Generics (Parametric Polymorphism)

Generics

- Java **Generic** methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration.

- Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

- We might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

Generic Methods

- Write a single generic method declaration that can be called with arguments of different types.
- Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately.

Generic Methods

```
public class TestGenerics4{

   public static < E > void printArray(E[] elements)
   {
        for ( E element : elements)
         {
            System.out.println(element );
         }
}
    public static void main( String args[] )
    {
        Integer[] intArray = { 10, 20, 30, 40, 50 };
        Character[] charArray = { 'J', 'A', 'V', 'A' };

        System.out.println( "Printing Integer Array" );
        printArray( intArray  );

       System.out.println( "Printing Character Array" );
        printArray( charArray );
    }
}
```

Generic Methods

Output:

Printing Integer Array 10 20 30 40 50

Printing Character Array J A V A

Generic class

```
class Test<T>
{
  // An object of type T is declared
  T obj;
  Test(T obj) {  this.obj = obj;  }  // constructor
  public T getObject()  { return this.obj; }
}

// Driver class to test above
class TestDemo
{
  public static void main (String[] args)
  {
    // instance of Integer type
    Test <Integer> iObj = new Test<Integer>(15);
    System.out.println(iObj.getObject());

    // instance of String type
    Test <String> sObj =
             new Test<String>(“Hello");
    System.out.println(sObj.getObject());
  }
}
```

Generic class

Output:

```
15
Hello
```

A Generic Class with Two Type Parameters

```
class Test<T, U>
{
  T obj1;  // An object of type T
  U obj2;  // An object of type U

  // constructor
  Test(T obj1, U obj2)
  {
    this.obj1 = obj1;
    this.obj2 = obj2;
  }

  // To print objects of T and U
  public void print()
  {
    System.out.println(obj1);
    System.out.println(obj2);
  }
}

// Driver class to test above
class Main
{
  public static void main (String[] args)
  {
    Test <String, Integer> obj =
      new Test<String, Integer>("Hello", 15);

    obj.print();
  }
}
```

Bounded Types

- In generic classes the type parameters could be replaced by any class type.
- This is fine for many purposes, but sometimes it is useful to limit the types that can be passed to a type parameter.
- For example, assume that you want to create a generic class that contains a method that returns the average of an array of numbers.
- Furthermore, you want to use the class to obtain the average of an array of any type of number, including integers, floats, and doubles.
- Thus, you want to specify the type of the numbers generically, using a type parameter.

Bounded Types

// The class contains an error!

```
class Stats<T>
 {
        T[] nums;    // nums is an array of type T

       Stats(T[] o)
         {
             nums = o;
         }
      // Return type double in all cases.
        double average()
          {
               double sum = 0.0;
                for(int i=0; i < nums.length; i++)
                    sum += nums[i].doubleValue(); // Error!!!
                     return sum / nums.length;
           }
}
```

Bounded Types

- In **Stats**, the **average( )** method attempts to obtain the **double** version of each number in the **nums** array by calling **doubleValue( )**.
- Because all numeric classes, such as **Integer** and **Double**, are subclasses of **Number**, and **Number** defines the **doubleValue( )** method, this method is available to all numeric wrapper classes.
- The trouble is that the compiler has no way to know that you are intending to create **Stats** objects using only numeric types.
- Thus, when you try to compile **Stats**, an error is reported that indicates that the **doubleValue( )** method is unknown.
- To solve this problem, you need some way to tell the compiler that you intend to pass only numeric types to **T**.
- Furthermore, you need some way to *ensure* that *only* numeric types are actually passed.

Bounded Types

- To handle such situations, Java provides *bounded types.*
- When specifying a type parameter, you can create an upper bound that declares the superclass from which all type arguments
- **<*T* extends *superclass*>**
- This specifies that *T* can only be replaced by *superclass,* or subclasses of *superclass.*
- Thus, *superclass* defines an inclusive, upper limit.

Bounded Types

```
class Stats<T extends Number>
 {
        T[] nums;    // nums is an array of type T

       Stats(T[] o)
         {
            nums = o;
         }
      // Return type double in all cases.
         double average()
           {
                double sum = 0.0;
                 for(int i=0; i < nums.length; i++)
                       sum += nums[i].doubleValue();
                        return sum / nums.length;
           }
}
```

Bounded Types

```
class BoundsDemo {
public static void main(String args[])
  {
        Integer inums[] = { 1, 2, 3, 4, 5 };
        Stats<Integer> iob = new Stats<Integer>(inums);
                double v = iob.average();
                 System.out.println("iob average is " + v);

        Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
        Stats<Double> dob = new Stats<Double>(dnums);
                double w = dob.average();
                System.out.println("dob average is " + w);
// This won't compile because String is not a subclass of Number.
       // String strs[] = { "1", "2", "3", "4", "5" };
       // Stats<String> strob = new Stats<String>(strs);
       // double x = strob.average();
       // System.out.println("strob average is " + v);
}
}
```

