Collections in Java

GLTi Institutional Presentation

HSBC Technology and Services

HSBC
The world's local bank

Objective

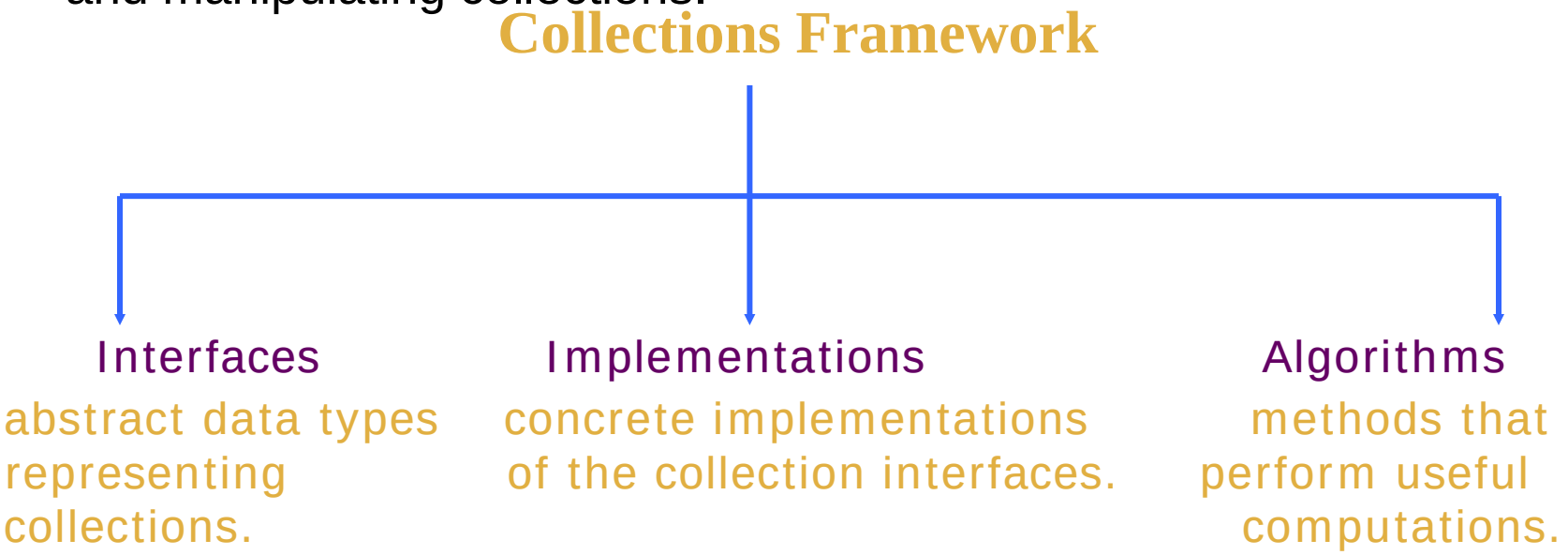
- Introduction to Collections
  - SET
  - LIST
  - MAP

Collections

- A collection - a *container* is an object that groups multiple elements into a single unit.
- Collections are used to store, retrieve and manipulate data, and to transmit data from one method to another.
- Collections typically represent data items that form a natural group, a mail folder, a telephone directory…

Collections Framework?

- A collections framework is a unified architecture for representing and manipulating collections.



Benefits

- It reduces programming effort.
- It increases program speed and quality: The collections framework does this primarily by providing high-performance, high-quality implementations of useful data structures.
- It allows interoperability among unrelated APIs
- It reduces the effort to learn and use new APIs
- It encourages software reuse

