

CHAPTER 6

Generics and Collections

(<http://docs.oracle.com/javase/tutorial/collections/index.html>)

<https://www.javatpoint.com/collections-in-java>

Objectives

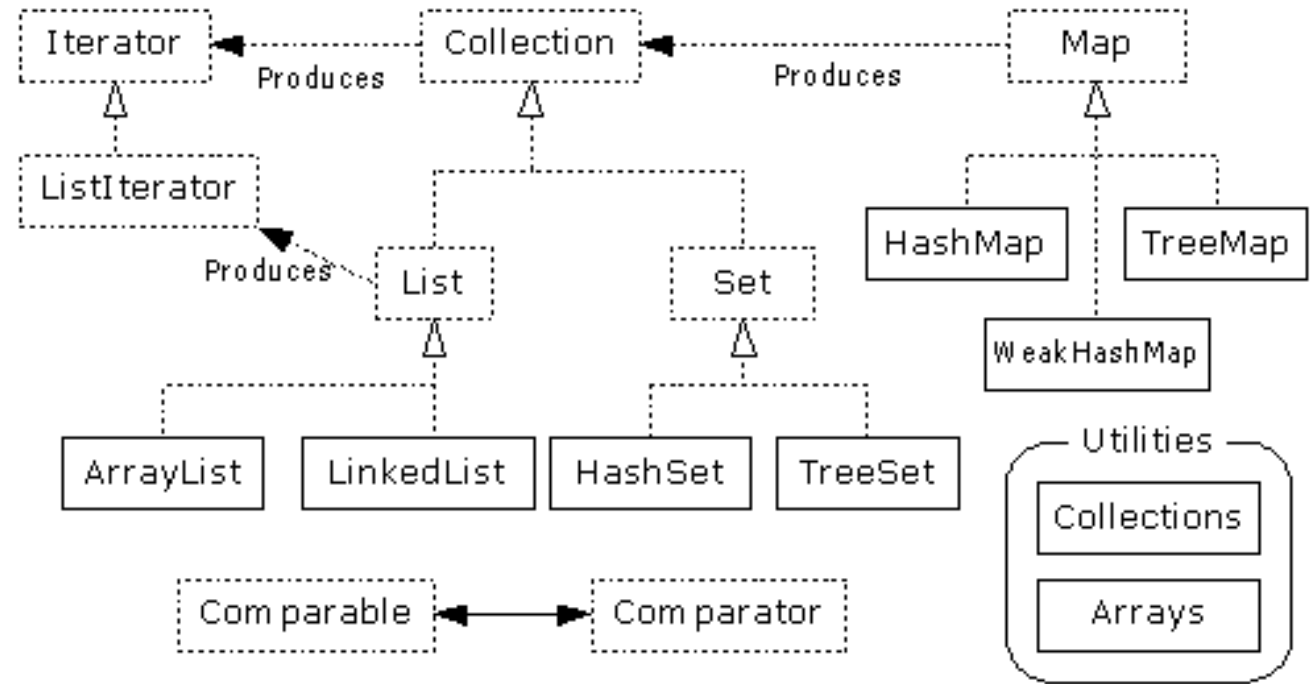
- Introduction to Generics in Java
- Declare and use Generics classes
- Collections Framework (package `java.util`):
 - **List**: ArrayList, Vector, LinkedList → Duplicates are agreed
 - **Set**: HashSet, TreeSet → Duplicates are not agreed
 - **Map**: HashMap, TreeMap
- Case study

The Collections Framework (1)

- The Java 2 platform includes a new *collections framework*.
- A *collection* is an object that represents a group of objects.
- The Collections Framework is a unified architecture for representing and manipulating collections.
- The collections framework as a whole is **not** threadsafe.
- Very useful
 - **Reduces programming effort** by providing useful data structures and algorithms so you don't have to write them yourself.
 - **Increases performance** by providing high-performance implementations of useful data structures and algorithms.
 - **Provides interoperability between unrelated APIs** by establishing a common language to pass collections back and forth.
 - **Reduces the effort required to learn APIs** by eliminating the need to learn multiple ad hoc collection APIs.
 - **Reduces the effort required to design and implement APIs** by eliminating the need to produce ad hoc collections APIs.

Collections Framework (2)

- Unified architecture for representing and manipulating collections.
- A collections framework contains three things
 - Interfaces
 - Implementations
 - Algorithms



- Interfaces, Implementations, and Algorithms
- From Thinking in Java, page 462

Collection Interfaces (1)

- java.lang.**Iterable**<T>
 - java.util.**Collection**<E>
 - java.util.**List**<E>
 - java.util.**Queue**<E>
 - java.util.**Deque**<E>
 - java.util.**Set**<E>
 - java.util.**SortedSet**<E>
 - java.util.**NavigableSet**<E>
 - java.util.**Map**<K,V>
 - java.util.**SortedMap**<K,V>
 - java.util.**NavigableMap**<K,V>

Methods declared in these interfaces can work on a list containing elements which belong to arbitrary type. T: type, E: Element, K: Key, V: Value

Details of this will be introduced in the topic Generic

3 types of group:

List can contain duplicate elements

Set can contain distinct elements only

Map can contain pairs <key, value>. Key of element is data for fast searching

Queue, Deque contains methods of restricted list.

Common methods on group are: Add, Remove, Search, Clear,...

Common Methods of the interface Collection

Method	Description
<code>add(Object x)</code>	Adds x to this collection
<code>addAll(Collection c)</code>	Adds every element of c to this collection
<code>clear()</code>	Removes every element from this collection
<code>contains(Object x)</code>	Returns true if this collection contains x
<code>containsAll(Collection c)</code>	Returns true if this collection contains every element of c
<code>isEmpty()</code>	Returns true if this collection contains no elements
<code>iterator()</code>	Returns an Iterator over this collection (see below)
<code>remove(Object x)</code>	Removes x from this collection
<code>removeAll(Collection c)</code>	Removes every element in c from this collection
<code>retainAll(Collection c)</code>	Removes from this collection every element that is not in c
<code>size()</code>	Returns the number of elements in this collection
<code>toArray()</code>	Returns an array containing the elements in this collection

Elements can be stored using some ways such as an array, a tree, a hash table. Sometimes, we want to traverse elements as a list → We need a list of references → **Iterator**

The Collection Framework...

