# 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:

```java
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;
    }
  //...
}
```

# Bounds

```java
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 &] I`$_1$ `[& I`$_2$ `&...& I`$_n$`]`

    T inherits the class C and implements the interfaces $I_1$, ... $I_n$.

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

# *Bounds*

```java
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){
        //...
    }
}
```

# Motivation for *Wildcards*

```
ListOrd<String> ls=new ListOrd<String>();
ListOrd<Object> lo=ls; //ASSUME this is CORRECT

lo.add(23);
String s=ls.retElemPoz(0);   //ERROR


Our assumption is wrong!!!
```

# Motivation for *Wildcards*

**The correct rule:**

If `SB` is a subclass (subtype) of `T` and `G` is a generic container class then `G<SB>` is not a subclass (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**

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);
```

# Wildcards

```
SingleBox<?>

- is the top of the hierarchy


SingleBox<?> wildbox= new SingleBox<Number>;

SingleBox<Integer> intbox= new SingleBox<Integer>;

wildbox = intbox; //OK

intbox.setFirst(new Integer(2)); //OK

wildbox.setFirst(new Integer(2)); //ERROR at compile time

Integer = intbox.getFirst(); //OK

Integer = wildbox.getFirst(); //ERROR at compile time

Object = wildbox.getFirst(); //OK


in fact SingleBox<?> means SingleBox<T> where bottom < T<Object
```

# *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);//OK
lp.retElemPoz(0).getNume();
ListOrd<? super Angajat> linf=la;
linf.add(new Angajat(...));  //correct
```

# Upper Bounds—Covariant Subtyping

- we can only read the content

```
SingleBox<? extends Integer> a = new SingleBox<Integer>();//OK

SingleBox<? extends Number>  b = a;  // OK.

SingleBox<? extends Number>  b =new SingleBox<Integer>();//OK

SingleBox<? extends Integer> a = new SingleBox<Number>();//ERROR
```

# *Upper Bounds—Covariant Subtyping*

Upper bound means that we can only read elements of the type (or of superclass of the type) given by the upper bound.

SingleBox<? extends Number> means SingleBox<T> where  bottom<T<Number

and we can read Number or a superclass of Number.

an example:

public Number processReadBox(Box<? extends Number> a) { return
    a.getFirst(); }

and can be called by proccessReadBox(v) where

v has the type Box<? extends Integer>,Integer is derived from Number
    or v can have the type Box<Integer>

# *Lower Bounds—Contravariant Subtyping*

```
SingleBox<? super Number> a = new SingleBox<Number>();//OK

SingleBox<? super Integer>  b = a;   // OK.

SingleBox<? super Number>  b =new SingleBox<Integer>();//ERROR

SingleBox<? super Integer> a = new SingleBox<Number>();//OK
```

# *Lower Bounds—Contravariant Subtyping*

Lower bound means that we can write elements of type (or  of subclasses of the type) given by the lower bound.

SingleBox<? super Integer> means SingleBox<T> where  Integer<T<Object

and we can write Integer or a subclass of Integer.


an example:

```
public static void proccessWriteBox(Box<? super Integer> a, Integer x)
   { a.setFirst(x); }
```


and can be called by proccessWriteBox(v) where

v has the type Box<? super Number>,Integer is derived from Number

or v can have the type Box<Integer>

# Wildcards

```
SingleBox<?>

- is the top of the hierarchy


SingleBox<?> wildbox;

SingleBox<? extends Number> nrbox;

SingleBox<? super Integer> intbox;