Advantages & Disadvantages

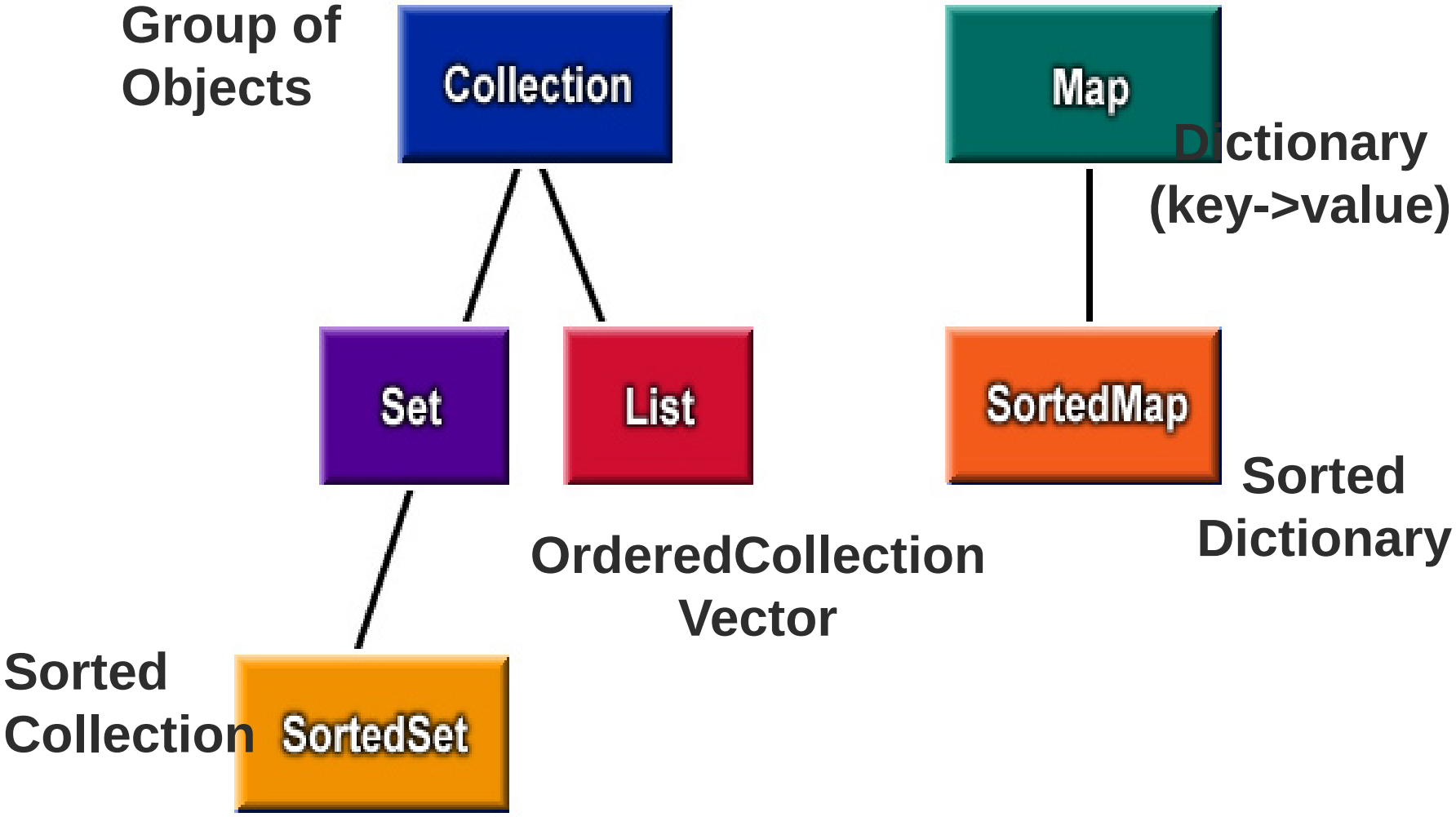
Advantages

- Can hold different types of objects.
- Resizable

Disadvantages

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

Types of Collections



Kinds of Collections

- Collection—a group of objects, called *elements*
  - Set—An unordered collection with no duplicates
    - SortedSet—An ordered collection with no duplicates
  - List—an ordered collection, duplicates are allowed
- Map—a collection that maps *keys* to *values*
  - SortedMap—a collection ordered by the keys
- Note that there are *two* distinct hierarchies

The Interfaces in detail

- The *core collection interfaces* are the interfaces used to manipulate collections, and to pass them from one method to another. It defines basic framework for collections. It provides general functions to add, remove, count the items in/from collection.