The output is shown here:
Average is 3.0
Average is 3.3

Java Collections

15

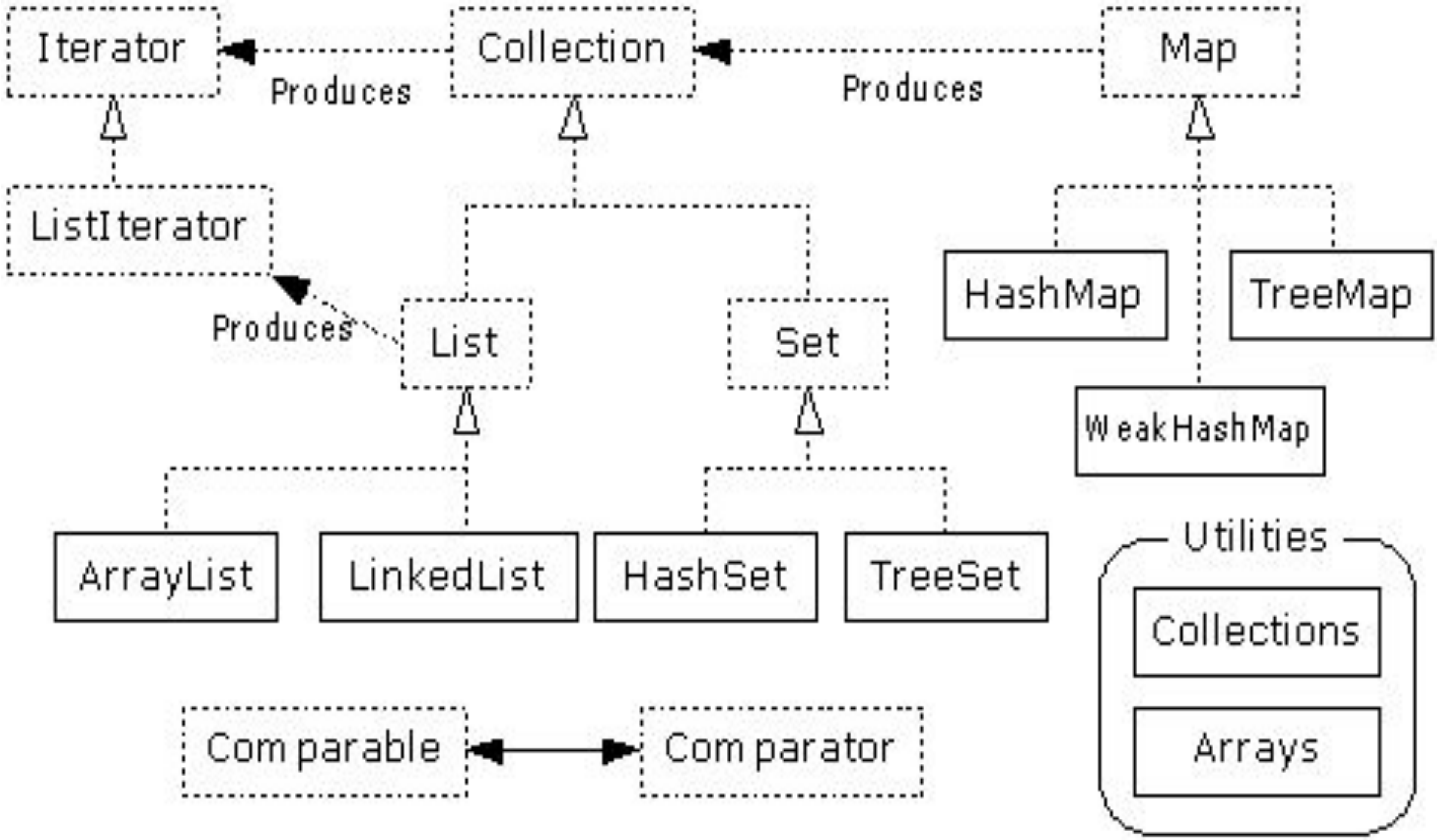
Java Collections

- **Collections in java** is a framework that provides an architecture to store and manipulate the group of objects.
- A collection is an object that groups multiple elements into a single unit
- Very useful
  - store, retrieve and manipulate data
  - transmit data from one method to another

Collections Framework

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

Collections Framework



Collection Interface

- Defines fundamental methods
  - `int size();`
  - `boolean isEmpty();`
  - `boolean contains(Object element);`
  - `boolean add(Object element);`
  - `boolean remove(Object element);`
  - `Iterator iterator();`
- These methods are enough to define the basic behavior of a collection
- Provides an Iterator to step through the elements in the Collection

