Object-oriented Design, Refactoring & Testing

Lecture 2

Java 2 Collections

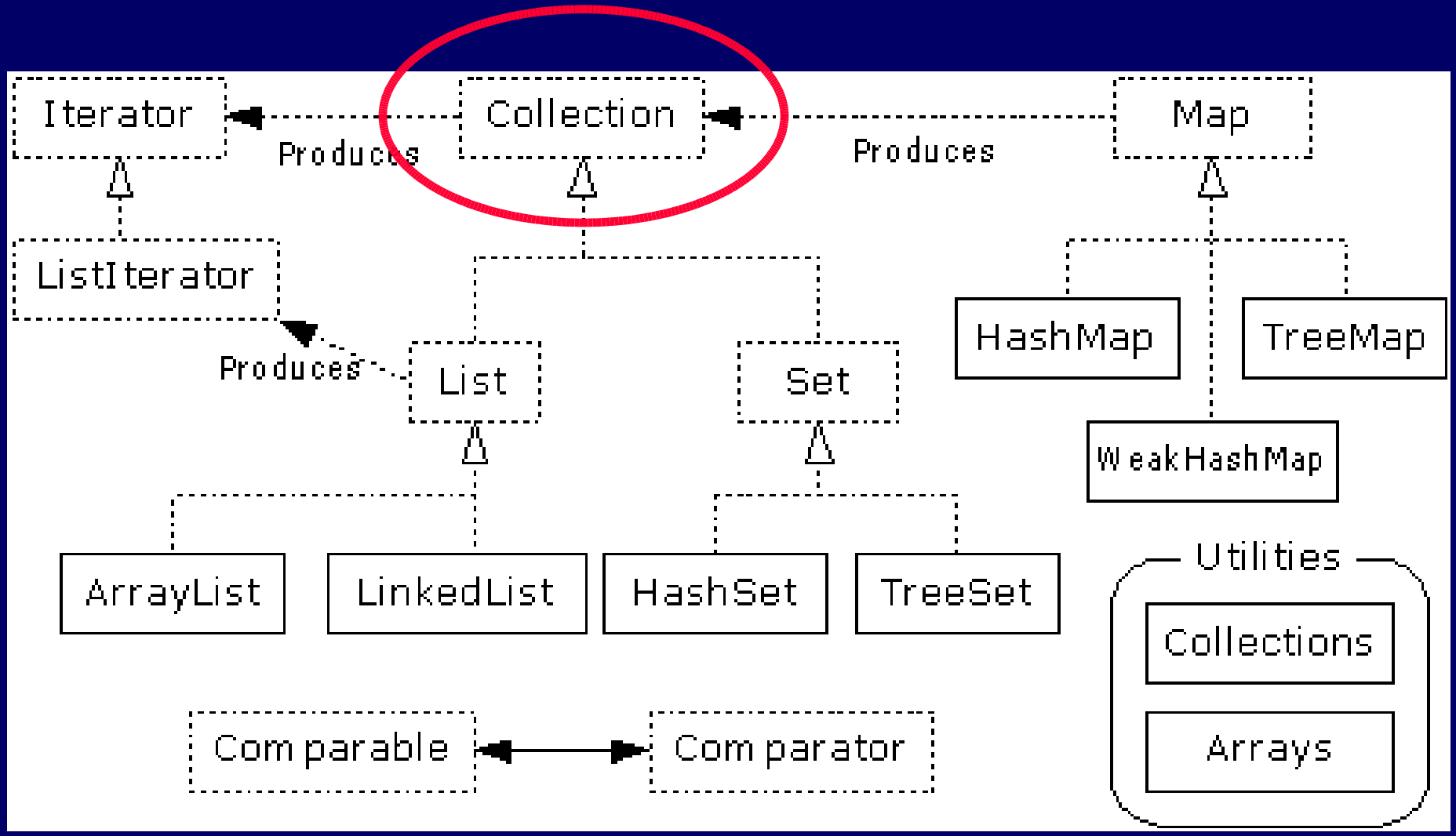
Bartosz Walter

<Bartek.Walter@man.poznan.pl>

Agenda

1. Basic interfaces: Collection, Set, List, Map
2. Iterators
3. Utility classes for Collections and Arrays
4. Special implementations
5. Comparing objects