Central Interfaces

- `java.util.Collection<E>`
 - `java.util.List<E>`
 - `java.util.Queue<E>`
 - `java.util.Deque<E>`
 - `java.util.Set<E>`
 - `java.util.SortedSet<E>`
 - `java.util.NavigableSet<E>`
 - `java.util.Map<K,V>`
 - `java.util.SortedMap<K,V>`
 - `java.util.NavigableMap<K,V>`

Common Used Classes

- `java.util.ArrayList<E>`
- `java.util.Vector<E>`
- `java.util.HashSet<E>`
- `java.util.TreeSet<E>`
- `java.util.HashMap<K,V>`
- `java.util.TreeMap<K,V>`

Store: Dynamic array
Use index to access an element.

Store: Specific structure/tree
Use iterator to access elements

`keySet()`
`values()`

Use
iterator

A TreeSet will stored elements using ascending order. Natural ordering is applied to numbers and lexicographic (dictionary) ordering is applied to strings.

If you want a TreeSet containing your own objects, you must implement the method `compareTo(Object)`, declared in the `Comparable` interface.

Arrays vs Collections

Arrays

1. Size is fixed

2. to memory arrays are not good to use, but to performance its better to use arrays

3. Can hold both primitive types(byte, sort, int, long ...etc) and object types.

4. There is no underlying data structures in arrays. The array itself used as data structure in java.

5. There is no utility methods in arrays

Collections

1. Size is not fixed (dynamic in size), size is growable.

2. to memory Collections are better to use, but to performance collection are not good to use.

3. Can hold only object types

4. Every Collection class there is underlying data structure.

5. Every Collection provides utility methods (sorting, searching, retrieving etc...). It will reduce the coding time.

Iterator Interface and Position

- Defines three fundamental methods
 - `Object next()`
 - `boolean hasNext()`
 - `void remove()`
- These three methods provide access to the contents of the collection
- An Iterator knows position within collection
- Each call to `next()` “reads” an element from the collection
 - Then you can use it or remove it

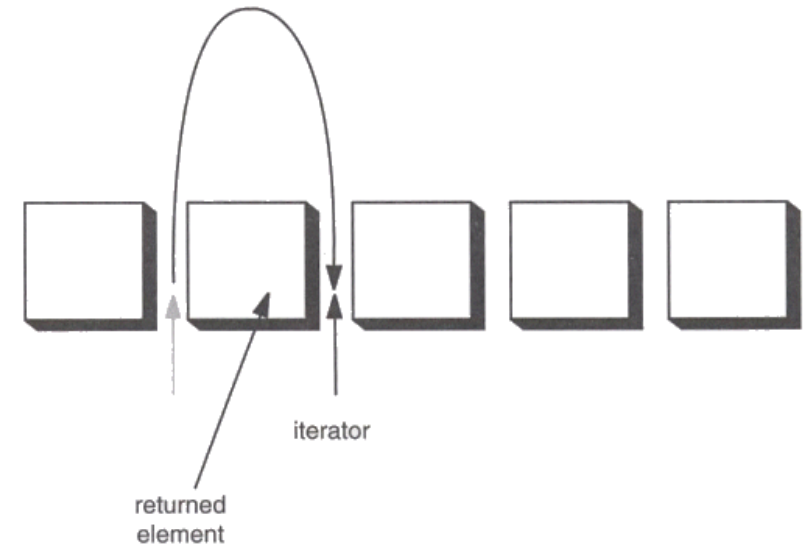
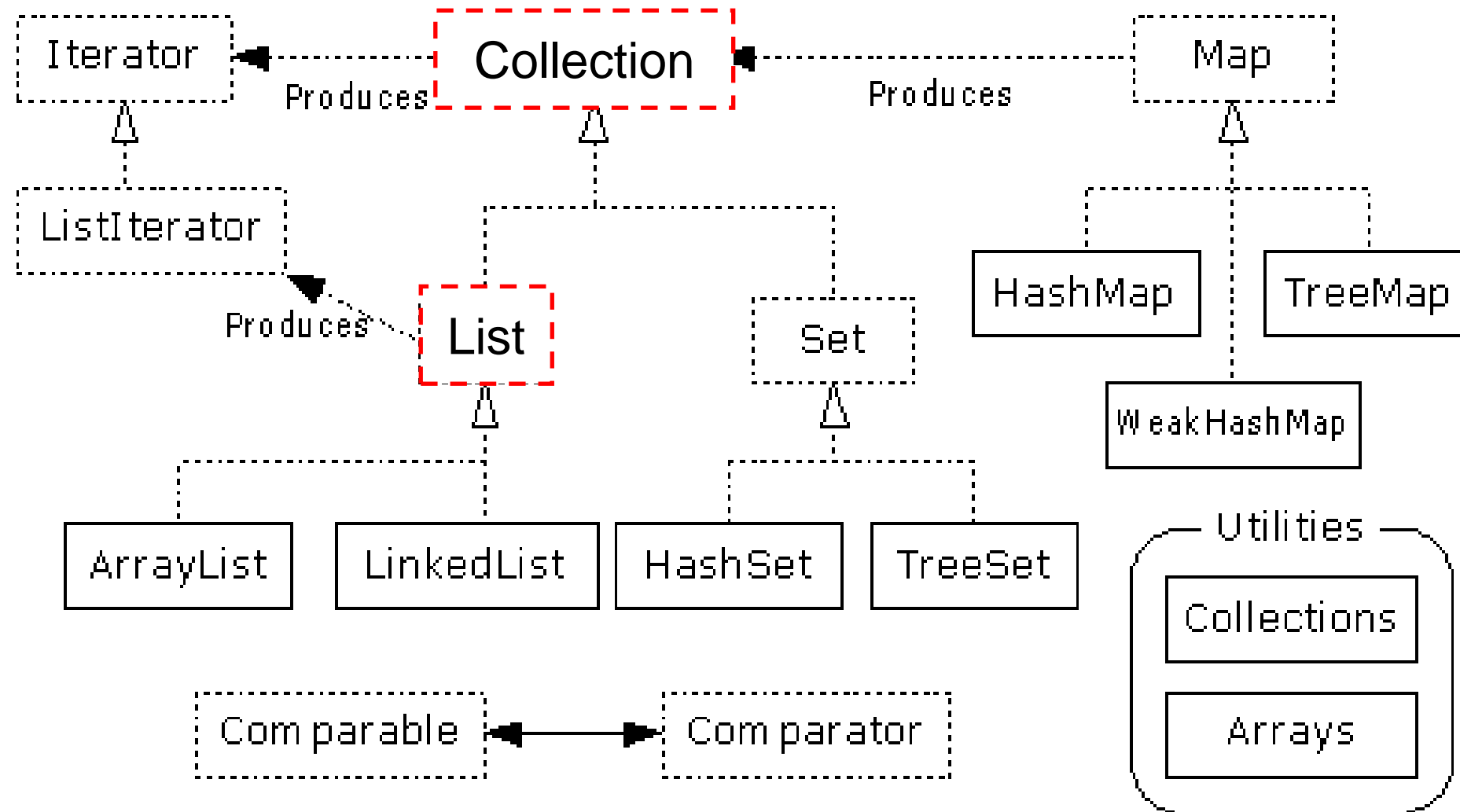