- Purpose
  - To allow collections to be manipulated independently of the details of their representation

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[]);
}
```

List

- The List interface corresponds to an order group of elements.
- List can contain duplicate elements.
- User has precise control over where in the List

  each element is inserted.
- In addition to the operations inherited from Collection, the List interface includes operations for:
  - Positional Access
  - Search
  - List Iteration
  - Range-view

List Interface

```
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);
}
```

Sets

- A group of unique items, meaning that the group
  - contains no duplicates
- Some examples
  - The set of uppercase letters ‘A’ through ‘Z’
  - The set of nonnegative integers { 0, 1, 2, … }
  - The empty set {}
- The basic properties of sets
  - Contain only one instance of each item
  - May be finite or infinite
  - Null is allowed- only once

Set Interface

```
public interface Set extends Collection
{  // Basic Operations

     int size();
   boolean isEmpty();
   boolean contains(Object element);
   boolean add(Object element); // Optional
   boolean remove(Object element); // Optional
   Iterator iterator();
```

Set Interface

```
// 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[]);
}
```

Maps

- A map is a special kind of set.
- A map is a set of pairs, each pair representing a one-directional "mapping" from one set to another
  - An object that maps keys to values

- Some examples
  - A map of keys to database records
  - A dictionary (words mapped to meanings)
  - The conversion from base 2 to base 10

Map Interface