Iterator Interface

- 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

Iterator Position

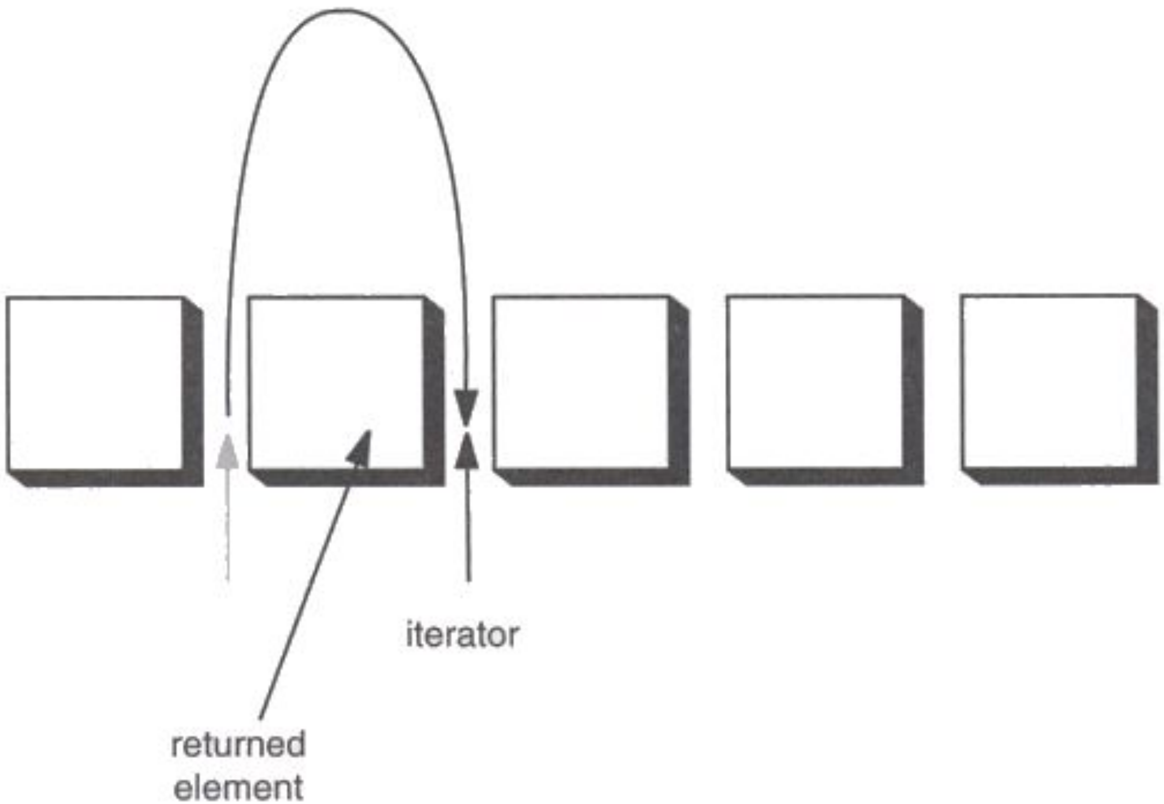
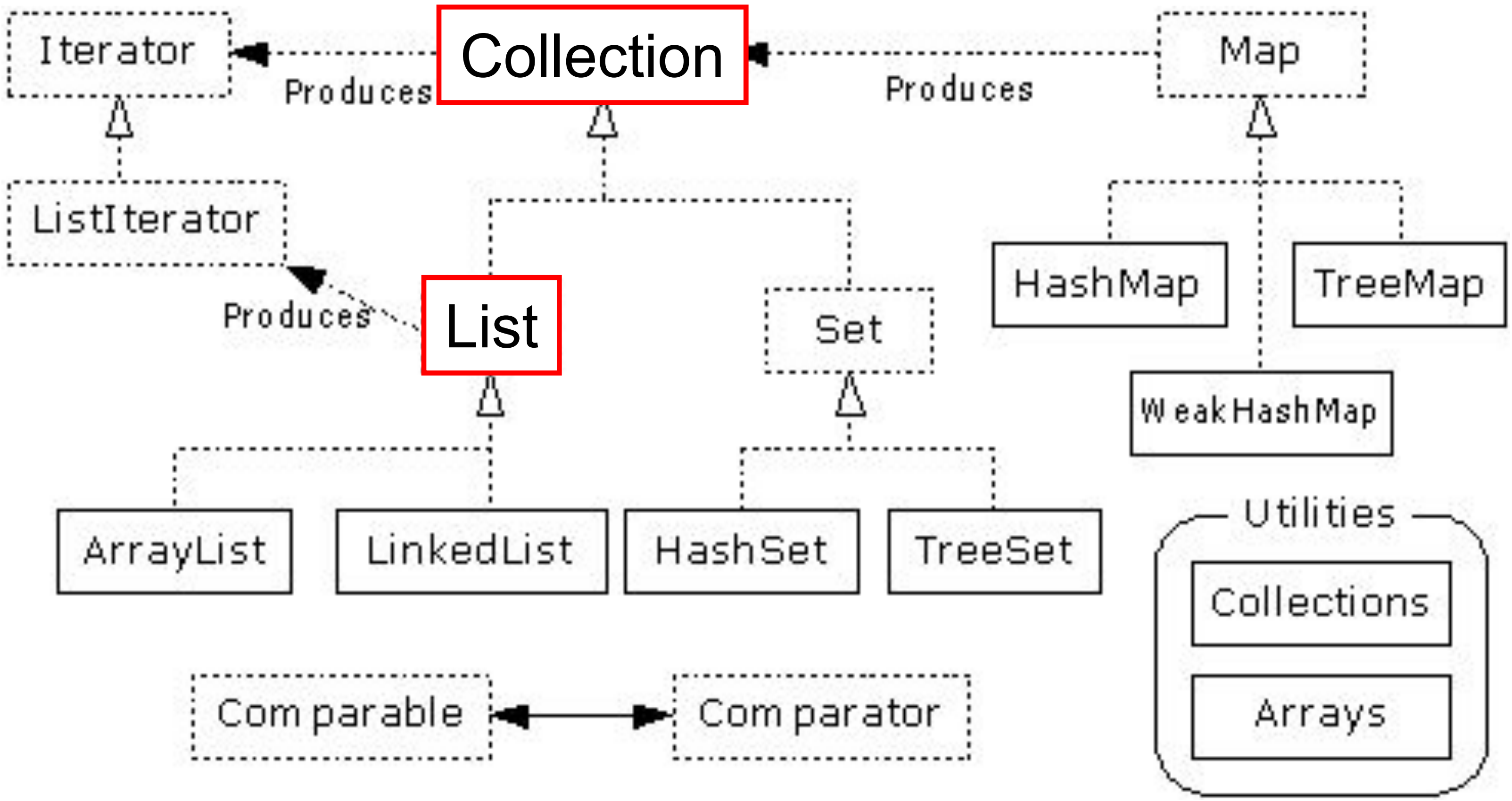