Figure 2-3: Advancing an iterator

Example - SimpleCollection

```
public class SimpleCollection {
    public static void main(String[] args) {
        Collection c;
        c = new ArrayList();
        System.out.println(c.getClass().getName());
        for (int i=1; i <= 10; i++) {
            c.add(i + " * " + i + " = "+i*i);
        }
        Iterator iter = c.iterator();
        while (iter.hasNext())
            System.out.println(iter.next());
    }
}
```

List Interface Context



Lists

- A List keeps its elements in the order in which they were added.
- Each element of a List has an index, starting from 0.
- Common methods:
 - **void add(int index, Object x)**
 - **Object get(int index)**
 - **int indexOf(Object x)**
 - **Object remove(int index)**
- Classes Implementing the interface List
 - AbstractList
 - ArrayList
 - Vector (like ArrayList but it is **synchronized**)
 - LinkedList: *linked lists can be used as a stack, queue, or double-ended queue (deque)*

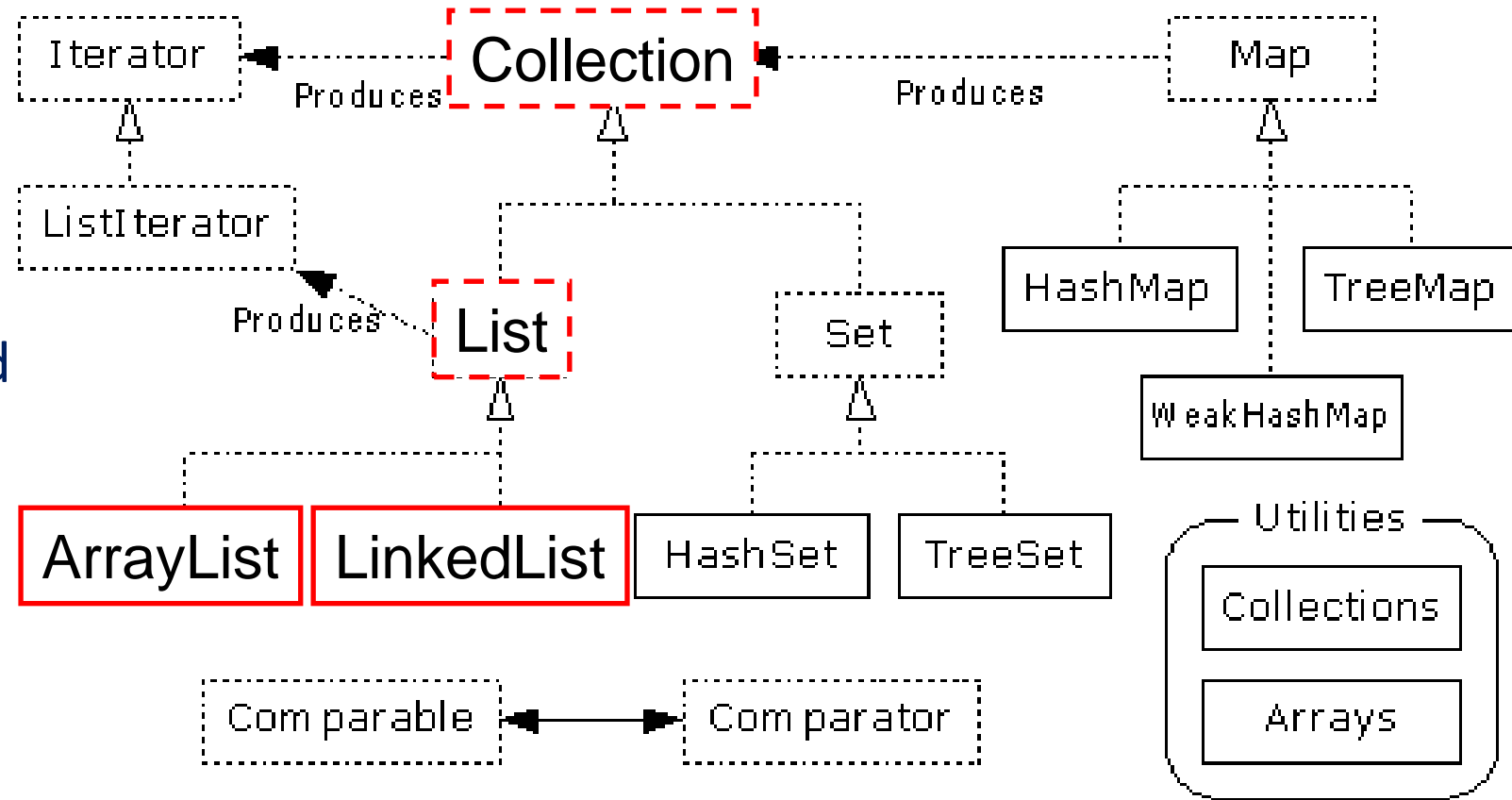
List Implementations

- ArrayList

- low cost random access
- high cost insert and delete
- array that resizes if need be

- LinkedList

- sequential access
- low cost insert and delete
- high cost random access



ArrayList

- ArrayList is a class that implements the List interface.
- The advantage of ArrayList over the general Array is that ArrayList is dynamic and the size of ArrayList can grow or shrink.
- ArrayList stores the elements in the insertion order.
- ArrayList can have duplicate elements. ArrayList is **non** synchronized.

Sr.No.	Constructor & Description
1	ArrayList() This constructor is used to create an empty list with an initial capacity sufficient to hold 10 elements.
2	ArrayList(Collection c) This constructor is used to create a list containing the elements of the specified collection.
3	ArrayList(int initialCapacity) This constructor is used to create an empty list with an initial capacity.

