Collections in Java

- Arrays
  - Has special language support
- Iterators
  - **`Iterator`**
- Collections (also called containers)
  - **`Collection`**
  - **`Set`**
  - **`List`**
  - **`Map`**
- The composite design pattern, revisited

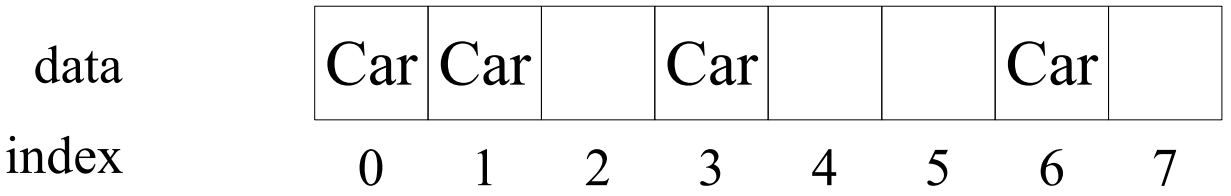
Array, Example

```
class Car{};                        // minimal dummy class
Car[] cars1;                        // null reference
Car[] cars2 = new Car[10];          // null references

for (int i = 0; i < cars2.length; i++)
   cars2[i] = new Car();

// aggregated initialization
Car[] cars3 = {new Car(), new Car(), new Car(), new Car()};
cars1 = cars3;
```

- Helper class **`java.util.Arrays`**
  - Search and sort: **`binarySearch(), sort()`**
  - Comparison: **`equals()`**
  - Instantiation: **`fill()`**
  - Conversion: **`asList()`**
  - All static methods

Array

- Most efficient way to hold references to objects.



- Advantages
  - Knows the type it holds, i.e., compile-time type checking
  - Knows its size, i.e., ask for the length
  - Can hold primitive types directly
  - Automatic boundary checking, raises exceptions if violated
- Disadvantages
  - Only hold one type of objects (including primitives)
  - Fixed size

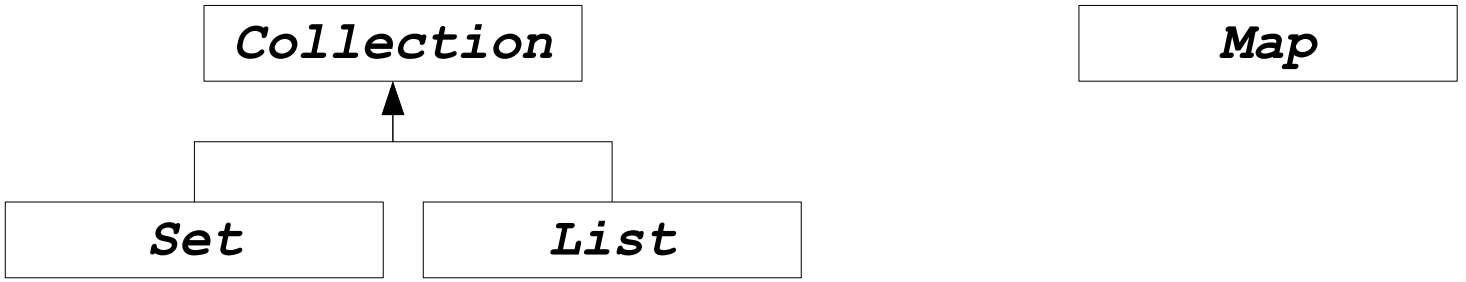
Overview of Collection

- A *collection* is a group of data manipulated as a single object. Corresponds to a *bag*.