wildbox=nrbox; //OK

wildbox=intbox;//OK
```
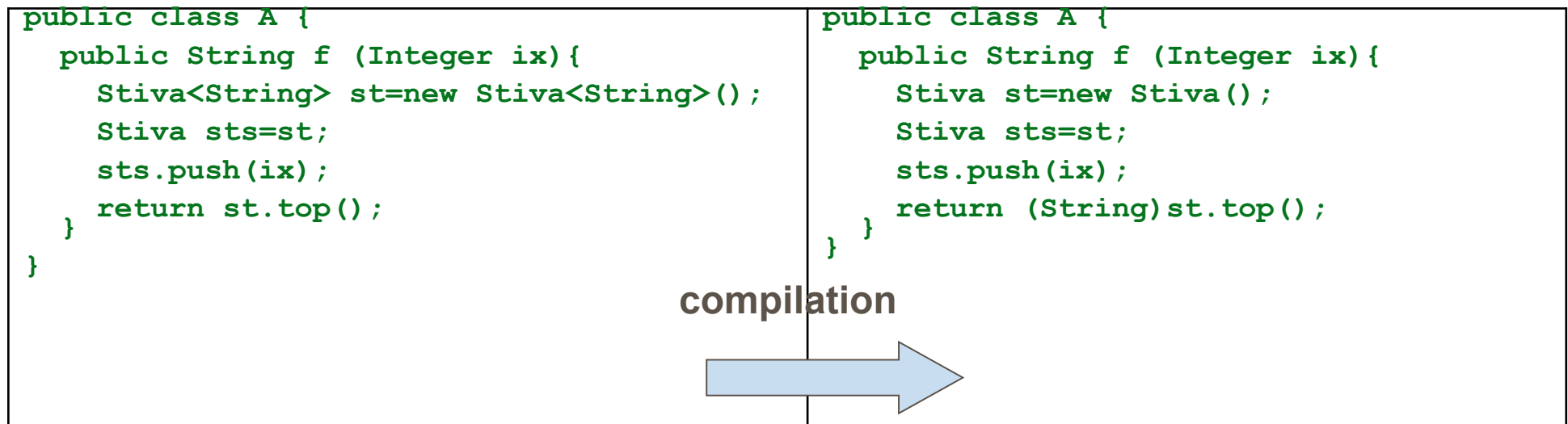
# *Erasure*

- 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++.

# JAVA COLLECTIONS

# Java Collections Framework (JCF)

A *collection* is an object that maintains references to others 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
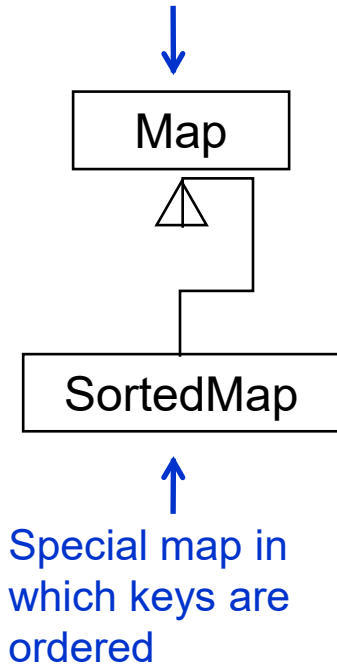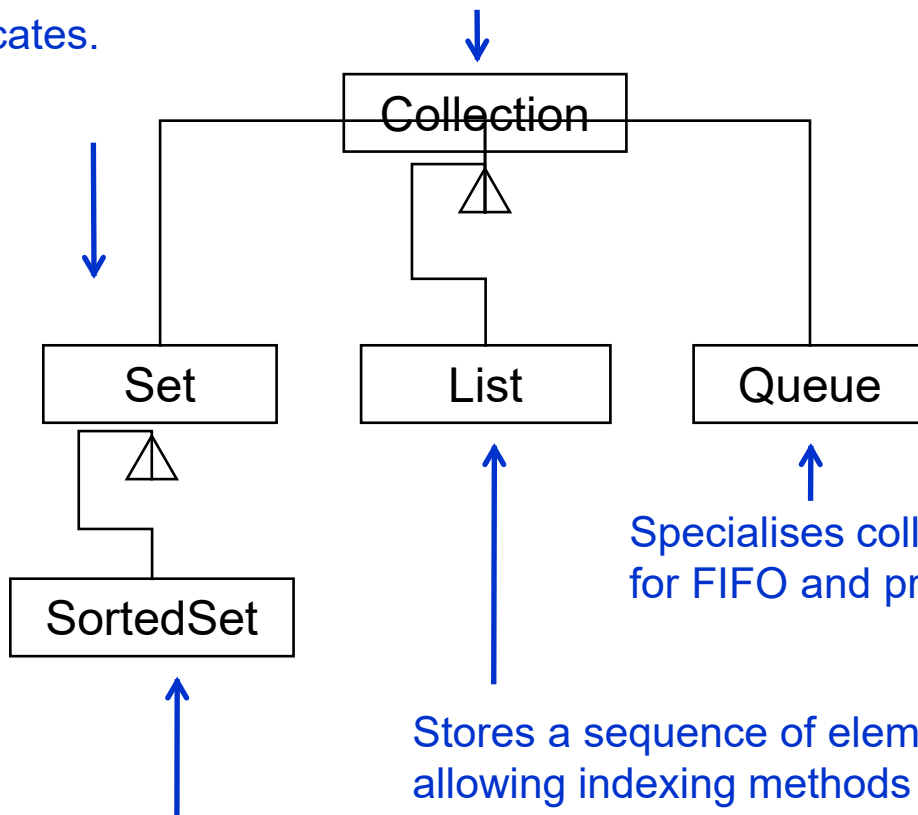
•Sorting, index searching, reversing, replacing etc.

# Interfaces

**Generalisation**

**Specialisation**

A special Collection that cannot contain duplicates.

Root interface for operations common to all types of collections

Stores mappings from keys to values

Collection