Methods (1)

Methods	Description
void add(int index, Object element)	Inserts the specified element at the specified position in this list.
boolean add(Object o)	Appends the specified element to the end of this list.
boolean addAll(Collection c)	Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator
void clear()	Removes all of the elements from this list.
boolean contains(Object o)	returns true if this list contains the specified element.
Object get(int index)	returns the element at the specified position in this list.

Methods (2)

Methods	Description
protected void removeRange(int fromIndex, int toIndex)	removes from this list all of the elements whose index is between fromIndex(inclusive) and toIndex(exclusive).
Object set(int index, Object element)	replaces the element at the specified position in this list with the specified element.
int size()	returns the number of elements in this list.
int indexOf(Object o) int lastIndexOf(Object o)	returns the index of the first (last) occurrence of the specified element in this list, or -1 if this list does not contain the element.
Object remove(int index)	removes the element at the specified position in this list.
boolean remove(Object o)	removes the first occurrence of the specified element from this list, if it is present.

Example

```
public class ArrayListDemo {  
    public static void main(String args[]) {  
        ArrayList al = new ArrayList();  
        System.out.println("The size at beginning: " +  
al.size());  
        //add elements  
        al.add("C");           al.add("A");  
        al.add("E");           al.add("B");  
        al.add("D");           al.add("F");  
        al.add(1, "A2");  
        System.out.println("The size after: " + al.size());  
        System.out.println("Content: " + al);  
        al.remove("F"); al.remove(2);  
        System.out.println("The Size after to remove:"+ al.size());  
        System.out.println("Content: " + al);  
    }  
}
```

The size at beginning: 0
The size after: 7
Content: [C, A2, A, E, B, D, F]
The Size after to remove: 5
Content: [C, A2, E, B, D]

Vector

- Vector implements a dynamic array. It is similar to ArrayList, but with two differences
- Vector is synchronized.
- Vector contains many legacy methods that are not part of the collections framework.
- Constructors:
- **Vector()**: This constructor creates a default vector, which has an initial capacity of 10.
- **Vector(int size)**: This constructor accepts an argument that equals to the required size
- **Vector(int size, int incr)**: This constructor creates a vector whose initial capacity is specified by size and whose increment is specified by incr.
- **Vector(Collection c)**: This constructor creates a vector that contains the elements of collection c.

Methods (1)

void add(int index, Object element)

Inserts the specified element at the specified position in this Vector.

boolean add(Object o): Appends the specified element to the end of this Vector.

boolean addAll(Collection c)

Appends all of the elements in the specified Collection to the end of this Vector

void clear(): Removes all of the elements from this vector.

boolean contains(Object elem): Tests if the specified object is a component in this vector.

boolean containsAll(Collection c)

Returns true if this vector contains all of the elements in the specified Collection.

Object elementAt(int index): Returns the component at the specified index.

Enumeration elements(): Returns an enumeration of the components of this vector.

boolean equals(Object o): Compares the specified Object with this vector for equality.

Methods (2)

Object firstElement(): Returns the first component (the item at index 0) of this vector.

Object get(int index): Returns the element at the specified position in this vector.

int indexOf(Object elem), int indexOf(Object elem, int index) : Searches for the first occurrence of the given argument (beginning the search at index,), testing for equality using the equals method.

void insertElementAt(Object obj, int index)

Inserts the specified object as a component in this vector at the specified index.

boolean isEmpty(): Tests if this vector has no components.

Object lastElement(): Returns the last component of the vector.

Object remove(int index), boolean remove(Object o): Removes the element at the specified position in this vector. Removes the first occurrence of the specified element in this vector,

void removeAllElements()

Removes all components from this vector and sets its size to zero.

Methods (3)

Object set(int index, Object element)

Replaces the element at the specified position in this vector with the specified element.

int size(): Returns the number of components in this vector.

List subList(int fromIndex, int toIndex)

Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive.

Object[] toArray()

Returns an array containing all of the elements in this vector in the correct order.

String toString()

Returns a string representation of this vector, containing the String representation of each element.

void trimToSize()

Trims the capacity of this vector to be the vector's current size.

LinkedList overview

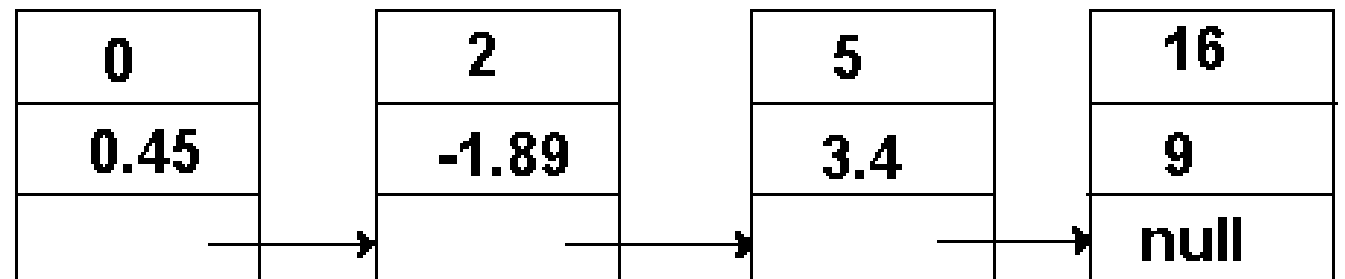
- Stores each element in a node
- Each node stores a link to the next and previous nodes
- Insertion and removal are inexpensive
 - just update the links in the surrounding nodes
- Linear traversal is inexpensive
- Random access is expensive
 - Start from beginning or end and traverse each node while counting

```
private static class Entry {
    Object element;
    Entry next;
    Entry previous;
    Entry(Object element, Entry
next, Entry previous) {
        this.element = element;
        this.next = next;
        this.previous = previous;
    }
}

private Entry header = new
Entry(null, null, null);
public LinkedList() {
    header.next = header.previous
= header;
}
```

LinkedList methods

- The list is sequential, so access it that way
 - `ListIterator listIterator()`
- ListIterator knows about position
 - use `add()` from ListIterator to add at a position
 - use `remove()` from ListIterator to remove at a position
- LinkedList knows a few things too
 - `void addFirst(Object o)`, `void addLast(Object o)`
 - `Object getFirst()`, `Object getLast()`
 - `Object removeFirst()`, `Object removeLast()`
- Example: Polynomial
 - $0.45 - 1.89 x^2 + 3.4 x^5 + 9 x^{16}$



Generic in java

- The **Java Generics** programming is introduced in Java SE 5 to deal with type-safe objects.
- Generics, forces the java programmer to store specific type of objects.
- There are mainly 3 advantages of generics:
 - **Type-safety** : We can hold only a single type of objects in generics. It doesn't allow to store other objects.
 - **Type casting is not required**: There is no need to typecast the object. we need to type cast.

```
List list = new ArrayList();
```

```
list.add("hello");
```

```
String s = (String) list.get(0); //typecasting
```

- **we don't need to typecast the object.**

```
List<String> list =new ArrayList<String>();
```

```
list.add("hello");
```

```
String s = list.get(0);
```


Generic in java...