Java 2 Collections



java.util.Collection

- Manages a group of ***elements***
- Least common denominator that all collections implement
- Imposes no constraints on the elements
- No direct implementation

General constraints on implementations

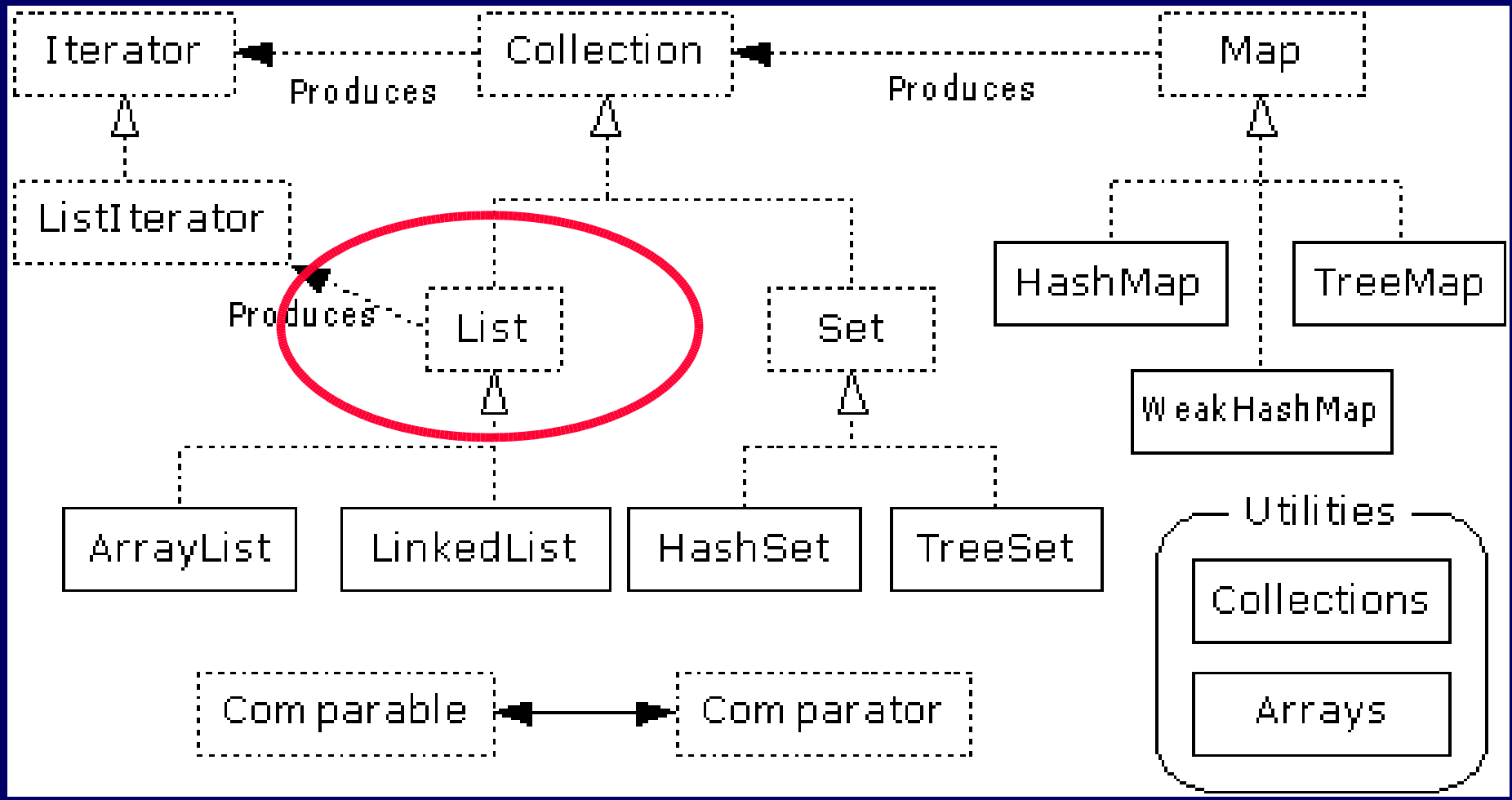
- Two obligatory constructors:
  - parameterless `Collection()`
  - Collection-based `Collection(Collection)`

java.util.Collection

Groups of methods:

- **basic operations** – adding, removing, measuring
  - `boolean add(Object)`
  - `boolean remove(Object)`
  - `boolean contains(Object)`
  - `boolean isEmpty()`
  - `Iterator iterator()`
- **bulk operations** – operations on groups of elements
  - `boolean addAll(Collection)`
  - `boolean removeAll(Collection)`
  - `boolean containsAll(Object)`
  - `boolean retainAll()`
  - `void clear()`
- **array operations** – conversion to arrays

Java 2 Collections



java.util.List

- Subinterface of Collection
- Manages a group of ***ordered elements***
- Puts some constraints on derivatives
- Direct implementations: `ArrayList, Vector, LinkedList`

General constraints on implementations

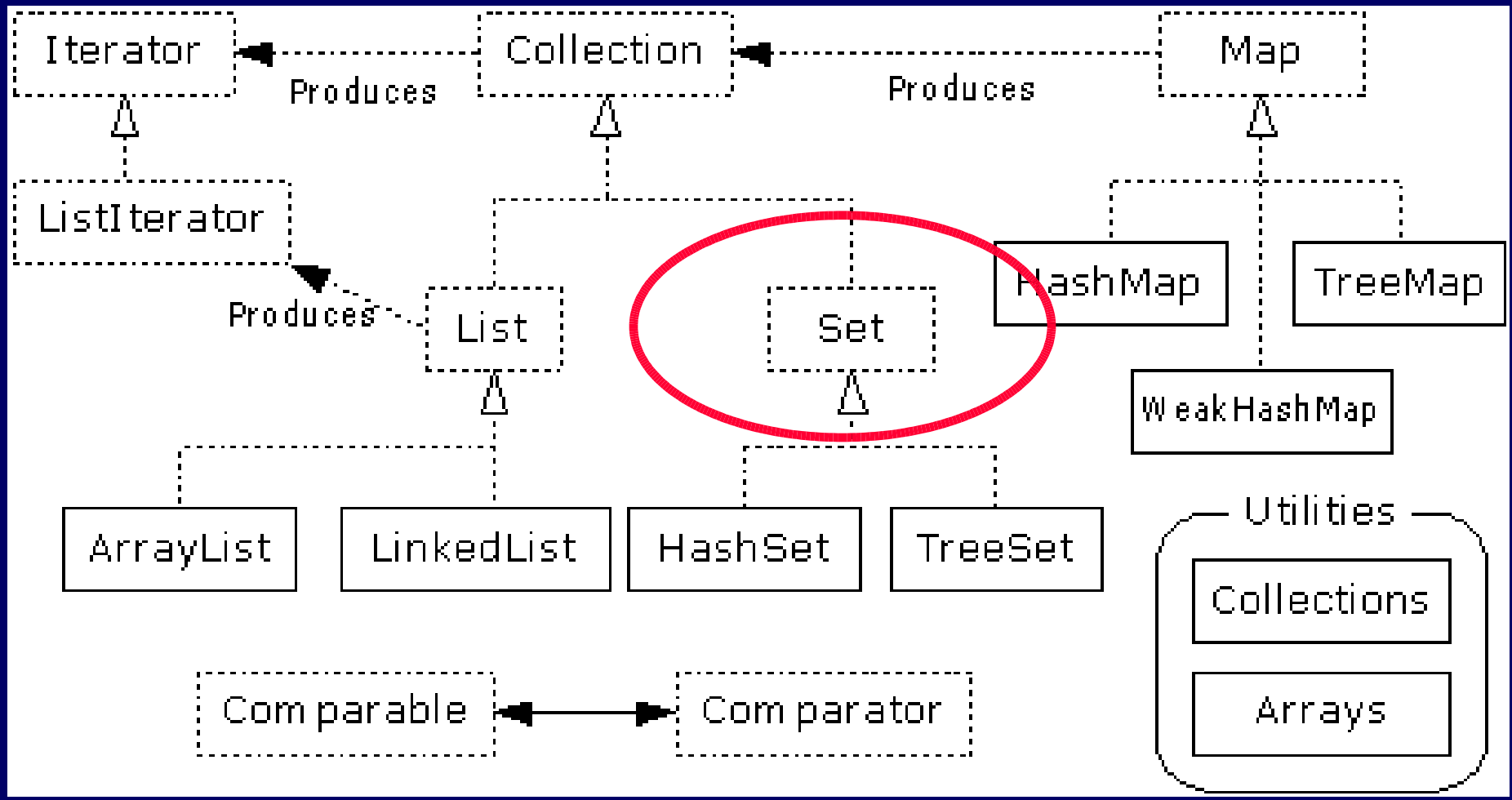
- Positional Access: manipulate elements based on their numerical position in the list.
- Search: search for a specified object in the list and return its numerical position.
- List Iteration: extend Iterator semantics to take advantage of the list's sequential nature.
- Range-view: perform arbitrary *range operations* on the list.