```
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);
    }
}
```

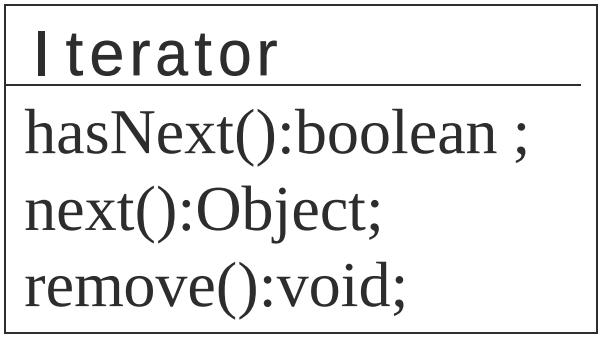
What Is The Real Difference?

- Collections
  - You can add, remove, lookup *isolated* items in the collection
- Maps
  - The collection operations are available but they work with a *key-value* pair instead of an isolated element
  - The typical use of a Map is to provide access to values stored by key

Iterator

- An object that implements the Iterator interface generates a series of elements, one at a time
  - Successive calls to the next()method return successive elements of the series.

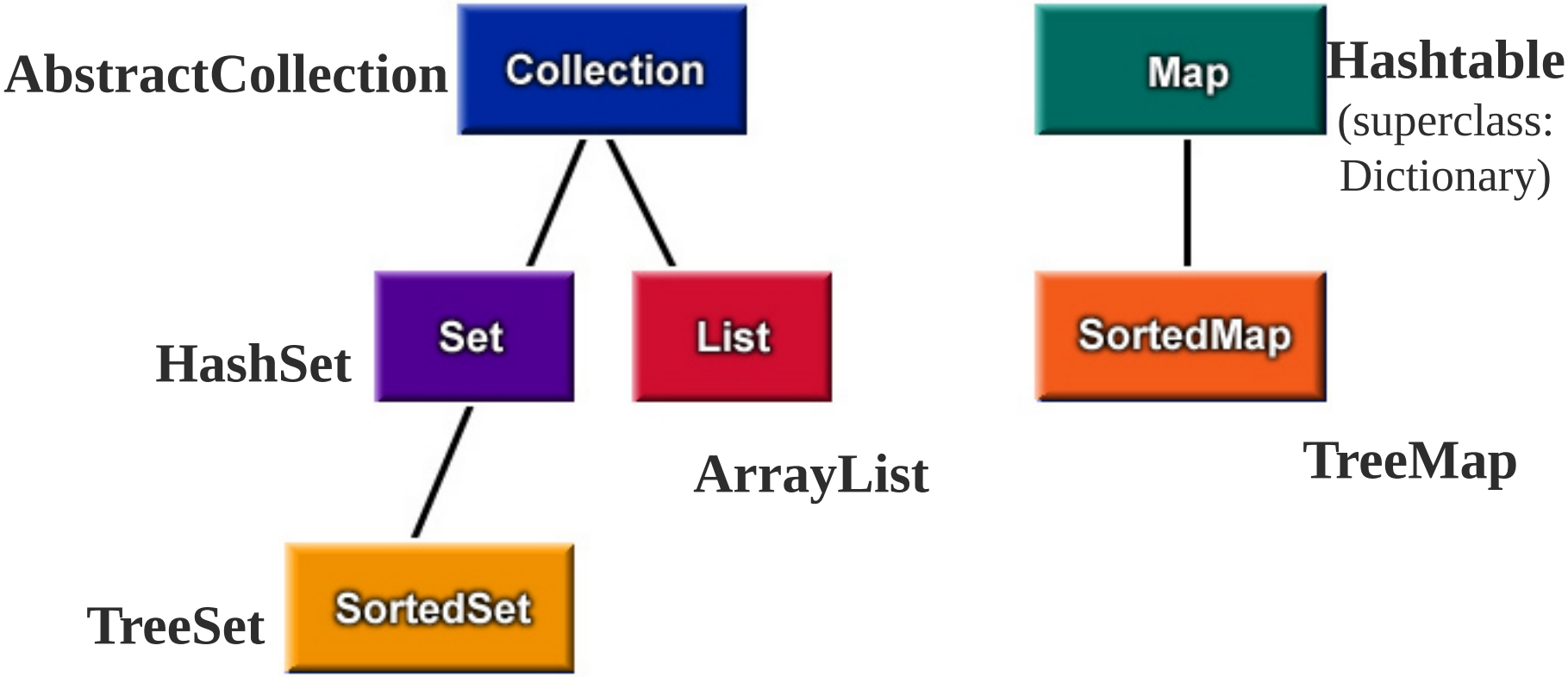
- The remove() method removes from the underlying Collection the last element that was returned by next.



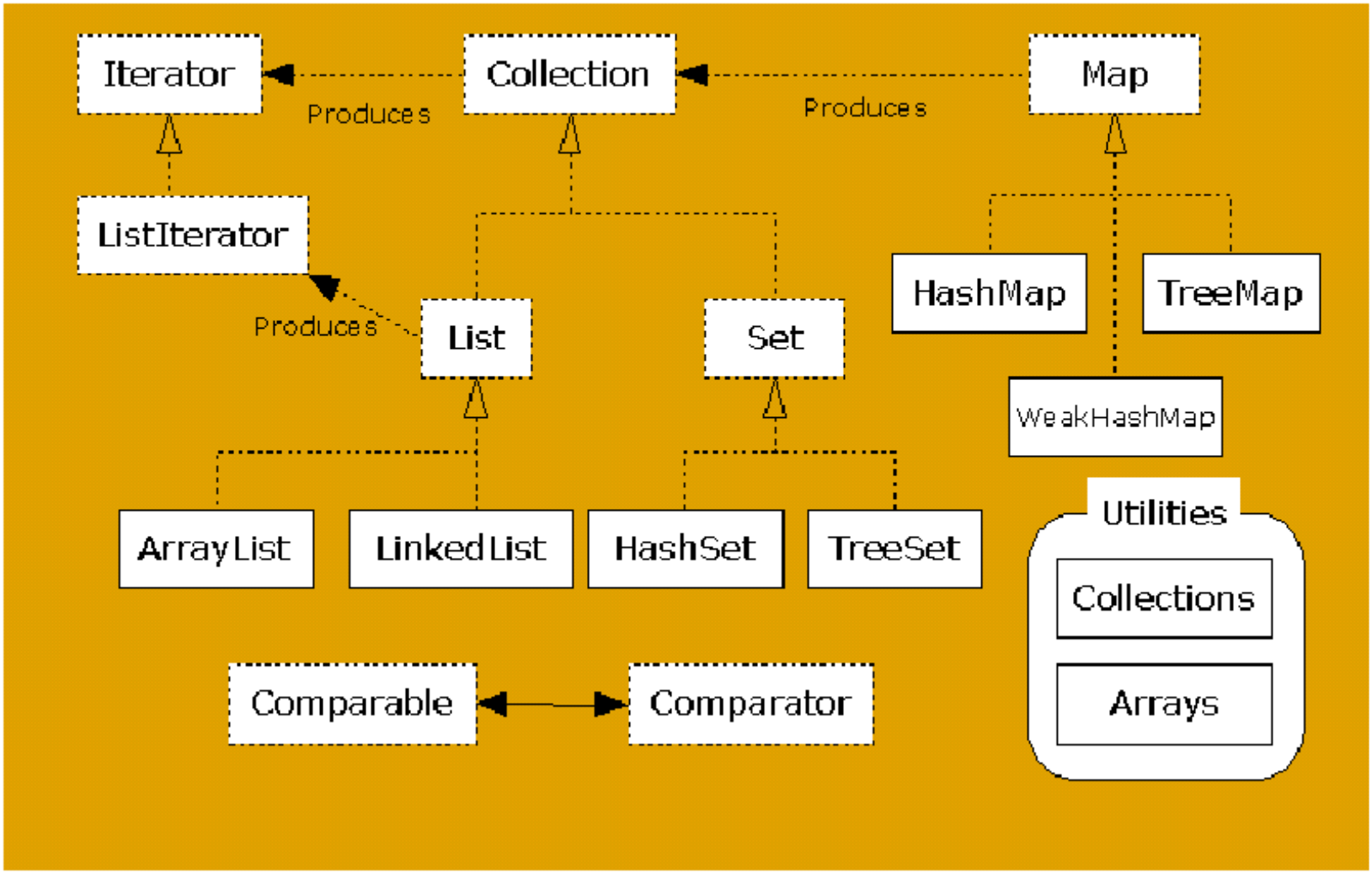
Implementation Classes

Data Structure	Implementation				
	Set	Sorted Set	List	Map	Sorted Map
Hash Table	HashSet			HashMap HashTable	
Resizable Array			ArrayList Vector		ArrayList
Tree		TreeSet			TreeMap
Linked List			LinkedList		

Collection Classes



Collection Interfaces and Classes



[Source: bruceeckel.com]

}

Implementations

ArrayList & LinkedList

Both ArrayList & LinkedList implement the **List** interface.

ArrayList

- an array based implementation
- elements can be accessed directly via **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.

HashSet & TreeSet

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

TreeSet

Implemented using a tree structure.
Guarantees ordering of elements.
add, **remove**, and **contains** methods logarithmic time complexity

HashMap & TreeMap