- Insulate client programs from the implementation.
  - array, linked list, hash table, balanced binary tree, red-black tree, etc.
- Can grow dynamically
- Contain only reference types (subclasses of **`Object`**)
  - i.e., no primitive data types, in Java 5.0 (autoboxing/unboxing)
- Heterogeneous
- Can be thread safe (concurrent access)
- Can be not-modifiable
- Like C++'s Standard Template Library (STL)

Collection Interfaces

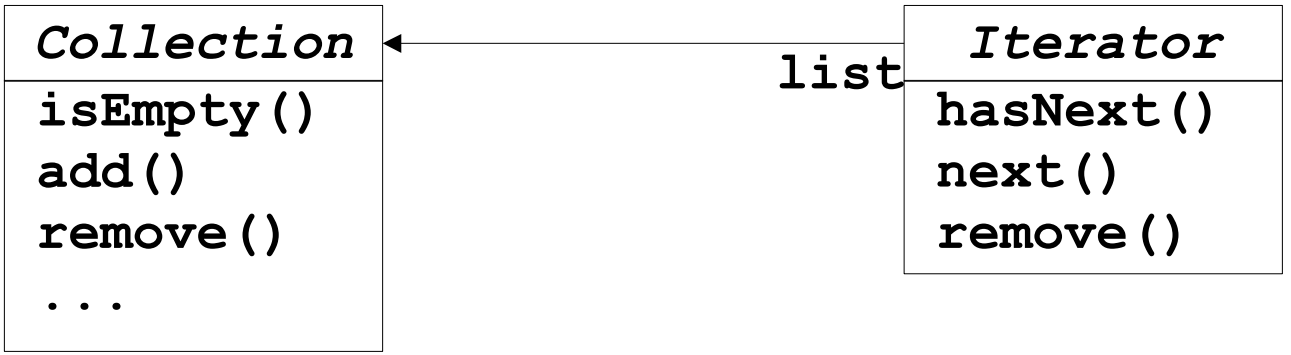
- Collections are primarily defined through a set of interfaces.
  - Supported by a set of classes that implement the interfaces



- Interfaces are used of flexibility reasons
  - Not tightened to an implementation

The `Iterator` Interface

- *The idea*: Select each element in a collection in turns
  - Hide the underlying collection can be bag, set, list, or map



- Another nice design pattern.
- Iterators are *fail-fast*
  - Exception thrown if collection is modified externally, i.e., not via the iterator (multi-threading).

The `Iterator` Interface, cont.

```
// the interface definition
interface Iterator<E> {
   boolean hasNext();
   E next();                  // note "one-way" traffic
   void remove();             // Optional
}
// an example using iterators
public static void main (String[] args){
   ArrayList<Car> cars = new ArrayList<Car>();
   for (int i = 0; i < 12; i++)
      cars.add(new Car());

   Iterator it = cars.iterator();
   while (it.hasNext())
      System.out.println (it.next());

   for (Car c : cars) // the same Java 5.0
      System.out.println(c);

}
```