Figure 2–3: Advancing an iterator

List Interface



List Implementations

- **ArrayList (class)**
  - low cost random access
  - high cost insert and delete
  - array that resizes if need be
- **LinkedList (Class)**
  - sequential access
  - low cost insert and delete
  - high cost random access

ArrayList methods

- The indexed get and set methods of the List interface are appropriate to use since ArrayLists are backed by an array
  - `Object get(int index)`
  - `Object set(int index, Object element)`
- Indexed add and remove are provided, but can be costly if used frequently
  - `void add(int index, Object element)`
  - `Object remove(int index)`
- May want to resize in one shot if adding many elements
  - `void ensureCapacity(int minCapacity)`

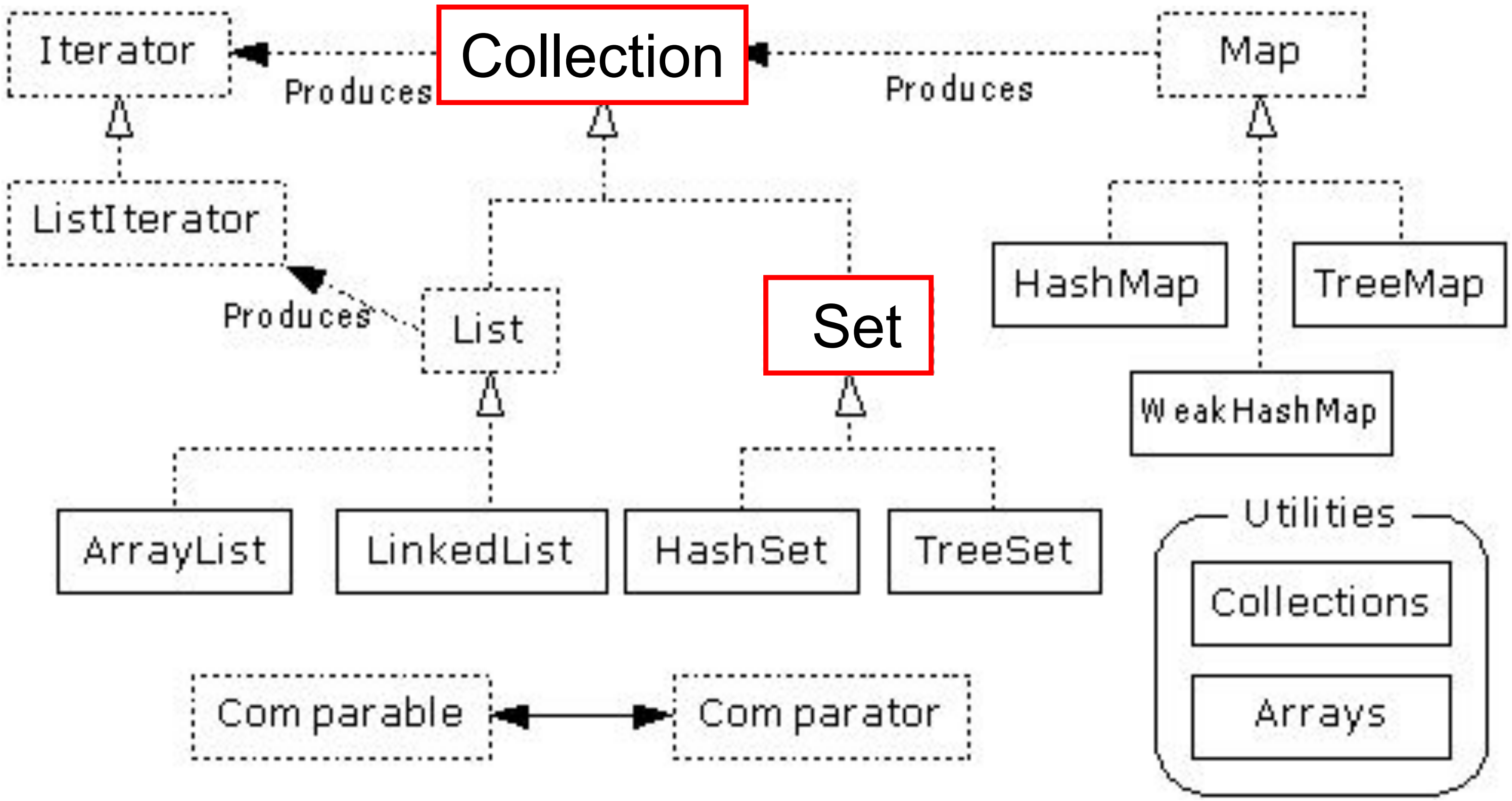
LinkedList overview

- 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

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()`

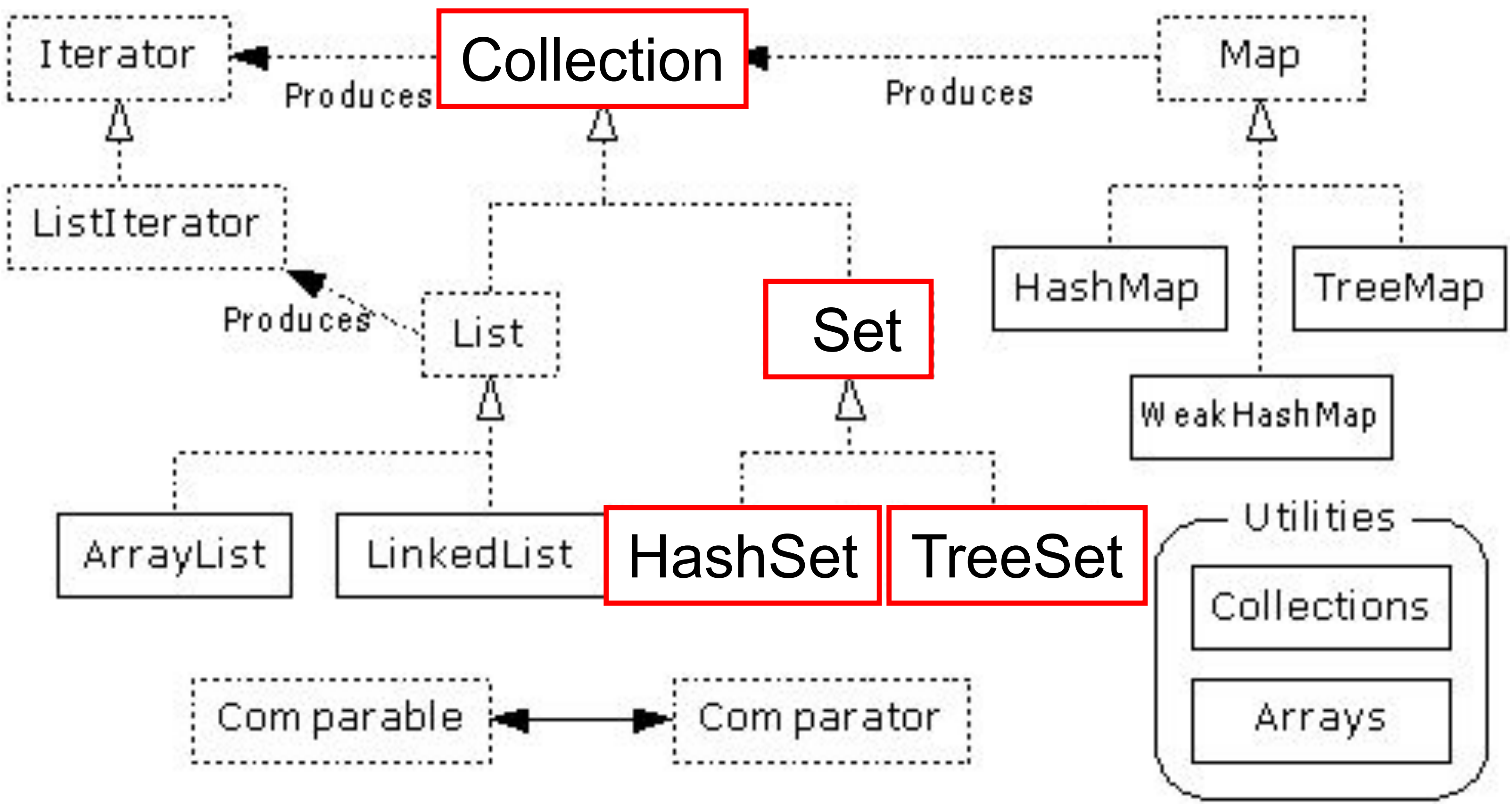
Set Interface Context



Set Interface

- Same methods as Collection
  - different contract - no duplicate entries
- Defines two fundamental methods
  - `boolean add(Object o)` - reject duplicates
  - `Iterator iterator()`
- Provides an Iterator to step through the elements in the Set
  - No guaranteed order in the basic Set interface
  - There is a SortedSet interface that extends Set