HashMap and TreeMap implement the interface **Map**

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.

Standard Collection Classes

Standard Map Classes

- AbstarctMap → Map
- HashMap → AbstarctMap
- TreeMap → AbstarctMap
- WeakHashMap → AbstarctMap

Collection Algorithms

- Applied to collections & maps
- Static methods with **Collections** class
- Exceptions:-
  - ClassCastException
  - UnsupportedOperationException

Arrays Class

- Not a part of Collection Package

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**.

The `Map` Interface

- Unlike **`List`** and **`Set`**, the **`Map`** interface is not extending the **`Collection`** interface.
- **`Map`** is heading an interface hierarchy which focuses on maintaining key-value associations.
  - The Data is stored in pairs of objects: key and value
    - Every value object has a key object attached to it.
    - The key determines where will the pair be stored in the **`Map`**.
    - You can only retrieve a value object through its key.
    - No duplicate keys are allowed.

Map's Methods

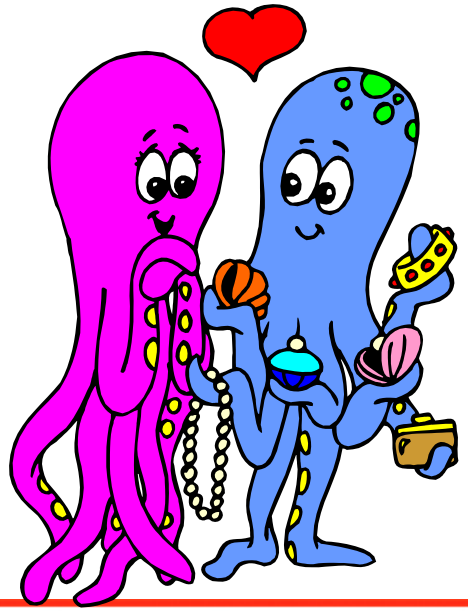
- Methods that deal with adding and removing of key-value pairs:
  - `public Object put(Object key, Object value)`
  - `public Object remove(Object key)`
  - `public void putAll(Map mapping)`
  - `public void clear()`

You can add a Map Object to your Map. But do not try to add a Map to itself....

- Methods that allow you to query the **`Map`**'s content:
  - `public Object get(Object key)`
  - `public boolean containsKey(Object key)`
  - `public boolean containsValue(Object value)`
