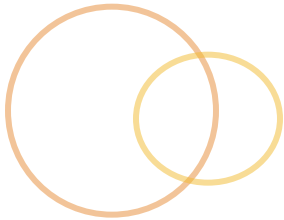
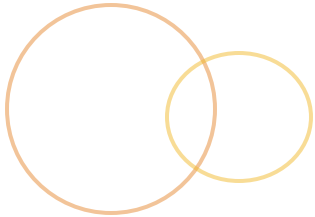
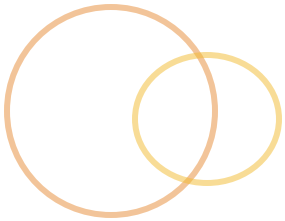
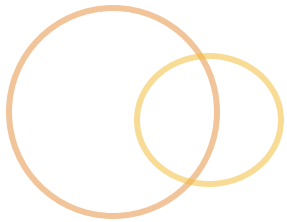
Collections







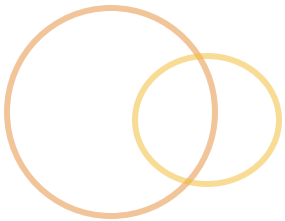
Objectives



At the end of this module you should be able to:

- Describe the collections framework architecture
- Use an `Iterator`
- Use a `Set`
- Use a `List`
- Use a `Map`
- Use collection algorithms
- Use wrappers