java.util.List

Groups of methods:

- **positional access**
  - `Object get(int)`
  - `Object set(int, Object)`
  - `Object add(int, Object)`
  - `Object remove(int)`
- **search**
  - `int indexOf(Object)`
  - `int lastIndexOf(Object)`
- **iteration**
  - `ListIterator listIterator()`
  - `ListIterator listIterator(int)`
- **range view**
  - `List subList(int, int)`

Java 2 Collections

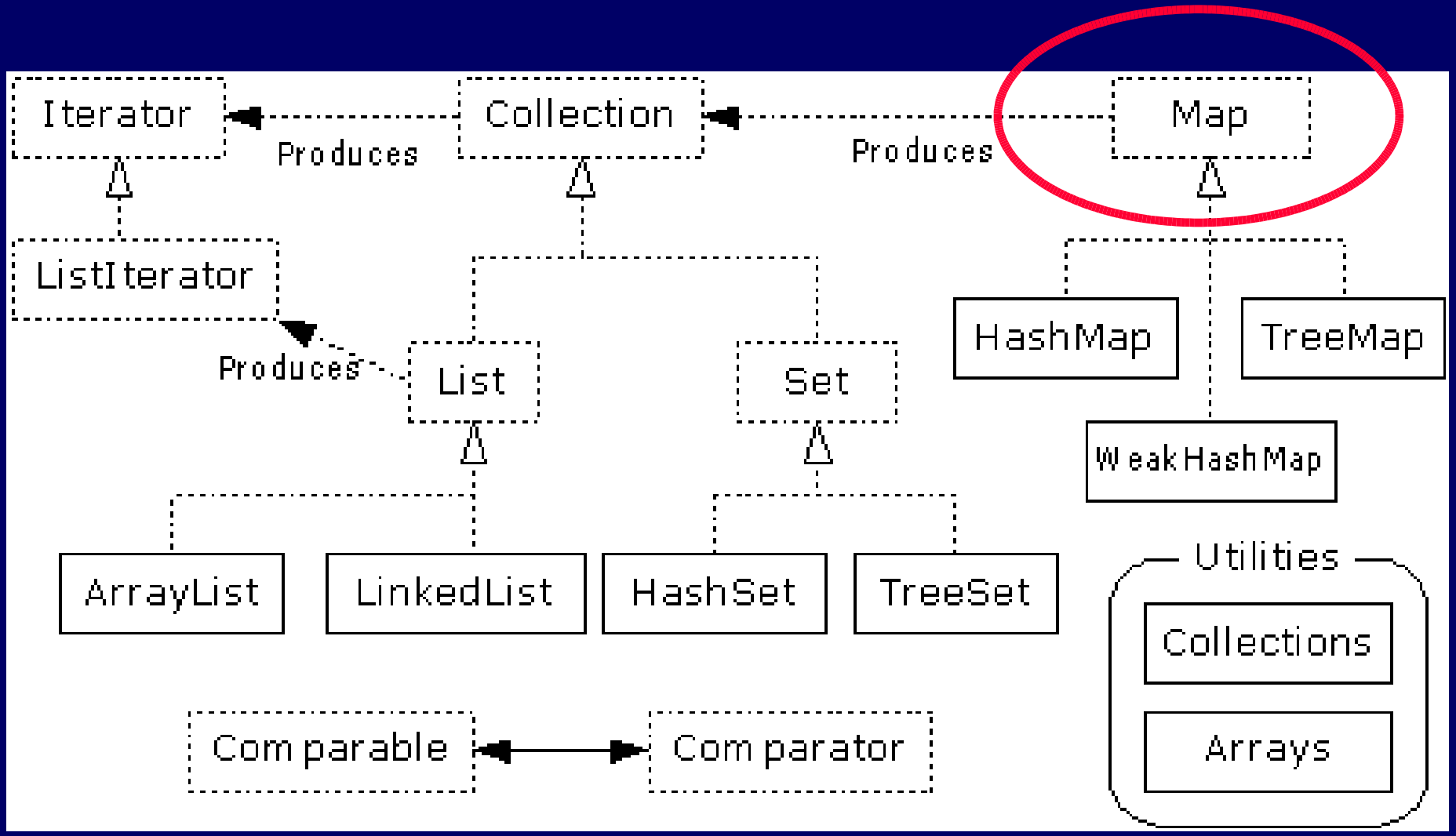


java.util.Set

- Subinterface of Collection, but no new methods (!)
- Manages a group of ***unique elements***
- Again, puts some constraints on derivatives
- Direct implementations: `HashSet, TreeSet`

```
public class FindDups {
    public static void main(String args[]) {
        Set s = new HashSet();
        for (int i=0; i<args.length; i++)
            if (!s.add(args[i]))
                System.out.println("Duplicate");
    }
}
```

Java 2 Collections



java.util.Map

- Stores pairs, not single objects
- Maps ***keys*** to ***values***
- Provides Collection views for keys, values and pairs
- Cannot contain duplicate keys
- Implementations: `HashMap, TreeMap, SortedMap`

General constraints on implementations

- Two obligatory constructors:
  - parameterless `Map()`
  - Map-based `Map(Map)`

java.util.Map

Groups of methods:

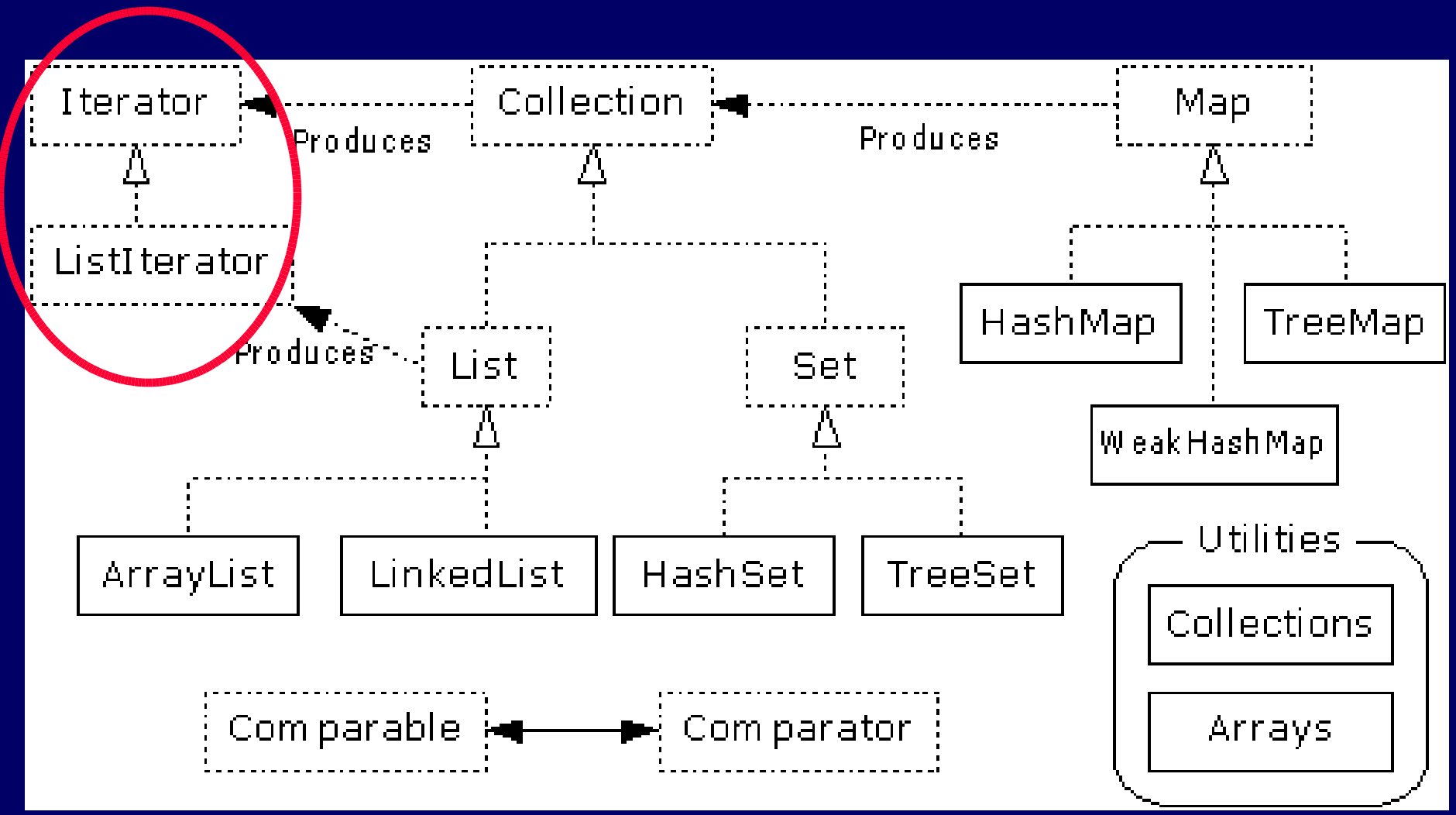
- **basic operations**
  - `Object put(Object, Object)`
  - `Object get(Object)`
  - `Object remove(Object)`
  - `boolean containsKey(Object)`
  - `boolean containsValue(Object)`
  - `boolean isEmpty()`
  - `int size()`
- **bulk operations**
  - `void putAll(Map)`
  - `void clear()`
- **Collection views**
  - `Set keySet()`
  - `Collection values()`
  - `Set entrySet()`

Example

```
public class Freq {
   private static final Integer ONE = new Integer(1);
   public static void main(String args[]) {
      Map m = new TreeMap();
      for (int i=0; i<args.length; i++) {
         Integer freq = (Integer) m.get(args[i]);
         m.put(args[i], (freq==null ? ONE :
            new Integer(freq.intValue() + 1)));
      }
      System.out.println(m.size()+" distinct words:");
      System.out.println(m);
   }
}
```

```
> java Freq if it is to be it is up to me to delegate
8 distinct words:
{to=3, me=1, delegate=1, it=2, is=2, if=1, be=1, up=1}
```

Java 2 Collections



java.util.Map

- Stores pairs, not single objects
- Maps ***keys*** to ***values***
- Provides Collection views for keys, values and pairs
- Cannot contain duplicate keys
- Implementations: `HashMap, TreeMap, SortedMap`

General constraints on implementations

- Two obligatory constructors:
  - parameterless `Map()`
  - Map-based `Map(Map)`

java.util.Iterator

- Generates a serie of elements, one at a time
- Successive calls return successive elements
- Replacement for `java.util.Enumeration`
- No publicly available implementation (!)
- No public constructor (!)