The `Iterable` Interface

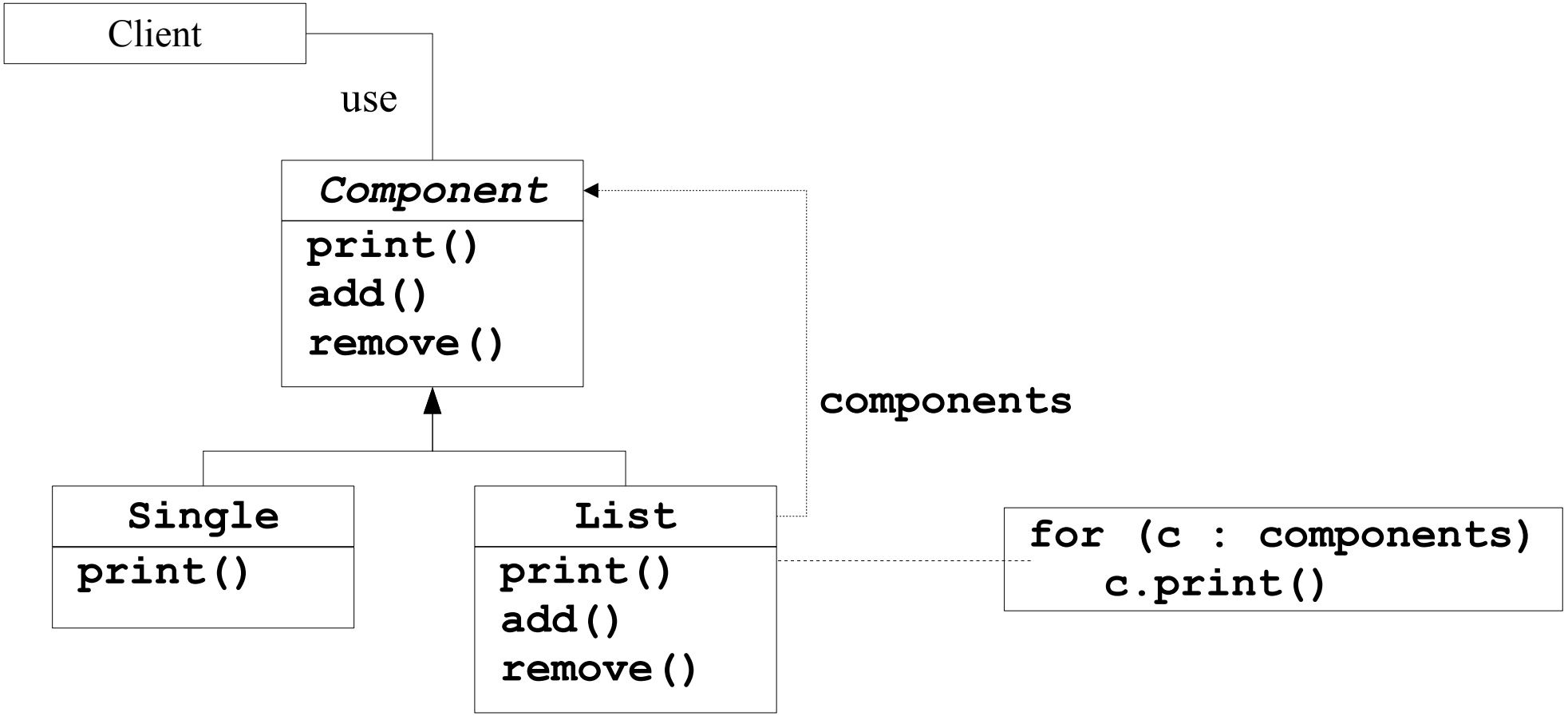
- Other interface closely related to the **`Iterator`** interface

```
// the interface definition
interface Iterable<T> {
   Iterator<T> iterator();
}
```

The `Collection` Interface

```
public interface Collection<E> extends Iterable<E> {
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);    // Optional
    boolean remove(Object element); // Optional
    Iterator<E> iterator();

    // Bulk Operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();

    // Array Operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

The Composite Design Pattern, revisited



- **`Component`** is an abstract class
- Single typically called *leaf*
- List typically called *composite*

Implementation of the Composite Pattern

```
import java.util.*;
public class List extends Component{
    private ArrayList<Component> comp; // prev. array + int counter

    /** Constructor */
    public List(){
        comp = new ArrayList<Component>(); // variable size
    }
    /** Print */
    public void print(){
        for (Component c : comp) { // previously "old" for
            c.print();
        }
    }
    /** Add */
    public void add(Component c){
        comp.add(c);
    }
    /** Remove */
    public void remove(Component c){
        comp.remove(c); // previously not implemented
    }
}
```

Evaluation of the Composite Design Pattern

- Made **`List`** and **`Single`** classes look alike when printing from the client's point of view.
  - The main objective!
- Can make instances of **`Component`** class, not the intension
  - Can call dummy add/remove methods on these instances (FIXED using abstract class)
- Can call add/remove method of **`Single`** objects, not the intension. (CANNOT BE FIXED).
- Fixed length, not the intension. (FIXED using collection instead of array)
- Added remove method very simple with collections.
- Nice design! And now complete!

The Collection Annoyance, Java 1.4

```
import java.util.*;
// skeleton classes
class Apple { private int weight; }
class Banana{ private int length;}


public class CollectionAnnoyance{
    public static void main(String[] args){
        Collection c = new ArrayList(); // 5.0 raw type
        for(int i = 0; i < 10; i++){
            c.add(new Apple(i)); // note up-cast
        }
        c.add(new Banana()); // ups
        // c.add(c); // ups ups do not try this at home!