Map

Set

List

Queue

SortedSet

SortedMap

Specialises collection with operations for FIFO and priority queues.

Special map in which keys are ordered

Stores a sequence of elements, allowing indexing methods

Special Set that retains ordering of elements.

# Expansion of contracts

## <<interface>>
## List<E>

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

## <<interface>>
## Collection<E>

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

## <<interface>>
## SortedSet<E>

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

## <<interface>>
## Set<E>

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

# 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
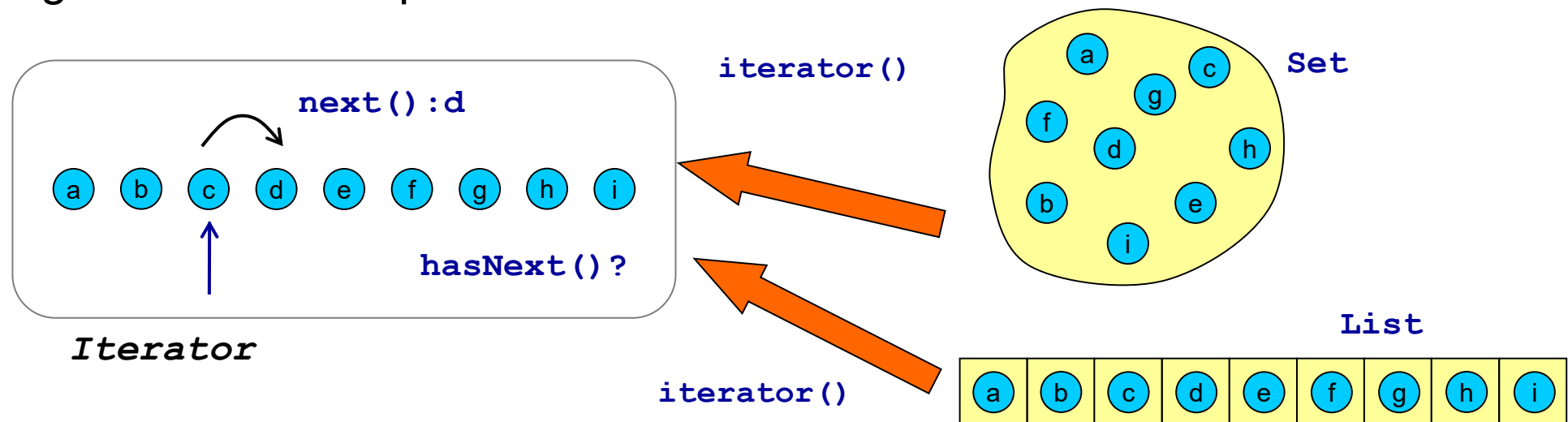