- `boolean hasNext()`
- `Object next()`
- `void remove()`

Example

```
public class Freq {
   public static void main(String args[]) {
      Collection coll = new ArrayList();
      for (int i=0; i<args.length; i++) {
         coll.add(args[i]);
      }
      for (Iterator iter = coll.iterator(); iter.hasNext();)
         String elem = (String) iter.next();
         System.out.print(elem + " ; ");
      }
   }
}
```

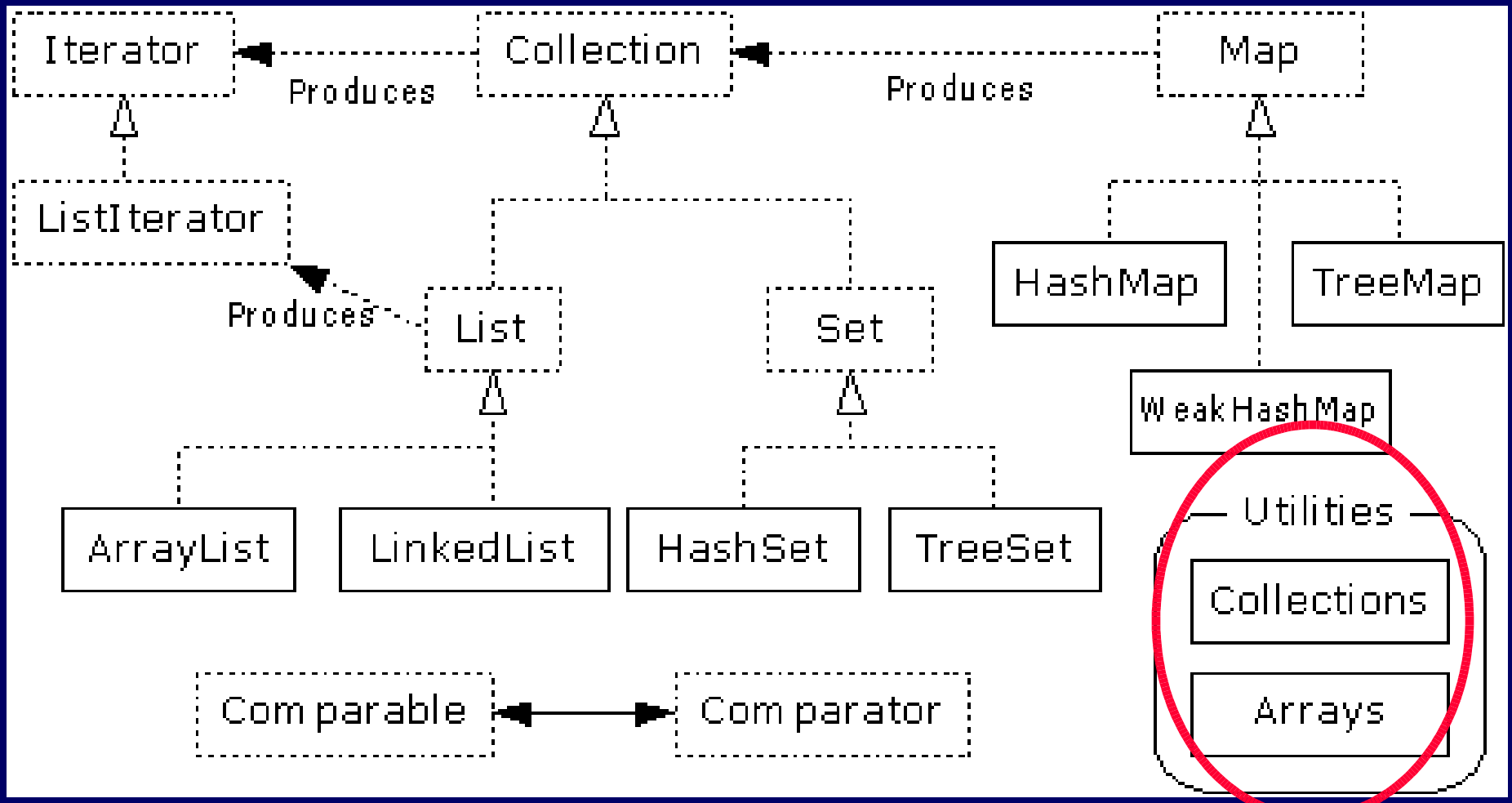
```
> java Freq if it is to be it is up to me to delegate
{if; it; is; be; up; is; me; delegate ...}
```