HashSet and TreeSet Context



29

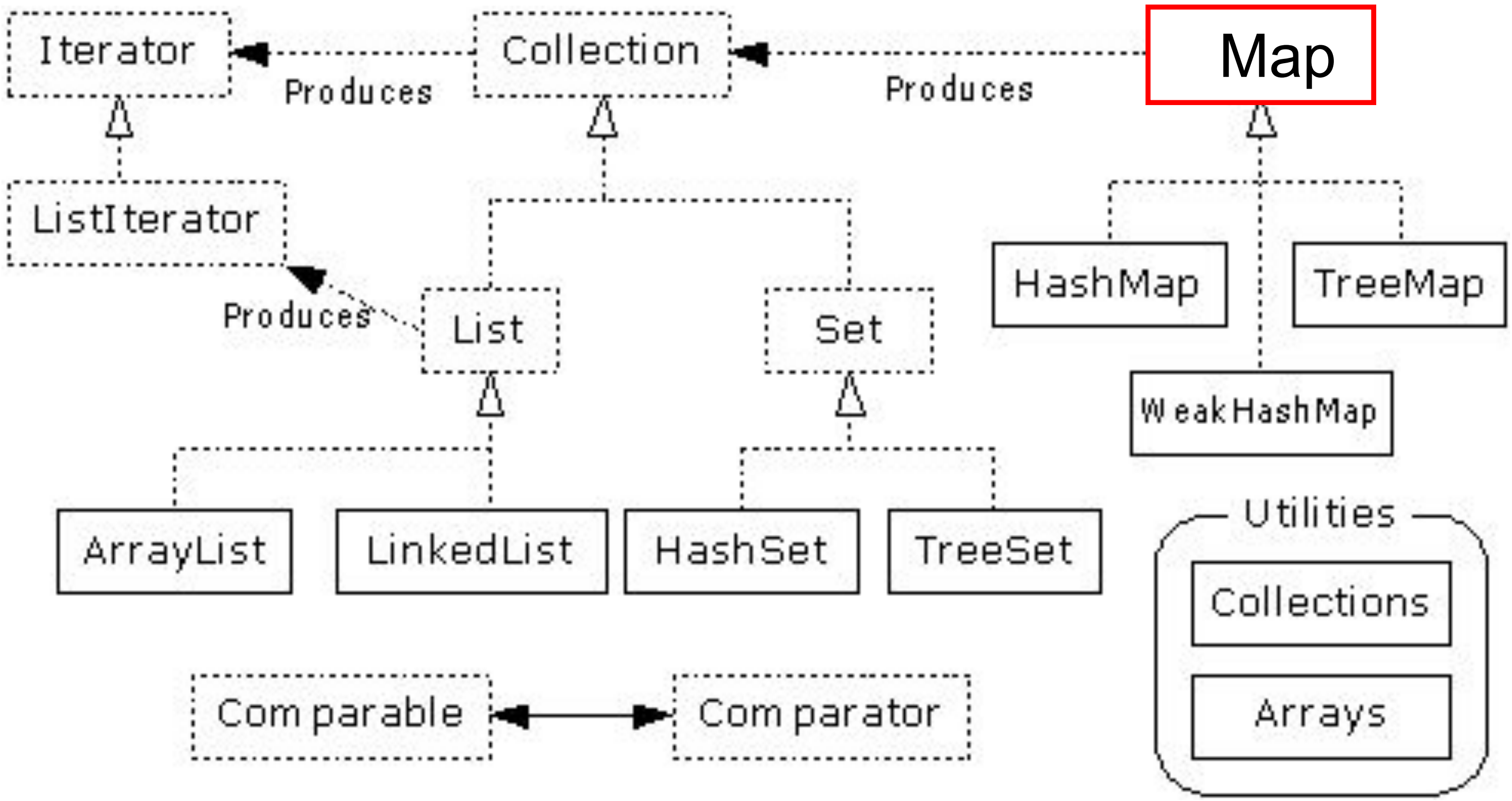
HashSet

- Find and add elements very quickly
  - uses hashing implementation in HashMap
- Hashing uses an array of linked lists
  - The **`hashCode()`** is used to index into the array
  - Then **`equals()`** is used to determine if element is in the (short) list of elements at that index
- No order imposed on elements
- The **`hashCode()`** method and the **`equals()`** method must be compatible
  - if two objects are equal, they must have the same **`hashCode()`** value

TreeSet

- Elements can be inserted in any order
- The TreeSet stores them in order
  - Red-Black Trees out of Cormen-Leiserson-Rivest
- An iterator always presents them in order
- Default order is defined by natural order
  - objects implement the Comparable interface
  - TreeSet uses **`compareTo(Object o)`** to sort
- Can use a different Comparator
  - provide Comparator to the TreeSet constructor

