CSA 1019
Imperative and OO Programming

Collections

Mr. Charlie Abela
Dept. of Artificial Intelligence
Objectives

- Getting familiar with
  - Collections interfaces
  - Abstract and Concrete Classes
  - Lists
  - Sets
  - Maps
Collections are a vital part of any programming language.

ArrayLists/Vectors and Arrays, allow many objects to be collected together and stored as one object.

Arrays are not considered as "collections" in Java.

These collections are located in the java.util.Collection package, along with some other useful tools.
Collections Interfaces

- There are many types of collections and these are organized into a hierarchy of Interfaces, with the roots being either Collection or Map.
- The objects stored in a collection are called elements.
- Primitive data types cannot be stored directly in a collection (wrapper classes need to be used though with Java 1.5 the wrapping is done automatically).
- The collections themselves allow heterogeneous objects to be stored.
Collections Interfaces (cont)

- The Collections interface definitions represent a general type of collection. The collections all differ with respect to:
  - accessing & modifying restrictions
    - some operations (such as `addFirst()` are allowed while others are not)
  - accessing & modifying functionality
    - `add()` may work differently depending on the collection
    - some sort the added elements while others do not
  - storage and efficiency
    - storing things in sorted order makes it easy to find them later.
Some collections, such as **Lists** allow duplicate elements, while **Sets** do not.

**Lists** are ordered, while **Sets** and **Maps** are not.

It is imperative that these characteristics are understood so that the right choice of collection can be made.

Consider issues such as ordering and duplication in the applications that need to be implemented:

- banks which keep collections of bank accounts
- companies which keep collections of employees
- airline reservation systems which keep collections of flights, customers, airlines, seating arrangements
Collection Interface

- Provides methods to modify a collection
  - `add(Object obj)` - adds the given object as an element to the collection (its location is not specified)
  - `addAll(Collection c)` - same as above but adds ALL elements specified in the given collection parameter.
  - `remove(Object obj)` - removes the given object from the collection
  - `removeAll(Collection c)` - same as above but removes ALL elements specified in the given collection parameter.
  - `retainAll(Collection c)` - same as above but removes all elements EXCEPT THOSE specified in the given collection parameter.
  - `clear()` - empties out the collection by removing all elements.
Collection Interface

- Provides methods to query a collection
  - `size()` - returns the number of elements in the collection
  - `isEmpty()` - returns whether or not there are any elements in the collection
  - `contains(Object obj)` - returns whether or not the given object is in the collection (uses `.equals()` method for comparison)
  - `containsAll(Collection c)` - same as above but looks for ALL elements specified in the given collection parameter.
Abstract and Concrete classes

- The abstract classes provide partial implementation for convenience.
- The concrete classes implement the interfaces with concrete data structures.
- All classes inherit from `AbstractCollection`, which contains common behaviour for all collections.
- All classes somehow implement one of the `Collection` interfaces.
- The classes are split as either having a `List` behaviour or a `Set` behaviour, but not both.
List Classes

- elements are ordered
- may contain duplicates
- have indexable elements (using zero-based indexing).
- includes:
  - LinkedList: methods for double-ended access
  - Stack: accessible from one end only.
**List Interface**

- extends **Collection** to define an ordered collection with duplication
- adds position-oriented operations as well as a new list iterator that enables users to traverse the list bi-directionally
  - `add(index, element)` : insert in a specified location
  - `addAll(index, collection)` : insert collection at a specified location
  - `remove(index)` : remove element at a specified index
  - `set(index, element)` : set the existing element at that location
  - `subList(fromIndex, toIndex)` : return a sublist from this collection
  - `listIterator()` and `listIterator(startIndex)` returns a ListIterator that allows for easy traversal of the List. The latter method returns a ListIterator positioned at the specified index
**LinkedList Class**

- A very basic and general data structure. It is actually used as a low-level data structure for implementing more complex data types such as stacks, queues, trees.
- Allows direct operations to the front and the back of the list.
  - Operations to the front and back of the list are typically quick whereas operations to access or modify an inner element of the list are typically very slow, requiring a linear search.
elements of the list are still kept in order and so operations such as adding and removing from any position in the list are possible.

These operations are slower though. It is however more efficient to use these methods:

- `addFirst(Object obj)` - add the given object to the front of the list
- `addLast(Object obj)` - add the given object to the end of the list
- `removeFirst()` - remove and return the first object from the list
- `removeLast()` - remove and return the last object from the list
- `getFirst()` - return the first object in the list
- `getLast()` - return the last object in the list
Example

```java
LinkedList linked = new LinkedList();
//add to specified locations in the List
linked.add("John");
linked.add("Peter");
linked.add("Jane");

for(String name: linked){
    System.out.println("Users name: "+ name);
}

- should make use of the following declaration
  LinkedList<String> linked =
    new LinkedList<String>();

- this declares that the LinkedList will only contain String objects.
```
Example II