# Using an `Iterator`

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<br>+**next**():E<br>+**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`

<<interface>>
Iterable\<T>

+iterator():Iterator\<T>

Collection\<T>

MyBag\<T>

*etc*

Set\<T>    List\<T>
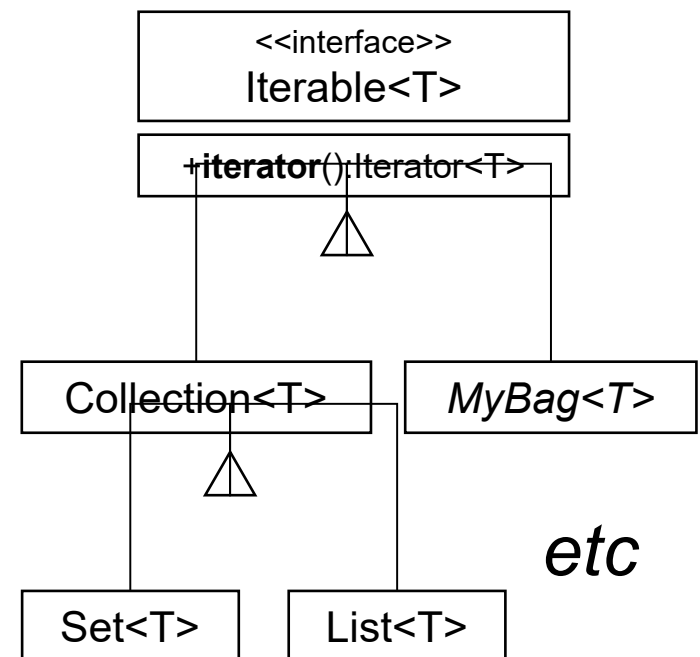
# java.util.`Collections`

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

**Sorting** lists

Index searching

Finding min/max

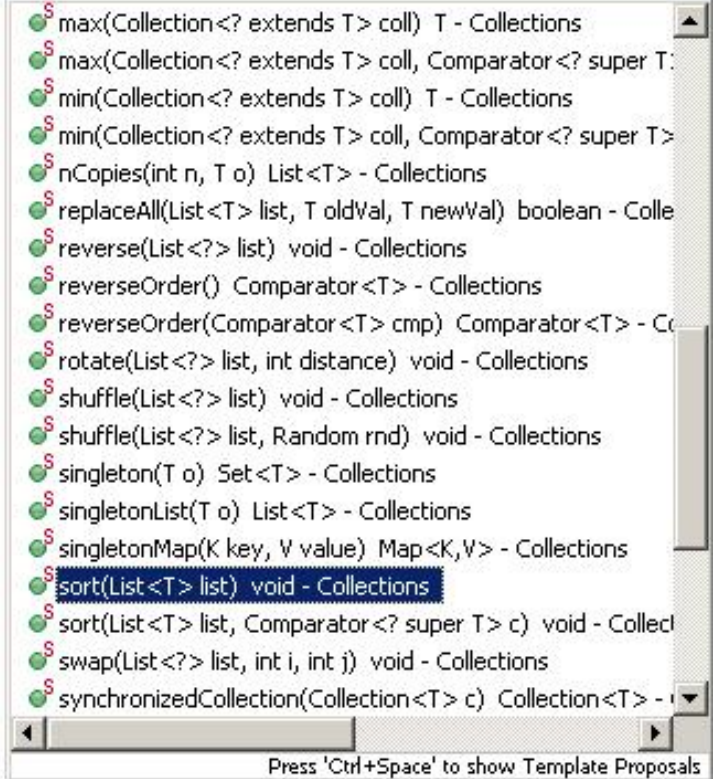Reversing elements of a list

Swapping elements of a list

Replacing elements in a list

Other nifty tricks

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