Map Interface Context



Map Interface

- Stores key/value pairs
- Maps from the key to the value
- Keys are unique
  - a single key only appears once in the Map
  - a key can map to only one value
- Values do not have to be unique

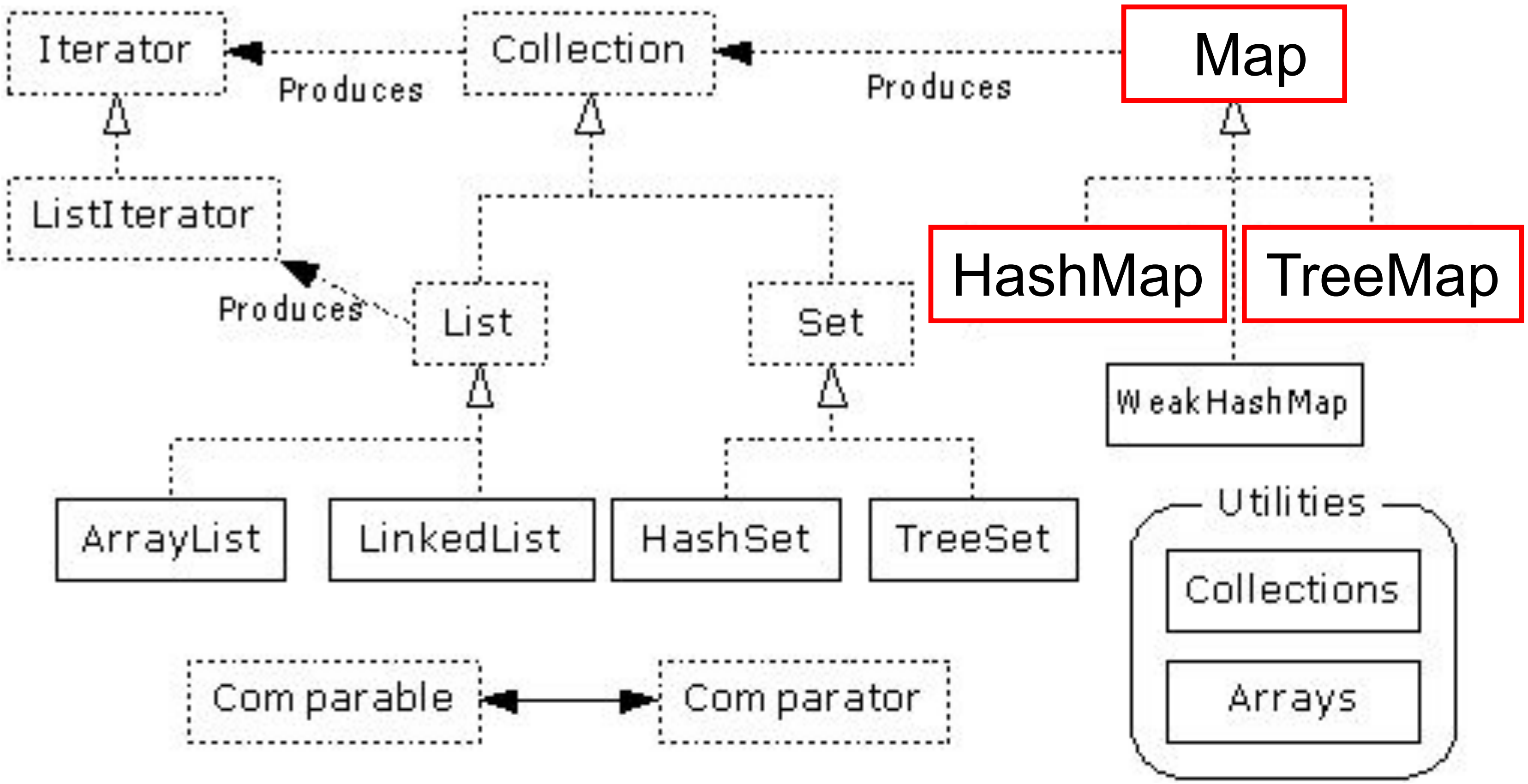
Map methods

- `Object put(Object key, Object value)`
- `Object get(Object key)`
- `Object remove(Object key)`
- `boolean containsKey(Object key)`
- `boolean containsValue(Object value)`
- `int size()`
- `boolean isEmpty()`

Map views

- A means of iterating over the keys and values in a Map
- **`Set keySet()`**
  - returns the Set of keys contained in the Map
- **`Collection values()`**
  - returns the Collection of values contained in the Map. This Collection is not a Set, as multiple keys can map to the same value.
- **`Set entrySet()`**
  - returns the Set of key-value pairs contained in the Map. The Map interface provides a small nested interface called Map.Entry that is the type of the elements in this Set.

