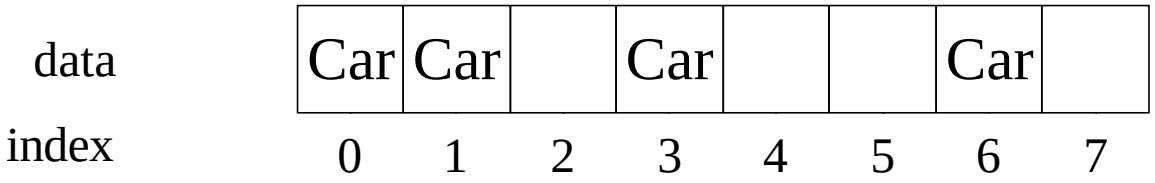
Collections in Java

- Arrays
  - Has special language support
- Iterators
  - **`Iterator`** (i)
- Collections (also called containers)
  - **`Collection`** (i)
  - **`Set`** (i),
    - **`HashSet`** (c), **`TreeSet`** (c)
  - **`List`** (i),
    - **`ArrayList`** (c), **`LinkedList`** (c)
  - **`Map`** (i),
    - **`HashMap`** (c), **`TreeMap`** (c)

Array

- Most efficient way to hold references to objects.



- Advantages
  - An array know the type it holds, i.e., compile-time type checking.
  - An array know its size, i.e., ask for the length.
  - An array can hold primitive types directly.
- Disadvantages
  - An array can only hold one type of objects (including primitives).
  - Arrays are fixed size.

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 = {new Car(), new Car(), new Car()};
```

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

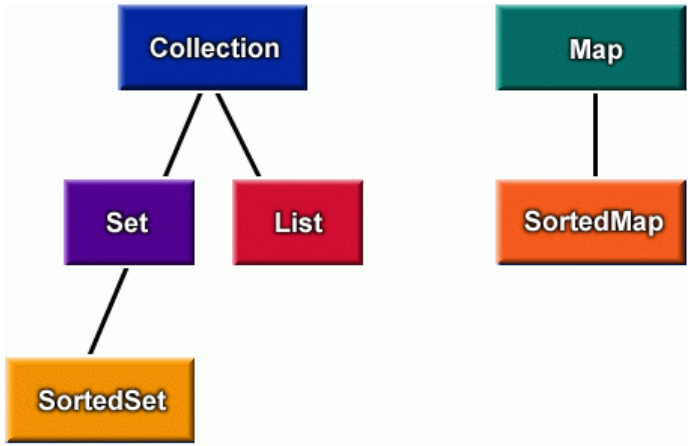
Overview of Collection

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

- Insulate client programs from the implementation.
  - array, linked list, hash table, balanced binary tree
- Like C++'s Standard Template Library (STL)
- Can grow as necessary.
- Contain only **`Objects`** (reference types).
- Heterogeneous.
- Can be made thread safe (concurrent access).
- Can be made not-modifiable.

Collection Interfaces

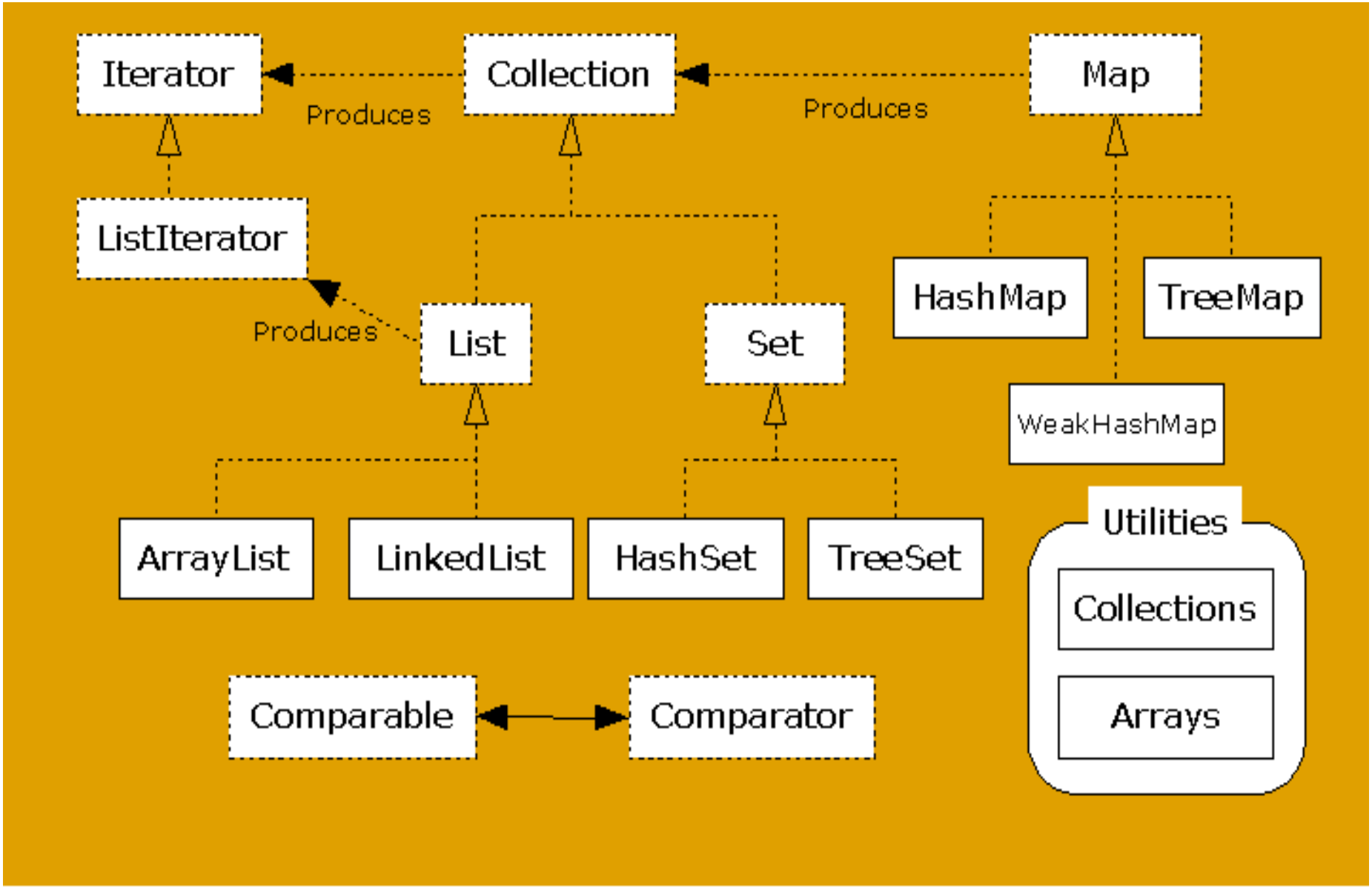
- Collections are primarily defined through a set of interfaces.
  - Supported by a set of classes that implement the interfaces



[Source: java.sun.com]

- Interfaces are used of flexibility reasons
  - Programs that uses an interface is not tightened to a specific implementation of a collection.
  - It is easy to change or replace the underlying collection class with another (more efficient) class that implements the same interface.

Collection Interfaces and Classes



[Source: bruceeckel.com]

The `Iterator` Interface

- *The idea*: Select each element in a collection
  - Hide the underlying collection

- 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 {
   boolean hasNext();
   Object next();          // note "one-way" traffic
   void remove();
}



// an example
public static void main (String[] args){
   ArrayList cars = new ArrayList();
   for (int i = 0; i < 12; i++)
      cars.add (new Car());

   Iterator it = cats.iterator();
   while (it.hasNext())
      System.out.println ((Car)it.next());
}
```