- LinkedLists can also be constructed from another collection:

```java
List<Integer> iList = new ArrayList<Integer>();
iList.add(10);
iList.add(21);
iList.add(21);
iList.add(0,20);

List<Object> linked = new LinkedList<Object>(iList);
//add to specified locations in the List
linked.add(0, "John");
linked.add(1, "Peter");
linked.add(2, "Jane");

for(Object item:linked)
    System.out.print(item.toString()+ " ");
```

- Result

```
John Peter Jane 20 10 21 21
```
Implementation issue

- the code should always refer to the Collection by its interface type (*List*) rather than by its implementation type (*LinkedList*).
- This is a *strongly* recommended programming practice because it provides the flexibility to change implementations merely by changing the constructor.
- If variables or parameters used are declared as being of the Collection's *implementation type* rather than its *interface type*, *ALL* such variables and parameters must be changed in order to change its implementation type.
ListIterator class

- The **ListIterator** interface extends the **Iterator** interface

- The **Iterator** provides the methods
  - `hasNext()` : check whether the iterator has more elements to traverse
  - `next()` : to return the next element
  - `remove()` : removes the last element returned by the iterator

- **ListIterator** enriches this definition by adding
  - `hasPrevious()` : checks whether the iterator has more elements when traversed in the backward direction
  - `previous()` : returns the previous element in the iterator
ListIterator iter = linked.listIterator();
//display LinkedList forward
while(iter.hasNext()){
    System.out.print(iter.next() + " ");
}

//display LinkedList backward
for (iter = linked.listIterator(linked.size());
    iter.hasPrevious(); ){
    System.out.print(iter.previous() + " ");
}

- Result
---> John, Peter, Jane, 20, 10, 21, 21,
--> 21: 21: 10: 20: Jane: Peter: John:
Stack Class

- A stack implies a last-in-first-out protocol when accessing the items.
- Items on the stack, are pushed at the top and when removing them, the topmost item is popped.
- Stacks are useful for:
  - Recording the state of a computation as it unfolds
  - Evaluating expressions that involve precedence and nesting
  - Reversing a collection of items

```java
Stack<aType> myStack = new Stack<aType>();
```
Example

### Brackets matching

```java
Stack<Character>aStack = new Stack<Character>();
/*Go through the characters and look for brackets */
for (int i=0; i<input.length(); i++) {
  char aCharacter = input.charAt(i);
  switch(aCharacter) {
    case '(': case '[': case '{': aStack.push(aCharacter); break;
    case ')':
      if(aStack.empty() || (aStack.pop() != '('))
        return(false); break;
    case ']':
      if(aStack.empty() || (aStack.pop() != '['))
        return(false); break;
    case '}':
      if(aStack.empty() || (aStack.pop() != '{'))
        return(false); break;
  }
}
//check for open brackets in the stack
return (aStack.empty());
```

### What if we wanted to program to the List interface?

Charlie Abela

CSA1019: Collections
Set Classes

- The **Set** classes are much like **Lists**, except that they do not allow duplicates.
- there cannot be two elements \(e_1\) and \(e_2\) such that \(e_1.equals(e_2)\)
- any modifications to the elements that affect the equality results in unspecified behaviour
Set Classes (cont)

- **HashSet**
  - elements are *not kept in order*
  - *fast adding/removing* operations
  - *searching is slow* for a particular element
  - elements MUST implement `.equals()` and `.hashCode()`

- **TreeSet**
  - elements are kept in *sorted order*, but are not indexable
  - order is user-defined (default is ascending)
  - *adding is slow* since it takes longer to find the proper location
  - *searching is fast* now since items are in order
Hashing

- A scheme for converting sparse (i.e. quite spread apart) keys into unique array subscripts.
- Hashing essentially generates for each item, a number (called the *hashcode*) which is used as the *key* (or index).
- The generated number has no significant meaning other than to allow efficient storage and retrieval of data.
HashSet class

- **HashSets** (as well as **Hashtables** and **HashMaps**) store their objects according to a **hashfunction** which:
  - returns an integer representation of the object (i.e. its hashcode).
  - the integer reflects the object’s characteristics.
  - equal objects return the same hashcode.

```java
Set<String> set = new HashSet<String>();
set.add("Valletta");
set.add("London");
set.add("Paris");
set.add("Rome");
set.add("Valletta");

Iterator iter = set.iterator();
while(iter.hasNext()){
    System.out.print(iter.next()+ " ");
}
```

**Result**

Valletta Paris Rome London
**TreeSet class**

- **TreeSet** class extends the **AbstractSet** but differently from the **HashSet**, it implements the **SortedSet** interface.

- This guarantees that the elements stored in a **TreeSet** are sorted.