```
List<Movie> movies = new ArrayList<Movie>();
Collections.
```

max(Collection<? extends T> coll) T - Collections
max(Collection<? extends T> coll, Comparator<? super T:
min(Collection<? extends T> coll) T - Collections
min(Collection<? extends T> coll, Comparator<? super T>
nCopies(int n, T o) List<T> - Collections
replaceAll(List<T> list, T oldVal, T newVal) boolean - Colle
reverse(List<?> list) void - Collections
reverseOrder() Comparator<T> - Collections
reverseOrder(Comparator<T> cmp) Comparator<T> - Co
rotate(List<?> list, int distance) void - Collections
shuffle(List<?> list) void - Collections
shuffle(List<?> list, Random rnd) void - Collections
singleton(T o) Set<T> - Collections
singletonList(T o) List<T> - Collections
singletonMap(K key, V value) Map<K,V> - Collections
sort(List<T> list) void - Collections
sort(List<T> list, Comparator<? super T> c) void - Collecl
swap(List<?> list, int i, int j) void - Collections
synchronizedCollection(Collection<T> c) Collection<T> -

Press 'Ctrl+Space' to show Template Proposals

# 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
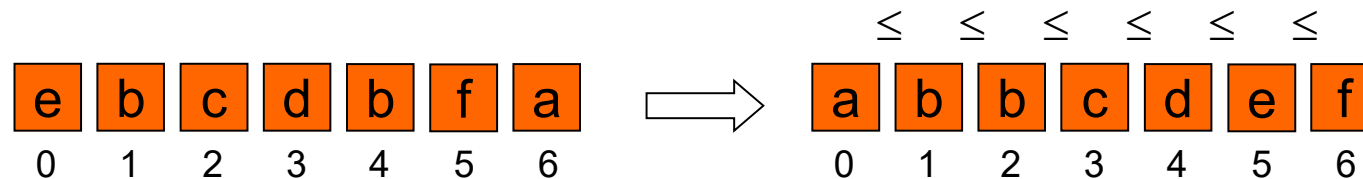


☐ What types of objects can you sort? Anything that has an **ordering**

☐ 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