Collections

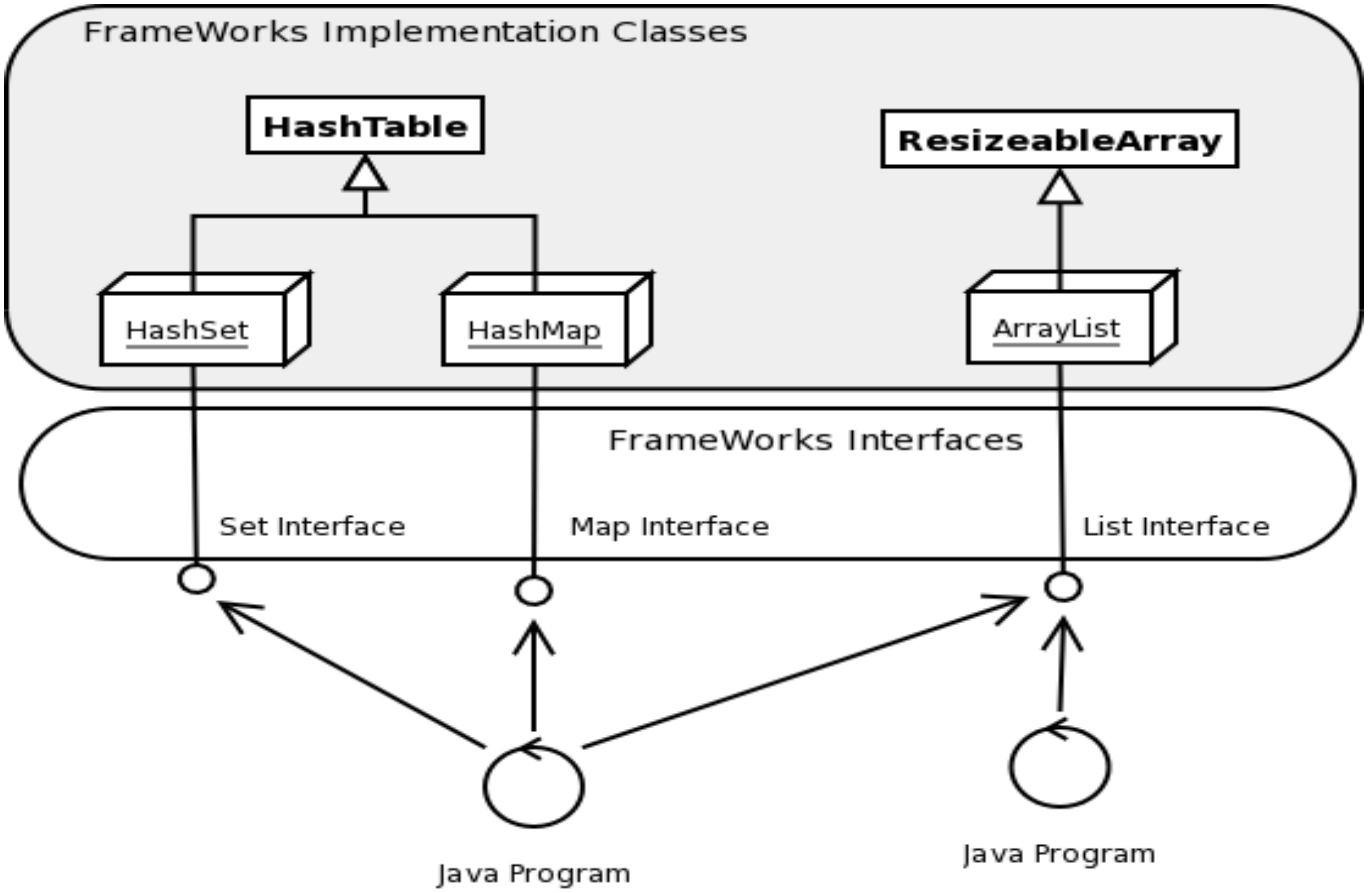


- A collection is a container for other objects
- Arrays are a basic type of collection
- Java provides several collection types, e.g.,:
  - **`List`**
  - **`Set`**
  - **`Map`**

Collections Framework

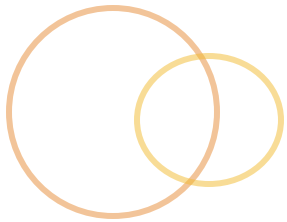
- The collections API has three key elements:
  - Interfaces
    - Expose the functionality of collections
    - Underlying container is manipulated through the interface
    - Client is not coded to the implementation
    - Trivializes changing implementations
  - Implementations
    - The data structure mechanisms themselves
    - Used to add more or specific functionality
  - Algorithms and Wrappers
    - Reusable external functionality
    - Sorting and searching