The **`Collection`** Interface

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

    // Bulk Operations
    boolean containsAll(Collection c);
    boolean addAll(Collection c);    // Optional
    boolean removeAll(Collection c); // Optional
    boolean retainAll(Collection c); // Optional
    void clear();                    // Optional

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

The `Set` Interface

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

- 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 interface **`Set`**.

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

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

`HashSet`, Example

```
// [Source: java.sun.com]
import java.util.*;
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 detected: " +
                                   args[i]);
        }
        System.out.println(s.size() +
                           " distinct words detected: " +
                           s);
  }
}
```

The `List` Interface

- The **`List`** interface corresponds to an order group of elements. Duplicates are allowed.

- 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

- **`add(Object)`** adds at the end of the list.
- **`remove(Object)`** removes at 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 extends Collection {
    // Positional Access
    Object get(int index);
    Object set(int index, Object element); // Optional
    void add(int index, Object element);   // Optional
    Object remove(int index);              // Optional
    abstract boolean addAll(int index, Collection c);
                                           // Optional

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

    // Iteration
    ListIterator listIterator();
    ListIterator listIterator(int index);

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

`ArrayList` and `LinkedList` Classes

- The classes **`ArrayList`** and **`LinkedList`** implement the **`List`** interface.

- **`ArrayList`** is an array based implementation where elements can be accessed directly via the **`get`** and **`set`** methods.
  - Default choice for simple sequence.

- **`LinkedList`** is 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`**.

`ArrayList`, Example

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

public class Shuffle {
    public static void main(String args[]) {
        List l = new ArrayList();
        for (int i = 0; i < args.length; i++)
            l.add(args[i]);
        Collections.shuffle(l, new Random());
        System.out.println(l);
    }
}
```

`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();
    Object next();

    boolean hasPrevious();
    Object previous();

    int nextIndex();
    int previousIndex();

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

The **`Map`** Interface

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

- 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 {
    // Basic Operations
    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();
    // Bulk Operations
    void putAll(Map t);
    void clear();
    // Collection Views
    public Set keySet();
    public Collection values();
    public Set entrySet();
    // Interface for entrySet elements
    public interface Entry {
        Object getKey();
        Object getValue();
        Object setValue(Object value);
    }
}
```

`HashMap` and `TreeMap` Classes

- The **`HashMap`** and **`HashTree`** classes 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 {
    private static final Integer ONE = new Integer(1);
    public static void main(String args[]) {
        Map m = new HashMap();

        // Initialize frequency table from command line
        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 detected:");
        System.out.println(m);
    }
}
```

Static Methods on Collections

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

Collection Advantages and Disadvantages

Advantages

- Can hold different types of objects.
- Resizable

Disadvantages

- Must cast to correct type
- Cannot do compile-time type checking.

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.
- Queue, Stack, Deque classes absent
  - Use **`LinkedList`**.