        Iterator i = c.iterator();
        while (i.hasNext()){
            Apple a = (Apple) i.next(); // note down cast
        }
    }
}
```

The `Set` Interface

- Corresponds to the mathematical definition of a set (no duplicates are allowed).

$$\{e_1, e_2, \ldots, e_n\}$$

$$\{7, 3, 6, 4\}$$

$$\{\text{“Holland”}, \text{“China”}, \text{“Brazil”}\}$$

- Compared to the `Collection` interface
  - Interface is identical.
  - Every constructor must create a collection without duplicates.
  - The operation `add` cannot add an element already in the set.
  - The method call `set1.equals(set2)` works at follows
    - set1 $\subseteq$ set2, and set2 $\subseteq$ set1

`Set` Idioms

- set1 ∪ set2
  - **`set1.addAll(set2)`**
- set1 ∩ set2
  - **`set1.retainAll(set2)`**
- set1 – set2
  - **`set1.removeAll(set2)`**

`HashSet` and `TreeSet` Classes

- **`HashSet`** and **`TreeSet`** implement the **`Set`** interface

- **`HashSet`**
  - Implemented using a hash table
  - No ordering of elements
  - **`add`**, **`remove`**, and **`contains`** methods constant time complexity *O(c),* c = constant

- **`TreeSet`**
  - Implemented using a tree structure.
  - Guarantees ordering of elements.
  - **`add`**, **`remove`**, and **`contains`** methods logarithmic time complexity *O(log (n)),* *n* = number of elements in collection

`HashSet`, Example

```
// [Source: java.sun.com]
import java.util.HashSet;

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])) // note returns boolean
                System.out.println("Duplicate detected: " +
                                   args[i]);
        }
        System.out.print(s.size() +
                         " distinct words detected: ");
        System.out.println(s);
  }
}
```

The `List` Interface

- The **`List`** interface corresponds to an order bag of elements.

$$<e_1, e_2, \ldots, e_n>$$

<3, 1, 7, 1>

<1, 1, 3, 7>

<"Holland", "China", "Brazil">

- Extensions compared to the **`Collection`** interface
  - Access to elements via indexes, like arrays
    - **`add(int, Object)`**, **`get(int)`**, **`remove(int)`**, **`set(int, Object)`** (note set = replace bad name for the method)
  - Search for elements
    - **`indexOf(Object)`**, **`lastIndexOf(Object)`**
  - Specialized **`Iterator`**, call **`ListIterator`**
  - Extraction of sublist
    - **`subList(int fromIndex, int toIndex)`**

The `List` Interface, cont.

Further requirements compared to the **`Collection`** Interface

- **`void add(E o)`**
  - adds at the end of the list
- **`boolean remove(Object)`**
  - removes the first occurrence, i.e, from the start of the list
- **`list1.equals(list2)`**
  - the ordering of the elements is taken into consideration
- Extra requirements to the method **`hashCode`**.
  - **`list1.equals(list2)`** implies that **`list1.hashCode()==list2.hashCode()`**

The `List` Interface, cont.

```
public interface List<E> extends Collection<E> {
    // Positional Access
    E get(int index);
    E set(int index, E element);
    void add(int index, E element);
    E remove(int index);
    boolean addAll(int index, Collection<? extends E> c);

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // Range-view
    List<E> subList(int from, int to);
}
```

`ArrayList` and `LinkedList` Classes

- Both implement the **`List`** interface.

- **`ArrayList`**
  - Array based implementation
  - Elements can be accessed directly via the **`get`** and **`set`** methods
  - Tip: use as default choice for a simple sequence

- **`LinkedList`**
  - Based on a double linked list
  - Gives better performance on **`add`** and **`remove`** compared to **`ArrayList`**
  - Gives poorer performance on **`get`** and **`set`** methods compared to **`ArrayList`**

The `java.util.Collections` Class

- binarySearch()
- copy()
- fill()
- max()
- min()
- reverse()
- rotate()
- shuffle()
- sort()
- swap()
- unmodifiableList()

`ArrayList`, Example

```
// [Source: java.sun.com]
import java.util.*;

public class Shuffle {
    public static void main(String[] args) {
        // - List<String> l = new ArrayList<String>()
        ArrayList<String> l = new ArrayList<String>();