```java
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 para-meterised with `CD` → `T` "becomes" `CD`

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

| CD |
| :---: |
| +getTitle():String<br>+getArtist():String<br>+**getPrice**():**Money** |

```java
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

```java
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)

```java
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.
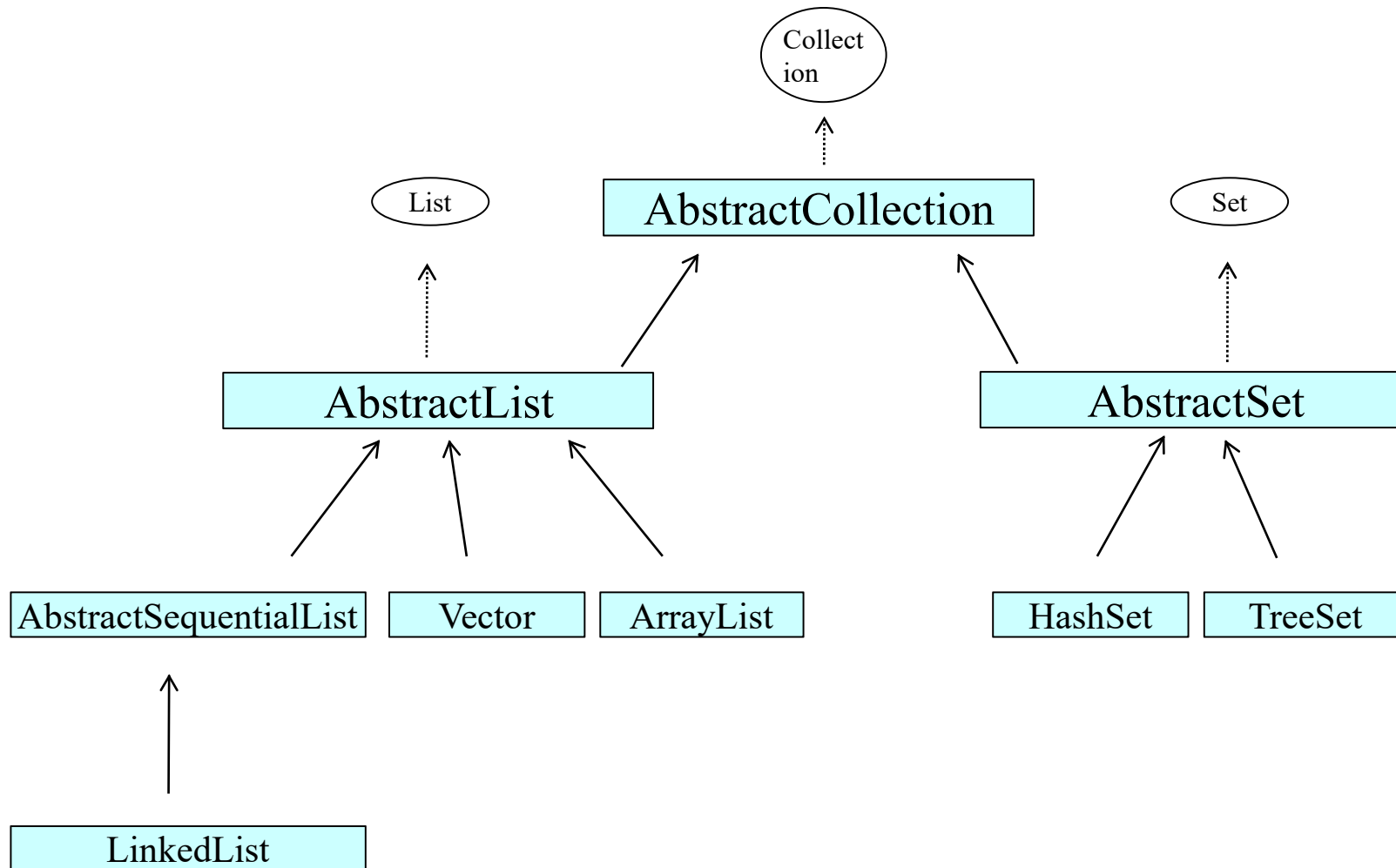
# Lists

- 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 e1 and e2 such that e1.equals(e2)

**add(x)**
→*true*

**add(b)**
→*false*
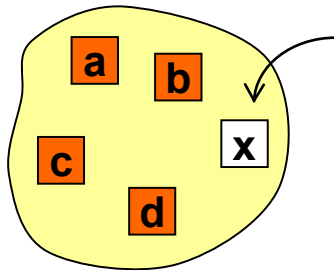
**remove(c)**
→*true*

**remove(x)**
→*false*
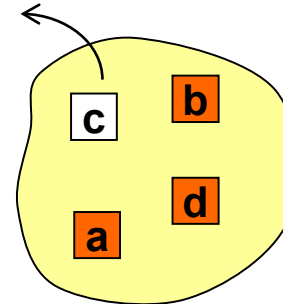
**contains(e)**
→true

**contains(x)**
→*false*
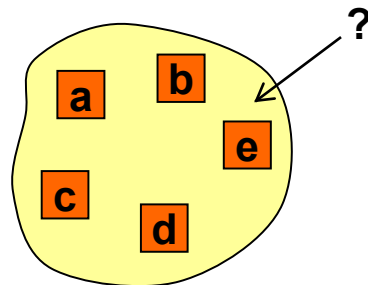
**isEmpty()**
→*false*

**size()**
→*5*