The Java Collections Framework Architecture



Part of the Collections Framework architecture

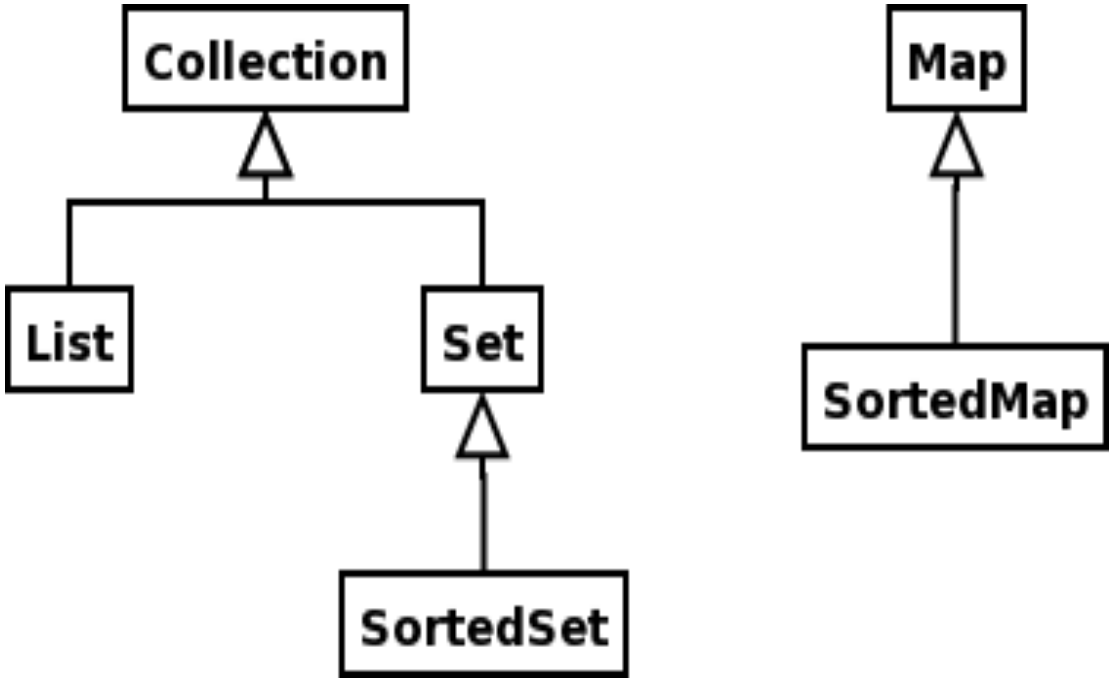
Collection Types



Two main categories of collections

- `java.util.Collection`
  - Root interface in the collection hierarchy
  - May contain duplicates
  - May be ordered
  - Useful only through implementations like
    - `ArrayList`
    - `HashSet`
- `java.util.Map`
  - An object that maps keys to values
  - Cannot contain duplicate keys
  - Each key can map to at most one value
  - Useful only through implementations
    - `TreeMap`
    - `HashMap`

The Collections Interfaces



Frameworks Interface Hierarchy

Categories of Collections

Two main categories of collections:

- Traditional (thread safe)
- Concurrent

Traditional Collections

- When dealing with collections in the traditional way, all methods are synchronized.
- Only one thread of execution is allowed to access any method at any given time.
- Traditional collections are thread safe, meaning that the data being retrieved or inserted into a collection is guaranteed to be up to date.
- Because of this characteristic traditional collections can be slow when different parts of a program are vying for access to a single collection.

Concurent Collections Overview

- In Java 1.5 concurrent collections were introduced collections.
- These new collections are housed in the `java.util.concurrent` package and don't have the same thread-safe guarantee.
- Instead of providing an iterator into the current collection of items, the new collections provide a copy of the current items in the collection as a separate collection.
- This copy can be browsed but is not guaranteed to be the most current copy if the collection has been changed since the copy was created.

Collection Interface API

```
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[]);
 }
```

Using the Collections Framework

Basic steps for using collections framework:

1. Select the interface appropriate for the application.
2. Select the desired data structure implementation.
3. Instantiate the implementation.
4. Manipulate the data structure using the interface.

Creating, Filling, and Printing Collections Example

```
import java.util.*;
// This is a utility class that provides a method for
//  filling a collection -- any collection because it only uses
//  the methods in the collection interface. This shows the
//  use of the Collections type as a general type for passing
// as an argument.
class Fill {
  static Collection init(Collection c, int slots) {
    for (int i = 0; i < slots; i++) {
       c.add("Test Value " + i);
    }
    return c;
  }
}
```

Creating, Filling, and Printing Collections (cont.)

```
public class UseSomeCollections {
  public static void main(String[] args) {
    Collection arrayList = new ArrayList();
    Collection hashSet = new HashSet();
    Collection treeSet = new TreeSet();
    Collection linkList = new LinkedList();
    arrayList = Fill.init(arrayList,5);
    hashSet = Fill.init(hashSet,5);
    treeSet = Fill.init(treeSet,5);
    linkList = Fill.init(linkList,5);
    System.out.println("ArrayList");
    System.out.println(arrayList);
    System.out.println("HashSet");
    System.out.println(hashSet);
    System.out.println("TreeSet");
    System.out.println(treeSet);
    System.out.println("LinkedList");
    System.out.println(linkList);
  }
}
```

Creating, Filling, and Printing Collections Output

```
// Output is
ArrayList
[Test Value 0, Test Value 1, Test Value 2, Test Value 3, Test Value 4]
HashSet
[Test Value 2, Test Value 3, Test Value 1, Test Value 0, Test Value 4]
TreeSet
[Test Value 0, Test Value 1, Test Value 2, Test Value 3, Test Value 4]
LinkedList
[Test Value 0, Test Value 1, Test Value 2, Test Value 3, Test Value 4]
```