        // put the arguments into the array list
        for (String s : args)
            l.add(s);
        Collections.shuffle(l, new Random());
        System.out.println(l);
    }
}
```

The `Map` Interface

- A Map is an object that maps keys to values. Also called an *associative array* or a *dictionary*.

$$\{k_1 \rightarrow v_1, k_2 \rightarrow v_2, \ldots, k_n \rightarrow v_n\}$$

- Methods for adding and deleting
  - `put(Object key, Object value)`
  - `remove (Object key)`
- Methods for extraction objects
  - `get (Object key)`
- Methods to retrieve the keys, the values, and (key, value) pairs
  - `keySet()  // returns a Set`
  - `values()  // returns a Collection`
  - `entrySet() // returns a Set`

The **Map** Interface, cont.

```
public interface Map<K,V> {
    // Basic Operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();
    // Bulk Operations
    void putAll(Map<? extends K, ? extends V> t);
    void clear();
    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K, V>> entrySet();
    // Interface for entrySet elements
    public interface Entry<K,V> {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

`HashMap` and **`TreeMap`** Classes

- Both implement the **`Map`** interface.

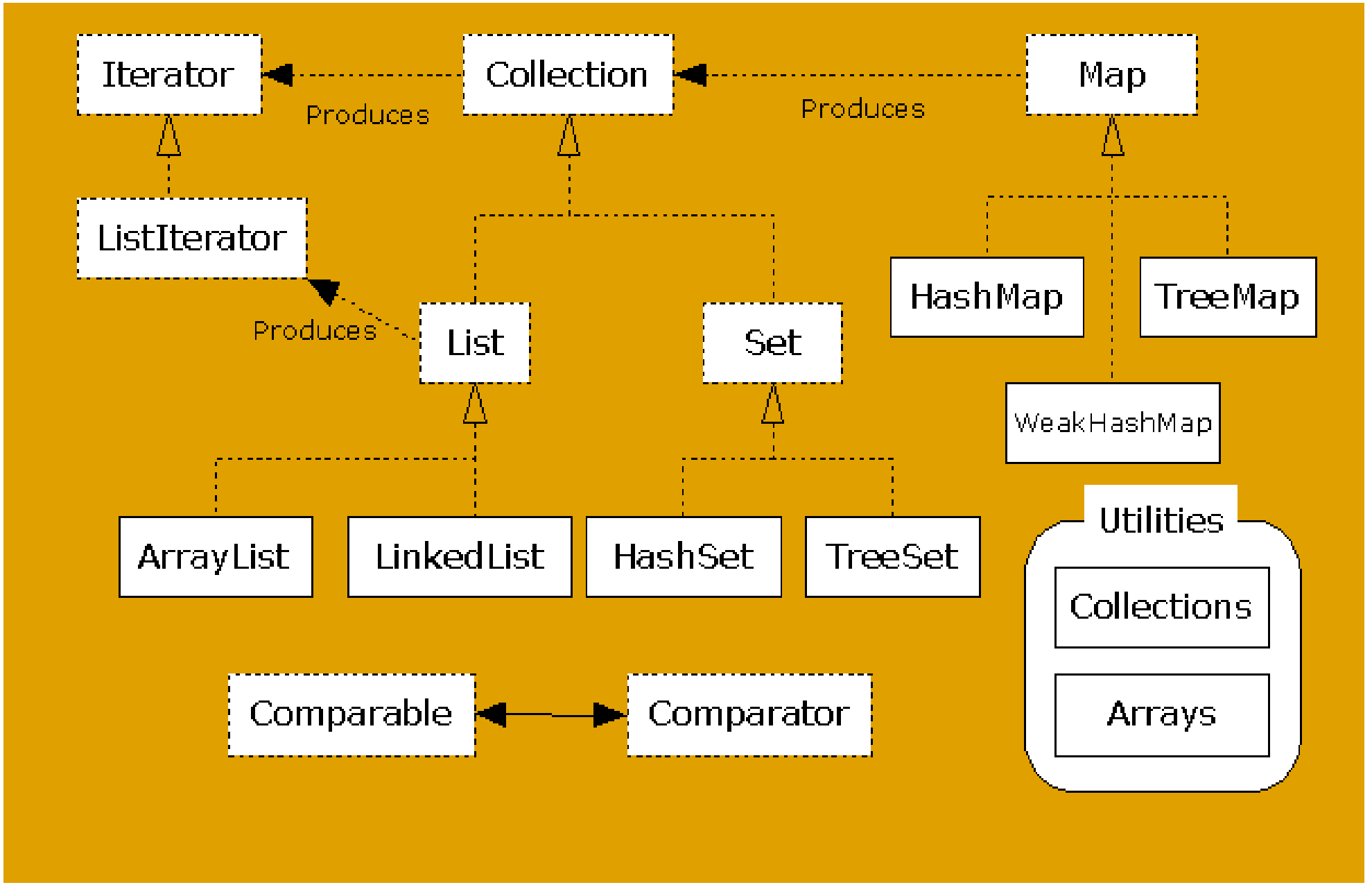
- **`HashMap`**
  - The implementation is based on a hash table
  - No ordering on (key, value) pairs