```
            <<interface>>
              Set<E>
────────────────────────────────
+add(E):boolean
+remove(Object):boolean
+contains(Object):boolean
+isEmpty():boolean
+size():int
+iterator():Iterator<E> etc...
```
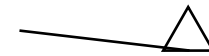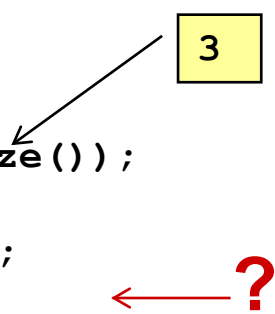
```
            <<interface>>
            SortedSet<E>
────────────────────────────────
+first():E
+last():E
etc...
```

# 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:

**HashSet<E>**

```
Set<String> words = new HashSet<String>();
words.add("Bats");
words.add("Ants");
words.add("Crabs");          3
words.add("Ants");
System.out.println(words.size());
for (String word : words) {
    System.out.println(word);
}                        ← ?
```

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

# TreeSet&lt;E&gt; (SortedSet&lt;E&gt;)

.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
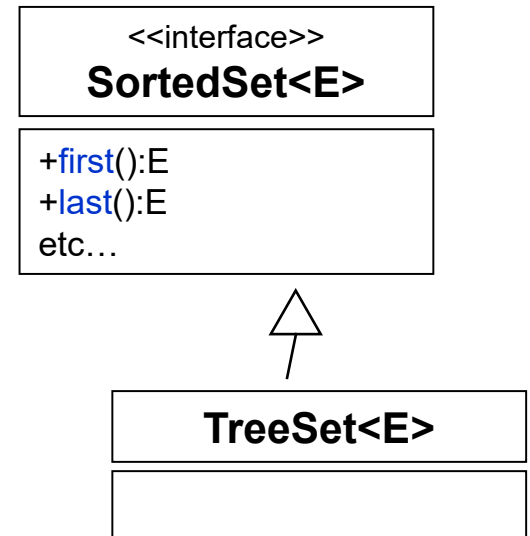
» **add()** and **remove()** preserve this condition

» **iterator()** always returns the elements in a specified order

.Two ways of specifying ordering
» Ensuring elements have natural ordering (**Comparable**)

» Giving a **Comparator&lt;E&gt;** to the constructor

.**Caution:** TreeSet considers x and y are duplicates if x.compareTo(y) == 0 (or compare(x,y) == 0)