Iterator Interface API

- Both `java.util.Collection` and `java.util.Map` provide a mechanism to iterate over the contained values.
- **`Iterator`** is an interface describing how to Iterator over the collection.
- Each implementation class will provide its own **`Iterator`** implementation

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();    // Optional
}
```

Iteration Example

```
import java.util.*;
// Now we have added a generic Iterator method
class Fill {
   static Collection init(Collection c, int slots) {
     for (int i = 0; i < slots; i++) {
       c.add("Test Value " + i);
     }
     return c;
   }
   static void deleteSecond(Collection c) {
     Iterator itr = c.iterator();
     boolean even = false;
     while (itr.hasNext()) {
       itr.next();
       if (even) {
        itr.remove();
       }
       even = !even;
     }
   }
}
```

Iteration Example (cont.)

```
public static void main(String[] args) {
  Collection arrayList = new ArrayList();
  Collection hashSet = new HashSet();
  Collection treeSet = new TreeSet();
  Collection linkList = new LinkedList();
  arrayList = Fill.init(arrayList, 5);
  hashSet = Fill.init(hashSet, 5);
  treeSet = Fill.init(treeSet, 5);
  linkList = Fill.init(linkList, 5);
  System.out.println("ArrayList");
  Fill.deleteSecond(arrayList);
  System.out.println(arrayList);
  System.out.println("HashSet");
  Fill.deleteSecond(hashSet);
  System.out.println(hashSet);
  System.out.println("TreeSet");
  Fill.deleteSecond(treeSet);
  System.out.println(treeSet);
  System.out.println("LinkedList");
  Fill.deleteSecond(linkList);
  System.out.println(linkList);
}
```

Iteration Example Output

```
// Output is
ArrayList
[Test Value 0, Test Value 2, Test Value 4]
HashSet
[Test Value 2, Test Value 1, Test Value 4]
TreeSet
[Test Value 0, Test Value 2, Test Value 4]
LinkedList
[Test Value 0, Test Value 2, Test Value 4]
```

`Set` Interface

- A set is a collection that contains no duplicates.
- Sets are sub-interfaces of `java.util.Collection`

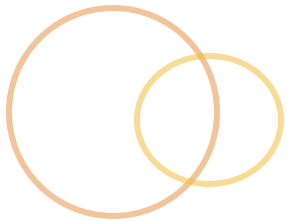
`Set` Interface Example

```
import java.util.*;
class Test{}  // something to put in the Set
public class TestASet {
  public static void main(String [] args) {
    Set s = new HashSet();  // create the set
    Test t = new Test();
    s.add(t);
    s.add(t);     // duplicate entry
    s.add("One");
    s.add("Two");
    s.add("One");
    s.add("One");
    s.add("Three");
    s.add("Four");
    s.add("Four");
    s.add("Four");
    s.add(new Test());  /// not a duplicate
    System.out.println(s);
  }
}
// Output is:
[Test@107077e, Test@11a698a, Four, Three, Two, One]
```

`List` Interface

- A list is an ordered collection or sequence.
- Lists are sub-interfaces of `java.util.Collection`
- May contain duplicate elements
- Implementations typically allow **`null`**
- Supports positional access for insertion and retrieval (based on index)
- Has a special type of `iterator`, **`ListIterator`**
  - Allows insertion and replacement while iterating over the collection
  - Supports **`Iterator`** interface operations

`List` Interface API



```
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);
}
```

`List` Iterator API