  - `public int size()`
  - `public boolean isEmpty()`

Map's Methods

- Methods that return objects of type **`Set`** or **`Collection`**:
  - Since the keys must be unique, they can create a Set.
    - **`public Set keySet()`**
    - **`public Set entrySet()`**
  - Value Objects might not be unique, so they can be returned as a **`Collection:`**
    - **`public Collection values()`**



Map Implementations

- The collection framework supplies two implementations to the **Map** interface.
  - **HashMap** and **TreeMap**
- **HashMap** is the best alternative when the desired operations consist on: inserting, deleting and locating.
- **TreeMap** is much slower in performing the operations described here, but it is your only option when you require your data to be sorted.
  - To reduce the amount of time needed, you can first populate a **HashMap** and then convert it to a **TreeMap**.

Iterating a Map

- Both **`HashMap`** and **`TreeMap`** does not supply a direct way to iterate on.
- To do so you can use one of the following methods:
  - **`public Collection values()`**
  - **`public Set keySet()`**
  - **`public Set entrySet()`**

Returns a `Collection` of the map values

Returns a `Set` of the map keys

Returns a `Set` of the mappings contained in this map. Each element is a Map.Entry.

Map Example

- The next example puts into a **`HashMap`** pairs of key: student name and value: **`Grades`** object.
- After filling the map it is converted to a sorted one, a **`TreeMap`**.
- Each of the Map implementations is iterated and printed using the **`Map.Entry`** interface.

Students.java