- **`TreeMap`**
  - The implementation is based on *red-black tree structure*
  - (key, value) pairs are ordered on the key

HashMap, Example

```
import java.util.*;

public class Freq {
    public static void main(String[] args) {
        HashMap<String, Integer> m =
                new HashMap<String, Integer>();

        // Initialize frequency table from command line
        for (String s : args) {
            Integer freq = m.get(s);
            m.put(s, (freq == null ? 1 : freq + 1));
        }

    S   System.out.println(m.size() + " distinct words:");
        System.out.println(m);
    }
}
```

Collection Interfaces and Classes



[Source: bruceeckel.com]

Collection Advantages and Disadvantages

Advantages

- Can hold different types of objects
- Resizable
- Same interface to many different implementations
- Simple
- Well-designed

Disadvantages

- Can hold different types of objects
- Must cast to correct type
- Cannot do compile-time type checking.
- Must wrap primitive types to store these

- Fixed in Java 5.0 with generics!!!

The Collection Annoyance, Java 5.0

```
import java.util.*;
// skeleton classes
class Apple { private int weight; }
class Banana{ private int length;}


public class CollectionAnnoyance{
    public static void main(String[] args){
        Collection<Apple> c = new ArrayList<Apple>();
        for(int i = 0; i < 10; i++){
            c.add(new Apple(i)); // no upcast
        }
        //c.add(new Banana()); // compile-time error
        //c.add(c);            // compile-time error

        Iterator i = c.iterator();
        while (i.hasNext()){
            Apple a = i.next(); // no downcast
        }
    }
}
```

Summary

- Array
  - Holds objects of known type.
  - Fixed size.
- Collections
  - Generalization of the array concept.
  - Set of interfaces defined in Java for storing object.
  - Multiple types of objects.
  - Resizable.
- Major changes to in Java 5.0
  - Introduced generics
  - Makes code short and more readable (Hint: Use it in your project)

- **`java.util.Vector`** is ugly use **`ArrayList`** instead!

Static Methods on Collections

- Collection
  - Search and sort: **`binarySearch()`**, **`sort()`**
  - Reorganization: **`reverse()`**, **`shuffle()`**
  - Wrappings: **`unModifiableCollection`**, **`synchonizedCollection`**

`LinkedList`, Example

```
import java.util.*;
public class MyStack {
    private LinkedList list = new LinkedList();
    public void push(Object o){
        list.addFirst(o);
    }
    public Object top(){
        return list.getFirst();
    }
    public Object pop(){
        return list.removeFirst();
    }

    public static void main(String args[]) {
        Car myCar;
        MyStack s = new MyStack();
        s.push (new Car());
        myCar = (Car)s.pop();
    }
}
```

The `ListIterator` Interface

```
public interface ListIterator extends Iterator {
    boolean hasNext();          // from Iterator
    Object next();              // from Iterator

    boolean hasPrevious();
    Object previous();

    int nextIndex();
    int previousIndex();

    void remove();              // from Iterator Optional
    void set(Object o);         // Optional
    void add(Object o);         // Optional
}
```