```
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
}
```

List Example

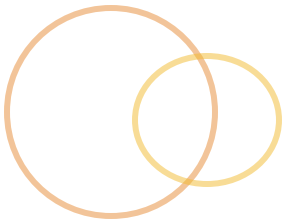
```
import java.util.*;
public class TestAList {
  public static void main(String[] args) {
    List L = new LinkedList();
    for (int i = 0; i < 10; i++) {
      L.add("" + i);
    }
    System.out.println("List created");
    System.out.println(L);
    L.add(4, "10");
    System.out.println(L);
    L.set(5, "11");
    System.out.println(L);
    ListIterator itl = L.listIterator(4);
    System.out.println("L[4]=" + L.get(4));
    itl.previous();
    itl.remove();
    System.out.println(L);}
  }
}
```

```
// output
List created
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 10, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 10, 11, 5, 6, 7, 8, 9]
L[4]=10
[0, 1, 2, 10, 11, 5, 6, 7, 8, 9]
```

`Map` Interface

- Maps keys to values
  - Like a micro-database with two columns: key and data
- Contains no duplicate keys, values may be duplicates
- No direct iterator functionality
- Provides three views of data that allow us to obtain iterators:
  - Keys
  - Values
  - Entry set (key-value mappings)

Map Interface API

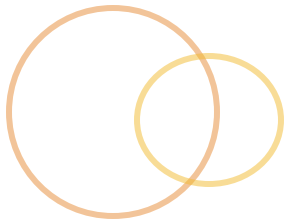


```
public interface Map {
    Object put(Object key, Object value);
    Object get(Object key);
    Object remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    void putAll(Map t);
    void clear();

    public Set keySet();
    public Collection values();
    public Set entrySet();

    public interface Entry {
        Object getKey();
        Object getValue();
        Object setValue(Object value);
    }
}
```

Map Example

```
public class TestAMap {
  public static void main(String[] args) {
    Map custs = new HashMap();
    custs.put("982098", new Customer("Bill White"));
    custs.put("116201", new Customer("Bob Green"));
    custs.put("983611", new Customer("Saj Black"));
    custs.put("661109", new Customer("Sharon Brown"));
    System.out.println(custs);

    custs.remove("116201");
    custs.put("761102", new Customer("Simone Blanc"));
    System.out.println(custs.get("661109"));
    System.out.println(custs);
  . . .
```

Map Example (cont.)



```
    // Now walk through the entries
    Set entries = custs.entrySet();
    Iterator iter = entries.iterator();
    while (iter.hasNext()) {
      Map.Entry entry = (Map.Entry) iter.next();
      Object key = entry.getKey();
      Object value = entry.getValue();
      System.out.println("key=" + key + ", value=" + value);
    }
  } //end main
} //end class
```

java.util.Collections

- A utility class that provides
  - Algorithms
  - Wrappers
- Static methods for common algorithms for things like
  - Binary search
  - Reversing
  - Shuffling
  - Sorting
- Wrappers for creating
  - Singletons
  - Synchronized collections
  - Unmodifiable collections
- See also `java.util.Arrays` class

Collections Example

```
public class TestCollectionsUtils {
    public static void main(String[] args) {
        List numbers = new ArrayList(12);
        for (int i = 1; i <= 12; i++) {
            numbers.add(new Integer(i));
        }
        System.out.println("Starting List\n" + numbers);

        Collections.shuffle(numbers); // Randomize
        System.out.println("Shuffled List\n" + numbers);

        Collections.sort(numbers); // Sort
        System.out.println("Sorted List\n" + numbers);

        numbers = Collections.unmodifiableList(numbers);
        Collections.shuffle(numbers); // woops!
    }
}
```

Collections Example (cont.)

```
Starting List
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
Shuffled List
[7, 10, 4, 1, 9, 11, 12, 8, 5, 2, 3, 6]
Sorted List
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
Exception in thread "main" java.lang.UnsupportedOperationException
   at java.util.Collections$UnmodifiableList.set(Collections.java:1156)
[...]
   at tests.TestCollectionsUtils.main(TestCollectionsUtils.java:24)
```

Summary

In this module, we covered

- The collections framework architecture
- Use of an `Iterator`
- Use of a `Set`
- Use of a `List`
- Use of a `Map`
- Using algorithms
- Using wrappers