```
1    import java.util.*;
2    class Students
3    {
4      public static void main(String args[])
5      {
6        Map map = new HashMap(); // A new collection
7        map.put("Ted Lee",new Grades(100,52,87));
8        map.put("Karli Wugofski",new Grades(99,52,77));
9        map.put("Leo Street",new Grades(100,84,87));
10       map.put("Patrick Levi",new Grades(67,59,81));
11       System.out.println("list of students and averages:");
12       print(map);
13       TreeMap sortedmap = new TreeMap(map);
14       System.out.println("same list, this time sorted:");
15       print(sortedmap);
16     }
```

Creating and populating a HashMap

Creating a TreeMap

Printing the Map Elements

```
1    public static void print(Map map)
2      {
3         Set set = map.entrySet();
4         Iterator it = set.iterator();
5         while(it.hasNext())
6         {
7           Map.Entry entry = (Map.Entry)it.next();
8           System.out.println((String)entry.getKey()+ " average is:
9                       " +((Grades)entry.getValue()).getAverage());
10        }
11     }
12   }
```

Grades.java

```
1   class Grades
2   {
3      int m_nYear1, m_nYear2, m_nYear3, m_nTotalYears = 3, m_nTotalSum;

4      public Grades(int year1,int year2,int year3)
5      {
6        m_nYear1 = year1;
7        m_nYear2 = year2;
8        m_nYear3 = year3;
9      }
10     public int getAverage()
11     {
12       int m_nTotalSum =  m_nYear1 + m_nYear2 + m_nYear3;
13       return m_nTotalSum/m_nTotalYears;
14     }
15  }
```

Map Exercise

- You are the chief secretary of a special Wizardry School.
- You receive information in the following format: student name, a grade (e.g. "Sam Marshal,85","Ruth Addison,75",Sam Marshal,99).
- You must place the information inside a map.
  - The key is the name of the student (yes, we know that an ID is a better key but is less applicable for this exercise).
  - The value is a new **`Grade`** object:

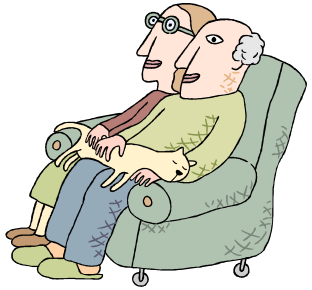
```
class Grade
{
  int m_nGrade;
  public Grade(int grade)
  {
        m_nGrade=grade;
  }
  public int getGrade()
  {
    return m_nGrade;
  }
}
```

Exercise Instructions

- You have to make sure that the map does not already contain the key. If it does, you'll need to remove the pair, create a new **`ArrayList`** object, add the **`Grade`** object to it and put the new key-value pair inside the map.
  - If this is the third time the same student appears on your list, you have already created an **`ArrayList`** for him/her and all you have to do is add the new **`Grade`** to it.
- Print all the students and grades.

Map Summary

- **`Map`** does not extend the **`Collection`** interface.
- It contains pairs of objects: key and value.
  - No duplicate keys are allowed.
- You don't **`add`** an **`Object`** to the **`Map`**, you **`put`** a pair of **`Object`**s inside it.
  - Values can only be removed or pulled out (**`get`**) using their key.
- In order to sort (by value) the content of a **`Map`**, you'll need to use its **`TreeMap`** implementation.



`List` Summary

- The **`List`** interface allows for duplicate content.
  - Therefore, it does not require its added elements to implement **`equals`**() or **`hashCode`**().
- It does not perform any kind of sorting.
- Elements are not randomly stored, as in a **`Set`**. Rather, they maintain their insert point.
  - Which is dynamically adjusted according to other add and remove operations.
- The **`ArrayList`** is a resizable array. All elements are added to the end of the list (and may only be removed of it).
- **`ArrayList`** allows operations on both ends of the list.

`Set` Summary

- **`Set`** is the simplest collection of the entire Collection framework.
- The **`Set`** interface has two concrete classes that implement it::
  - **`HashSet`** is very efficient and is mostly used to create, populate and iterate a collection of randomly-ordered elements.
  - **`TreeSet`** is helpful when you wish to sort the collection's elements.
    - Sorting is heavily leaning of the **`Comparable`** and **`Comparator`** interfaces.
- The **`Set`** interface is extended by the **`SortedSet`** interface which enhances its sorting possibilities.

Summary….

- Main Interfaces
- `Collection`
  - **`boolean containsAll(Collection collection)`**
  - **`boolean add(Object o)`**
  - **`boolean remove(Object o)`**
- Types of Collections
  - List
  - Set

}

Thank You

HSBC

The world's local bank

About HSBC Technology and Services

HSBC Technology and Services (HTS) is a pivotal part of the Group and seamlessly integrates technology platforms and operations with an aim to re-define customer experience and drive down unit cost of production. Its solutions connect people, devices and networks across the globe and combine domain expertise, process skills and technology to deliver unparalleled business value, thereby enabling HSBC to stay ahead of competition by addressing market changes quickly and developing profitable customer relationships.

Presenter's Contact Details:
Name: Deepak Ratnani
Role: Team Leader
Direct: + 91-20 -66423608
Email: DeepakRatnani@hsbc.co.in