- **Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime.

```
List<String> list = new ArrayList<String>();  
list.add("hello");  
list.add(32); //Compile Time Error
```

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

```
mylist.add(10);  
mylist.add("Hi"); //error  
mylist.add(true); //error  
mylist.add(15);
```

Generic



Good Type Variable Names

Type Variable	Name Meaning
E	Element type in a collection
K	Key type in a map
V	Value type in a map
T	General type
S, U	Additional general types

- A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

```
public class Box<T> {  
    private T t;  
    public void set(T t) {  
        this.t = t;  
    }  
    public T get() {  
        return t;  
    }  
}
```

Class Pair

```
public class Pair<T, S>
{
    private T first;
    private S second;

    public Pair(T firstElement, S secondElement)
    {
        first = firstElement;
        second = secondElement;
    }
    public T getFirst() { return first; }
    public S getSecond() { return second; }
    public String toString() {
        return "(" + first + ", " + second + ")";
    }
}
```

Declaring a Generic Class

Syntax *accessSpecifier* class *GenericClassName*<*TypeVariable*₁, *TypeVariable*₂, . . . >
 {
 instance variables
 constructors
 methods
 }

Example

Supply a variable for each type parameter.

```
public class Pair<T, S>  
{  
    private T first;  
    private S second;  
    . . .  
    public T getFirst() { return first; }  
    . . .  
}
```

A method with a variable return type

Instance variables with a variable data type

Class Pair

```
class Dictionary<K, V> {
    private K key;    private V
value;
    public Dictionary(K key, V value)
{
    this.key = key;
    this.value = value;  }
    public K getKey() {
        return key;    }
    public void setKey(K key) {
        this.key = key;
    }
    public V getValue() {
        return value;
    }
    public void setValue(V value) {
        this.value = value;
    }
}
```

```
public class DemoGeneric {
    public static void
main(String[] args) {
    Dictionary<String, String> d
= new Dictionary<String,
String>("Study", "hoc");
    String english = d.getKey();
    String vietnamese =
        d.getValue();
    System.out.println(english + ":
" + vietnamese); //Output: Study:
hoc
    }
}
```

```
class Book extends Dictionary<String, String> {
    public Book(String key, String value) {
        super(key, value);
    }
}

public class Demo {
    public static void main(String[] args) {
        Book l = new Book("Study", "hoc");
        String english = l.getKey();
        String vietnamese = l.getValue();
        System.out.println(english + ": " + vietnamese);
        // Ouput: Study: hoc
    }
}
```

Generic Methods

- **Generic method:** method with a type variable
- Can be defined inside non-generic classes
- Example: Want to declare a method that can print an array of any type:

```
public class ArrayUtil {  
    public <T> static void  
    print(T[] a)  
    {  
        . . .  
    }  
    . . .  
}
```

```
public class ArrayUtil{  
    public static void  
    print(String[] a) {  
        for (String e : a)  
            System.out.print(e + " ");  
            System.out.println();  
        }  
    . . .  
}
```

Generic Methods

- In order to make the method into a generic method:
 - *Replace `String` with a type parameter, say `E`, to denote the element type*
 - *Supply the type parameters between the method's modifiers and return type*

```
public static <E> void print(E[] a)
{
    for (E e : a)
        System.out.print(e + " ");
    System.out.println();
}
```

- When calling a generic method, you need not instantiate the type variables:

```
Rectangle[] rectangles = . . .;
ArrayUtil.print(rectangles);
```

- The compiler deduces that `E` is `Rectangle`

Defining a Generic Method

Syntax *modifiers* <TypeVariable₁, TypeVariable₂, . . .> *returnType* *methodName(parameters)*
 {
 body
 }

Example

```
public static <E> void print(E[] a)
{
    for (E e : a)
        System.out.print(e + " ");
    System.out.println();
}
```

Supply the type variable before the return type.

Local variable with a variable data type

Sorting with Comparable and Comparator

Comparable	Comparator
1) Comparable provides a single sorting sequence . In other words, we can sort the collection on the basis of a single element such as id, name, and price.	The Comparator provides multiple sorting sequences . In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc.
2) Comparable affects the original class , i.e., the actual class is modified.	Comparator doesn't affect the original class , i.e., the actual class is not modified.
3) Comparable provides compareTo() method to sort elements.	Comparator provides compare() method to sort elements.
4) Comparable is present in java.lang package.	A Comparator is present in the java.util package.
5) We can sort the list elements of Comparable type by Collections.sort(List) method.	We can sort the list elements of Comparator type by Collections.sort(List, Comparator) method.

Java Comparable Example

```
class Student implements Comparable<Student>{
.....
public int compareTo(Student st) {
    if (age==st.age)
        return 0;
    else if (age>st.age)
        return 1;
    else
        return -1;
}
}
```

```
ArrayList<Student> al=new ArrayList<Student>();
al.add(new Student(101,"Vijay",23));
al.add(new Student(106,"Ajay",27));
al.add(new Student(105,"Jai",21));
Collections.sort(al);
```

Comparator Interface

```
class AgeComparator implements Comparator<Student>{  
    public int compare(Student s1,Student s2) {  
        if (s1.age==s2.age)  
            return 0;  
        else if (s1.age>s2.age)  
            return 1;  
        else  
            return -1;  
    }  
}
```