java.util.ListIterator

- Subinterface of `java.util.Iterator`
- Works in either direction
- No publicly available implementation (!)
- No public constructor (!)

```
void add(Object)
boolean hasPrevious ()
int nextIndex()
Object previous()
int previousIndex()
void set(Object)
```

Java 2 Collections



Utility classes: java.util.Collections

- Binary searching for specific object
- Sorting arbitrary List (*)
- Copying Lists
- Filling Collections with Objects
- Finding max and min elements (*)
- Reversing a List
- Shuffling a List
- Swapping two elements

Utility classes: java.util.Collections

- `int binarySearch(List, Object)`
- `int binarySearch(List, Object, Comparator)`
- `void copy(List, List)`
- `Void fill(List, Object)`
- `Object max/min(Collection)`
- `Object max/min(Collection, Comparator)`
- `List nCopies(int, Object)`
- `void reverse(List)`
- `void shuffle(List)`
- `void sort(List)`
- `void sort(List, Comparator)`
- `void swap(List, int, int)`

Task

Suggest an implementation of multibag.
Bag is a map in that each key maps to multiple values.

Utility classes: java.util.Arrays

- Conversion to List
- Binary searching for specific object
- Equality of two arrays
- Filling Arrays with Objects
- Sorting Arrays

```
List asList(Object[])
int binarySearch(type[], type)
int binarySearch(Object[], Object, Comparator)
boolean equals(type[], type[])
void fill(type[], type)
void sort(type[], Comparator)
```

Wrapping objects: Immutables

- Make Collection immutable
- Block any mutating methods
- No publicly available implementation (!)
- No public constructor (!)

- `Collection unmodifiableCollection(Collection)`
- `Set unmodifiableSet(Set)`
- `List unmodifiableList(List)`
- `Map unmodifiableMap(Map)`

Wrapping objects: Synchronization

- Make Collection synchronized
- Block any mutating methods
- No publicly available implementation (!)
- No public constructor (!)

- `Collection synchronizedCollection(Collection)`
- `Set synchronizedSet(Set)`
- `List synchronizedList(List)`
- `Map synchronizedMap(Map)`

Wrapping objects: Singletons

- Return collection of one specific element
- The collection is immutable
- No publicly available implementation (!)
- No public constructor (!)

Remove all instances of element e

- `c.removeAll(Collections.singleton(e));`

Remove all Lawyers from the map:

-
```
profession.values().removeAll(
    Collections.singleton(LAWYER));
```

Wrapping objects: Empties

- Return empty collection
- The collection is immutable

- `Set EMPTY_SET`
- `Collection EMPTY_COLLECTION`
- `Map EMPTY_MAP`

Comparing objects: java.lang.Comparable

- Sorting, searching requires a method of comparing objects
- Comparison can be extracted and plugged as a parameter
- Comparable objects can be compared to other

Sort a List of people by birthday:

- `Collections.sort(people);`

```
public class Person implements Comparable {
   private name, givenName;
   // ...
   public int compareTo(Object obj) { }
   public boolean equals(Object obj) { }
   public int hashCode() {}
}
```

Comparing objects: java.util.Comparator

- Sorting, searching requires a method of comparing objects
- Comparison can be extracted and plugged as a parameter
- Comparators allow to compare two objects

Sort a List of people by birthday:

- `Collections.sort(people, comparator);`

```
public class Name implements Comparator {
   public int compare (Object obj1, Object obj2)
   public boolean equals(Object obj)
}
```

Task

Write two implementations:

- **using Comparable interface**
- **using Comparators.**

Q&A