HashMap and TreeMap



HashMap and TreeMap

- HashMap
  - The keys are a set - unique, unordered
  - Fast

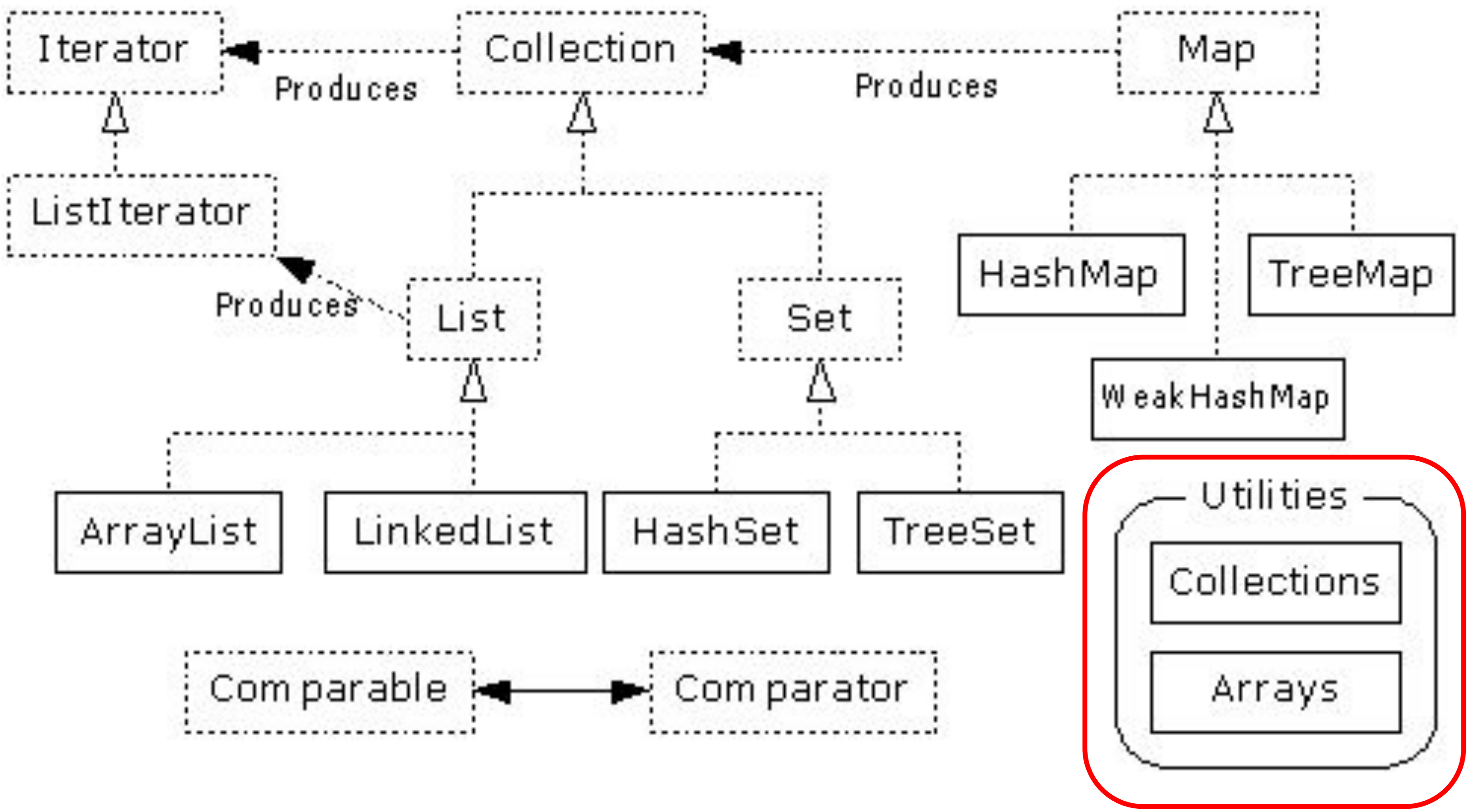
- TreeMap
  - The keys are a set - unique, ordered
  - Same options for ordering as a TreeSet
    - *Natural order (Comparable, compareTo(Object))*
    - *Special order (Comparator, compare(Object, Object))*

Bulk Operations

- In addition to the basic operations, a Collection may provide “bulk” operations

```
boolean containsAll(Collection c);
boolean addAll(Collection c);    // Optional
boolean removeAll(Collection c); // Optional
boolean retainAll(Collection c); // Optional
void clear();                    // Optional
Object[] toArray();
Object[] toArray(Object a[]);
```

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

Appendix

Legacy classes

- Still available
- Don't use for new development
  - unless you have to, eg, J2ME, J2EE in some cases
- Hashtable
  - use HashMap
- Enumeration
  - use Collections and Iterators
  - if needed, can get an Enumeration with Collections.enumeration(Collection c)

More Legacy classes

- Vector
  - use ArrayList
- Stack
  - use LinkedList
- BitSet
  - use ArrayList of boolean, unless you can’t stand the thought of the wasted space
- Properties
  - legacies are sometimes hard to walk away from …
  - see next few pages