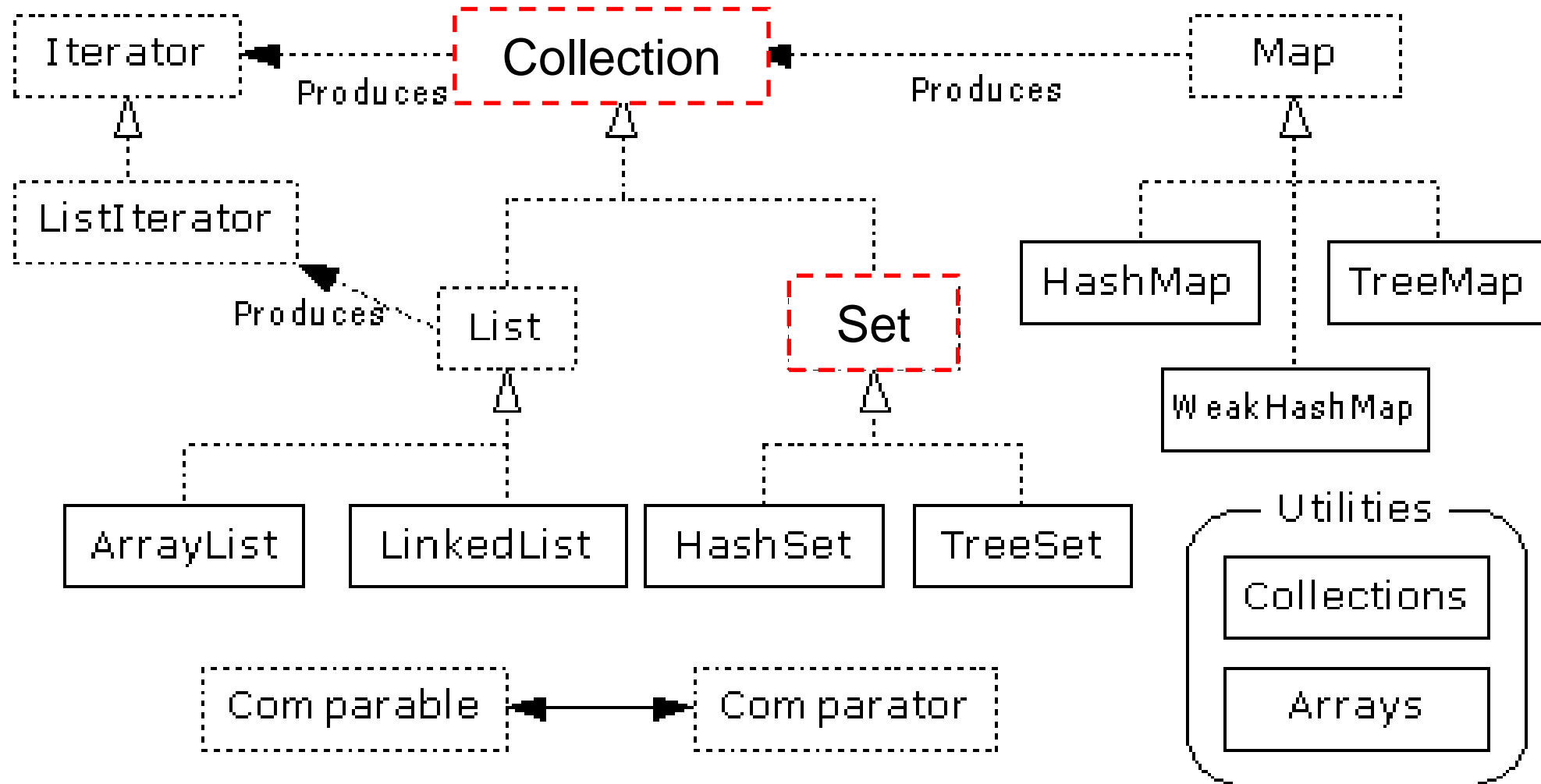
```
ArrayList<Student> al=new ArrayList<St  
udent>();  
al.add(new Student(101,"Vijay",23));  
al.add(new Student(106,"Ajay",27));  
al.add(new Student(105,"Jai",21));  
Collections.sort(al, new AgeComparator());
```

```
List<Student> al=new ArrayList<>();
    al.add(new Student(101,"Vijay", 23));
    al.add(new Student(106,"Ajay", 27));
    al.add(new Student(105,"Jai", 21));
    al.sort(new Comparator<Student>() {
        @Override
        public int compare(Student o1, Student o2) {
            return o1.getAge()-o2.getAge();
        }
    });
```

Lambda Expression

```
al.sort((Student o1, Student o2) -> o1.getAge()-
o2.getAge());
```

Set Interface Context



Sets

- Lists are based on an ordering of their members. Sets have no concept of order.
- A Set is just a cluster of references to objects.
- Sets may **not** contain **duplicate** elements.
- Sets use the equals() method, not the == operator, to check for duplication of elements.

```
void addTwice(Set set) {  
    set.clear();  
    Point p1 = new Point(10, 20);  
    Point p2 = new Point(10, 20);  
    set.add(p1);  
    set.add(p2);  
    System.out.println(set.size());  
}
```



will print out 1, not 2.

Sets...

- Set extends Collection but does not add any additional methods.
- The two most commonly used implementing classes are:
 - TreeSet
 - Guarantees that the sorted set will be in ascending element order.
 - $\log(n)$ time cost for the basic operations (add, remove and contains).
 - HashSet
 - Constant time performance for the basic operations (add, remove, contains and size).
- Ordered Tree – Introduced in the subject Discrete Mathematics
- Set: Group of different elements
- TreeSet: Set + ordered tree, each element is called as node
- Iterator: An operation in which references of all nodes are grouped to make a linked list. Iterator is a way to access every node of a tree.
- Linked list: a group of elements, each element contains a reference to the next

TreeSet = Set + Tree

The result may be:

```
Random r = new Random();
TreeSet myset = new TreeSet();
for (int i = 0; i < 10; i++) {
    int number = r.nextInt(100);
    myset.add(number);
}
//using Iterator
Iterator iter = myset.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
```



7
27
36
41
43
46
49
57
75
83

Using the TreeSet class & Iterator

```
import java.util.TreeSet;
import java.util.Iterator;
public class UseTreeSet {
    public static void main (String[] args) {
        TreeSet t= new TreeSet();
        t.add(5); t.add(2); t.add(9); t.add(30); t.add(9);
        System.out.println(t);
        t.remove(9);
        System.out.println(t);
        Iterator it= t.iterator();
        while (it.hasNext())
            System.out.print(it.next() + ", ");
        System.out.println();
    }
}
```

Output - Chapter08 (run)

run:

[2, 5, 9, 30]

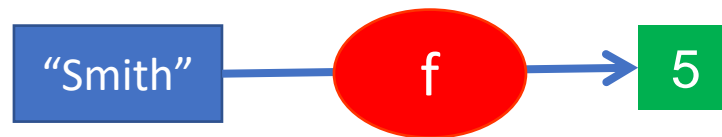
[2, 5, 30]

2, 5, 30,

- A TreeSet will stored elements using ascending order. Natural ordering is applied to numbers and lexicographic (dictionary) ordering is applied to strings. If you want a TreeSet containing your own objects, you must implement the method `compareTo(Object)`, declared in the `Comparable` interface.