```
<<interface>>
SortedSet<E>
─────────────
+first():E
+last():E
etc…
```

```
TreeSet<E>
─────────────
```

# 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 it's value will be removed from the Map.
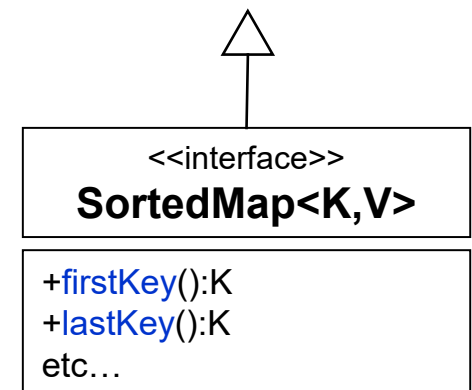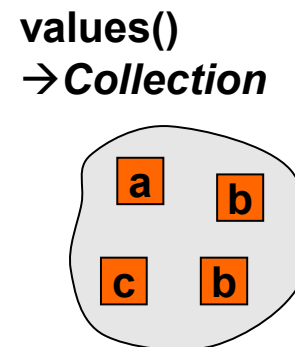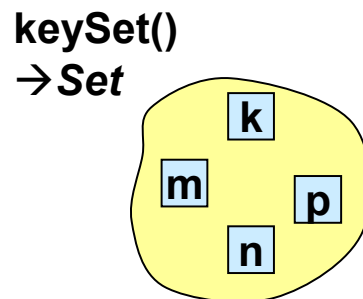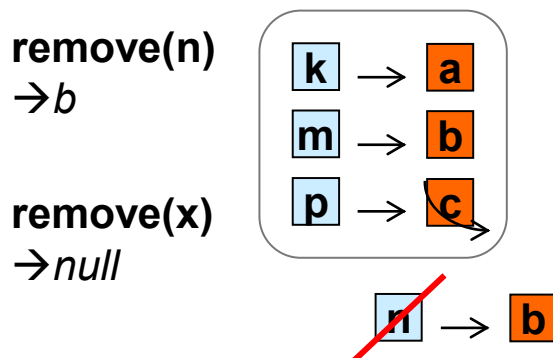
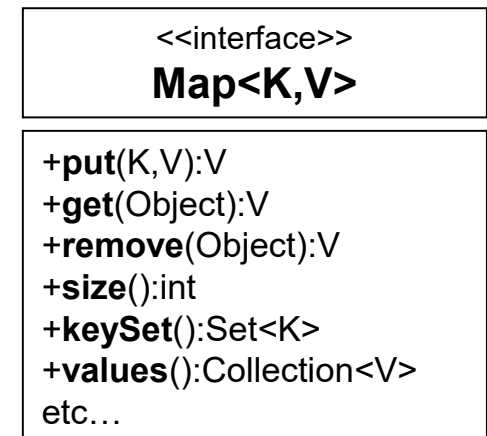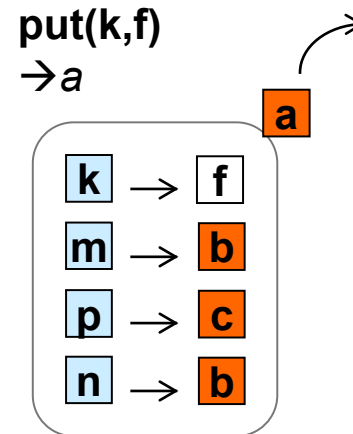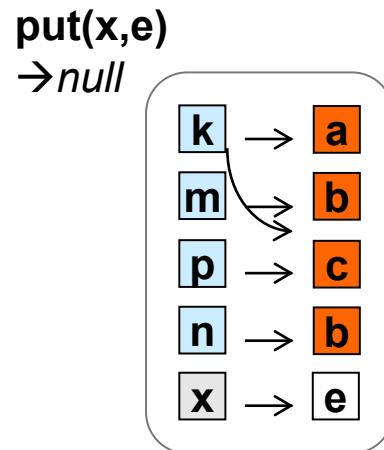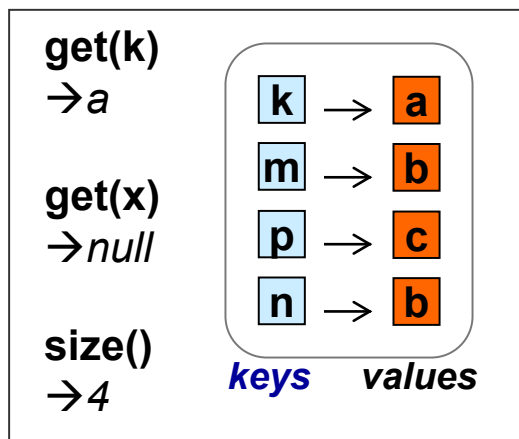- A Red-Black implementation of the Map interface

# Map<K,V>

- Stores mappings from (unique) keys (type **K**) to values (type **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`

**get(k)**
→*a*

**get(x)**
→*null*

**size()**
→*4*

keys    values
k → a
m → b
p → c
n → b

**put(x,e)**
→*null*

k → a
m → b
p → c
n → b
x → e

**put(k,f)**
→*a*

a

k → f
m → b
p → c
n → b

**remove(n)**
→*b*

**remove(x)**
→*null*

k → a
m → b
p → c

n → b

**keySet()**
→*Set*

k
m
p
n

**values()**
→*Collection*

a    b
c    b

<<interface>>
**Map<K,V>**

+**put**(K,V):V
+**get**(Object):V
+**remove**(Object):V
+**size**():int
+**keySet**():Set<K>
+**values**():Collection<V>
etc…

<<interface>>
**SortedMap<K,V>**

+firstKey():K
+lastKey():K
etc…

# HashMap<K,V>

<<interface>>
**Map<K,V>**

+**put**(K,V):V
+**get**(Object):V
+**remove**(Object):V
+**size**():int
+**keySet**():Set<K>
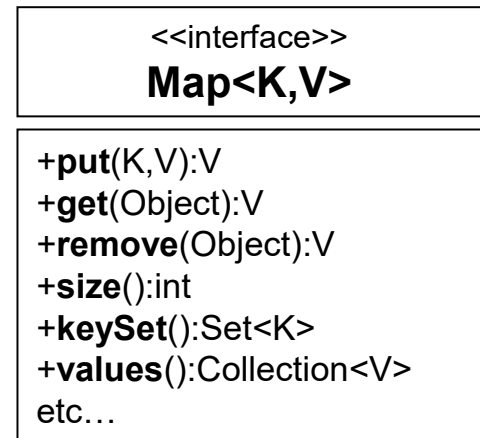+**values**():Collection<V>
etc…

**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?
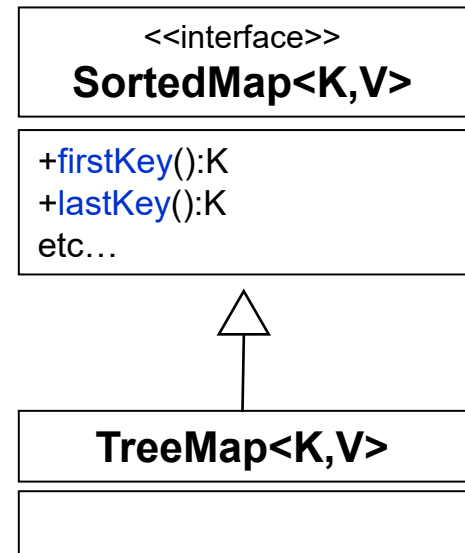
Set<String>

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**

Empty constructor
→ natural ordering

```
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());
```

4

```
<<interface>>
SortedMap<K,V>

+firstKey():K
+lastKey():K
etc…
```

```
TreeMap<K,V>
```

**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