Hash Table

- In array, elements are stored in a contiguous memory blocks → Linear search is applied → slow, binary search is an improvement.
- Hash table: elements can be stored in a different memory blocks. The index of an element is determined by a function (hash function) → Add/Search operation is very fast ($O(1)$).



The hash function f may be:

$'S' * 10000 + 'm' * 1000 + 'i' * 100 + 't' * 10 + 'h' \% 50$

49	
14	Brown
9	Hoa
5	Smith
0	Line1

HashSet = Set + Hash Table

```
Random r = new Random();  
HashSet myset = new HashSet();  
for (int i = 0; i < 10; i++) {  
    int number = r.nextInt(100);  
    myset.add(number);  
}  
//using Iterator  
Iterator iter = myset.iterator();  
while (iter.hasNext()) {  
    System.out.println(iter.next());  
}
```

The result may be:



84
55
7
76
77
95
94
12
91
44

HashSet - TreeSet

- If you care about iteration order, use a Tree Set and pay the time penalty.
- If iteration order doesn't matter, use the higher-performance Hash Set.
- Tree Sets rely on all their elements implementing the interface `java.lang.Comparable`.

`public int compareTo(Object x)`

- Returns a positive number if the current object is “greater than” x, by whatever definition of “greater than” the class itself wants to use.

How to TreeSet ordering elements?

```
class Student implements
Comparable{
    int no;
    ...
    public int compareTo(Object o){
        Student st = (Student) o;
        if(no > st.getNo())
            return 1;
        else if(no == st.getNo())
            return 0;
        else
            return -1;
    }
    . . .
}
```

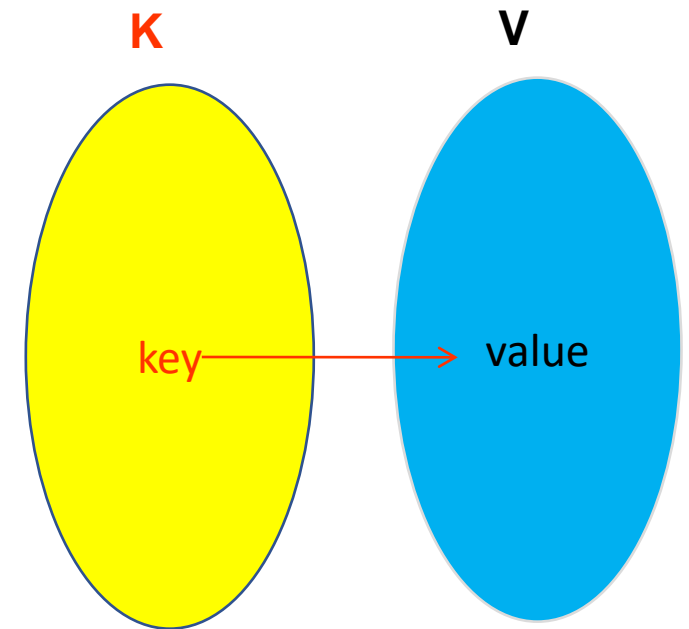
Comparing 2 students
based on their IDs (
field no)

```
public static void main(String[] args) {
    Random r = new Random();
    TreeSet myset = new TreeSet();
    for (int i = 0; i < 10; i++) {
        int no = r.nextInt(100);
        Student st = new Student(no, "abc");
        myset.add(st);
    }
    //using Iterator
    Iterator iter = myset.iterator();
    while (iter.hasNext()) {
        Student st = (Student)iter.next();
        System.out.println("No: " +
st.getNo());
    }
}
```

No: 2
No: 8
No: 11
No: 19
No: 33
No: 52
No: 78
No: 83
No: 92
No: 96

Maps

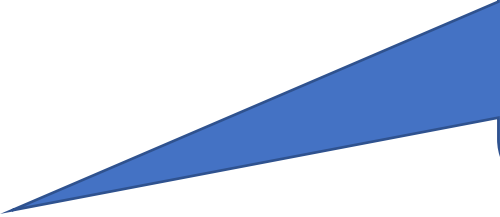
- Map doesn't implement the java.util.Collection interface.
- A Map combines **two** collections, called keys and values.
- The Map's job is to associate exactly one value with each key.
- A Map like a dictionary.
- Maps check for key uniqueness based on the equals() method, not the == operator.
- IDs, Item code, roll numbers are keys.
- The normal data type for keys is **String**.
- Java's two most important Map classes:
 - **HashMap** (mapping keys are unpredictable order – hash table is used, hash function is pre-defined in the Java Library).
 - **TreeMap** (mapping keys are natural order)-> all keys must implement Comparable (a tree is used to store elements).



Each element: <key,value>

HashMap

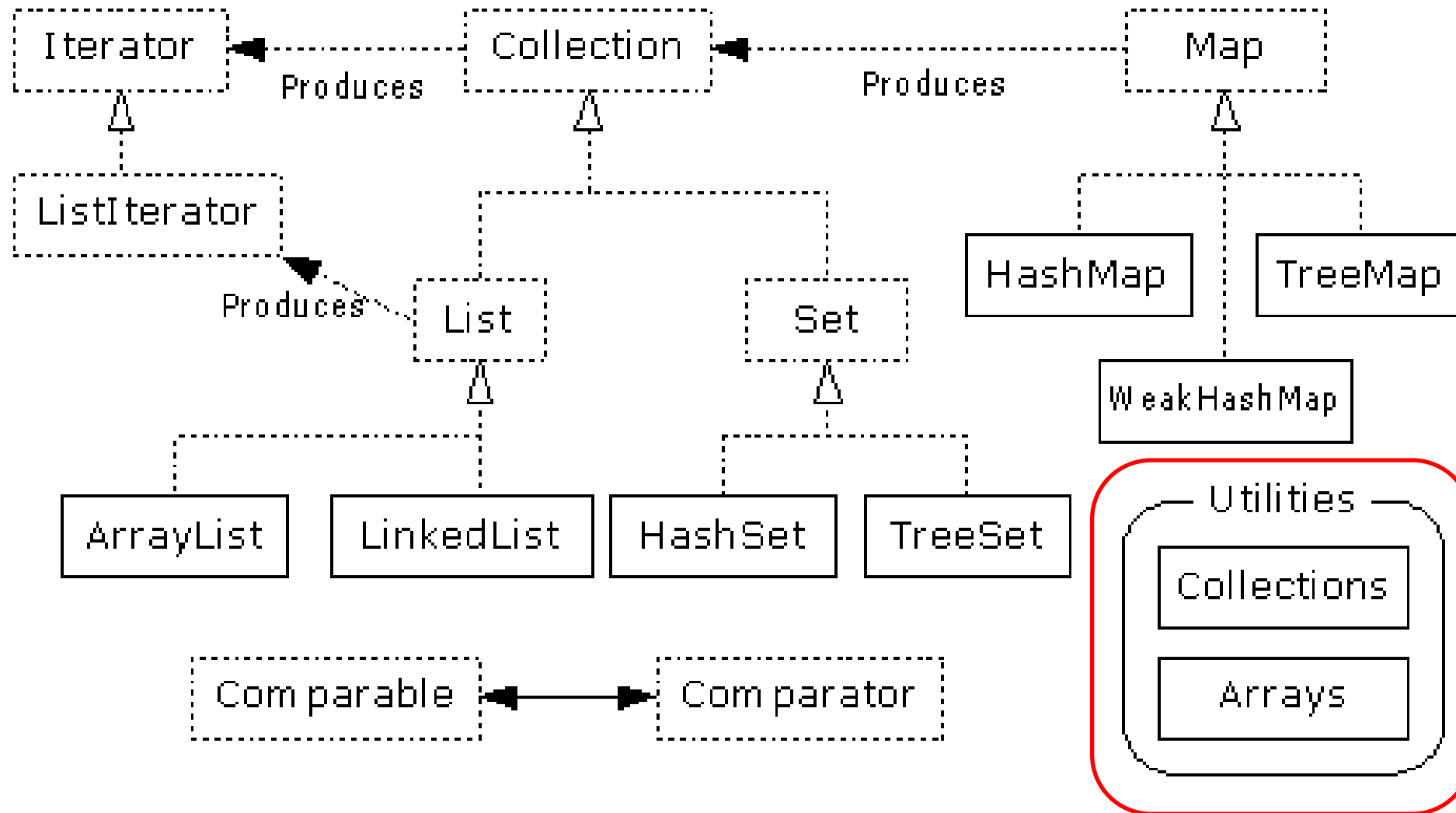
```
public static void main(String[] args) {  
    HashMap mymap = new HashMap();  
    mymap.put(1, "One");  
    mymap.put(2, "Two");  
    mymap.put(3, "Three");  
    mymap.put(4, "Four");  
    //using Iterator  
    Iterator iter = mymap.keySet().iterator();  
    while (iter.hasNext()) {  
        Object key = iter.next();  
        System.out.println(key + ": " + mymap.get(key));  
    }  
}
```



//output
1: One
2: Two
3: Three
4: Four

Key: integer, value: String

Utilities Context



Utilities

- The Collections class provides a number of static methods for fundamental algorithms
- Most operate on Lists, some on all Collections
 - Sort, Search, Shuffle
 - Reverse, fill, copy
 - Min, max
- Wrappers
 - synchronized Collections, Lists, Sets, etc
 - unmodifiable Collections, Lists, Sets, etc

ArrayList to Array Conversion and vice versa

- **public Object[] toArray()**

```
List<String> list = new ArrayList();  
list.add(" Journal of ");  
list.add("Science");  
list.add("and Technology");  
list.add("on Information and Communications");  
    //convert listString tới array.  
String[] array = list.toArray(new  
    String[list.size()]);  
System.out.println("\n "+Arrays.toString(array));
```

- **Arrays.asList(T... a)**

```
List<String> names = Arrays.asList("John", "Peter",  
    "Tom", "Mary");  
System.out.println(names);
```

Stream Collectors groupingBy()

- Java 8 now directly allows you to do GROUP BY in Java by using `Collectors.groupingBy()` method.
- Example

```
public class Document {  
    private String code,publisher;  
    private int num;  
    //constructor  
    //getter and setter  
    //.....  
}  
List<Document> list;
```

count and sum

```
public void count() {  
    Map<String, Long> count = list.stream().collect(  
        Collectors.groupingBy(Document::getPublisher, Collect  
            ors.counting()) );  
    System.out.println(count);  
}
```

```
public void sum() {  
    Map<String, Integer> sum = list.stream().collect(  
        Collectors.groupingBy(Document::getPublisher,  
            Collectors.summingInt(Document::getNum)) );  
    System.out.println(sum);  
}
```

max and min

```
public void max() {  
    Optional<Document> max =  
list.stream().collect(Collectors.maxBy(  
Comparator.comparing(Document::getNum)));  
    System.out.println("Document with max  
Number:"+(max.isPresent()? max.get(): "Not  
Applicable"));  
}
```

Max in group

```
public void maxGroup() {  
    Map<String, Document> o =  
list.stream().collect(Collectors.groupingBy(Docu  
ment::getPublisher,  
Collectors.collectingAndThen(  
    Collectors.reducing((Document d1,  
Document d2) -> d1.getNum() > d2.getNum() ? d1 :  
d2), Optional::get)));  
    System.out.println(o);  
}
```

Summary

- The Collections Framework
 - The *Collection* Super interface and Iteration
 - Lists
 - Sets
 - Maps
- Generic in java

Case study

```
public static void main(String[] args) {  
    ListVehicle a=new ListVehicle();  
    Scanner in=new Scanner(System.in);  
    while(true) {  
        System.out.print("\n 1. input a Car");  
        System.out.print("\n 2. input a Motor");  
        System.out.print("\n 3. input a Truck");  
        System.out.print("\n 4. output a list of Vehicles");  
        System.out.print("\n 5. Search by manufacturer");  
        System.out.print("\n 6. Search by manufacturer ()");  
        System.out.print("\n 7. Search by cost (greater than)");  
        System.out.print("\n 8. Search by cost (from to)");  
        System.out.print("\n 9. Sort by manufacturer");  
        System.out.print("\n 10. Sort by type of engine");  
        System.out.print("\n 0. Exit");  
        System.out.print("\n Your choice (0->10): ");  
        int choice;
```