- Additionally to the **HashSet** it provides:
  - `first()` : return the first element in the set
  - `last()` : return the last element in the set
  - `headSet(toElement)` : return parts of the set which includes elements that are less than toElement
  - `tailSet(fromElement)` : returns parts of the set which includes elements that are greater then the fromElement.

```java
Set<String> set = new TreeSet<String>();
```

- Result is now sorted

  **London Paris Rome Valletta**
Comparable

- Anything that implements the Comparable interface provides a natural ordering for a class, which allows objects of that class to be sorted automatically.
  - String, Date and Wrapper classes are amongst these
- A List of String objects can easily be sorted by using
  \[ \text{Collections.sort(myList)}; \]
- Trying to sort a list whose elements do not implement Comparable, will throw a `ClassCastException`
List<String> linked = new LinkedList<String>();

//add to specified locations in the List
linked.add("John");
linked.add("Peter");
linked.add("Jane");

Collections.sort(linked);

for(String name : linked){
    System.out.println(name + " ");
}

- **Result**
- Jane John Peter
This class does not represent a type of collection, rather it contains a bunch of useful tools that operate on collections of different types.

Apart from `.sort()` this class contains:

- `max(Collection coll)` : returns the maximum element in the given collection
- `min(Collection coll)` : returns the minimum element in the given collection
- `reverse(List list)` : reverses the order of the elements in the specified list.
- `shuffle(List list)` : randomly permutes the specified list using a default source of randomness
- `binarySearch(List list, Object key)` : searches the specified list for the specified object using the binary search algorithm.
Example

String[] names = 
{"Al", "Zack", "Sally", "Al", "Mel", "Zack"};
ArrayList<String>coll = new ArrayList<String>();

for(int i=0; i<names.length; i++)
    coll.add(names[i]);

System.out.println("Max = " +
    Collections.max(coll) + ", Min = " +
    Collections.min(coll));

System.out.println("Original:" + coll);
Collections.reverse(coll);
System.out.println("Reversed:" + coll);
Collections.shuffle(coll);
System.out.println("Shuffled:" + coll);
Collections.sort(coll);
System.out.println("Sorted:" + coll);

- Result

Max = Zack, Min = Al
Original:[Al, Zack, Sally, Al, Mel, Zack]
Reversed:[Zack, Mel, Al, Sally, Zack, Al]
Shuffled:[Al, Mel, Zack, Zack, Sally, Al]
Sorted:[Al, Al, Mel, Sally, Zack, Zack]
Comparator

- Whenever elements that need to be compared do not implement the `Comparable` interface, or else these elements are of different types, it is still possible to compare them using the `Comparator` interface.

- Classes that implement this interface need to implement the method

  
  ```java
  compare(Object obj1, Object obj2):
  ```

  returns

  - a negative value if `obj1` is less than `obj2`,
  - a positive value if `obj1` is greater than `obj2`
  - a 0 if they are equal
Example: Employee

```java
public class Employee {
    private Name name;
    private MyDate hireDate;

    public Employee(Name name, MyDate date) {
        this.name = name;
        this.hireDate = date;
    }

    public Name name() {
        return name;
    }

    public MyDate hireDate() {
        return hireDate;
    }

    public String toString() {
        return "Name: " + name.firstName() + " " + name.lastName() + "\nHireDate: " + hireDate.toString() + "\n";
    }
}
```
Example: MyDate

```java
public class MyDate implements Comparable{
    private Integer year;
    private Integer month;
    private Integer day;

    public MyDate(int year, int month, int day) {
        this.year = year;
        this.month = month;
        this.day = day;
    }

    public String toString() {
        return this.day + "/" + this.month + "/" + this.year;
    }

    public int compareTo(Object d) {
        int cmp =
            year.compareTo(((MyDate)d).year);
        return (cmp != 0 ? cmp :
            month.compareTo(((MyDate)d).month));
    }
}
```
public class EmpSort implements Comparator{
    // Employee database
    static final Employee[] employees = {
        new Employee(new Name("Jim","More"),new MyDate(2007,12,23)),
        new Employee(new Name("Jack","Less"),new MyDate(2000,10,20)),
        new Employee(new Name("Jane", "Cool"),new MyDate(2005,6,24))
    };

    public int compare(Object e1, Object e2){
        return ((Employee)e2).hireDate().compareTo(((Employee)e1).hireDate());
    }

    public static void main(String[] args) {
        EmpSort es = new EmpSort();
        List<Employee>e = Arrays.asList(employees);
        Collections.sort(e, es);
        for(Employee emp: e)
            System.out.println(emp.toString());
    }
}
Example: EmpSort

- Using an anonymous Comparator

```java
static final Comparator<Employee> SENIORITY = new Comparator<Employee>() {
    public int compare(Employee e1, Employee e2) {
        return e2.hireDate().compareTo(e1.hireDate());
    }
};

public static void main(String[] args) {
    List<Employee> e = Arrays.asList(employees);
    Collections.sort(e, SENIORITY_ORDER);
    for (Employee emp : e) {
        System.out.println(emp.toString());
    }
}
```
Maps

- The **Map** interface stores objects in a particular way such that the objects can be easily located later on.

- A **Map** really means a "Mapping" of one object to another. Maps are used when it is necessary to access the elements quickly by particular **keys**. Examples are:
  
  - storing many items by category (e.g. a video store is organized by sections (comedy/action/drama)).
  
  - phone books where a value (e.g. number) is associated with a particular key (e.g. name).

  - dictionaries where a value (e.g. definition) is associated with a particular key (e.g., word).
Maps (cont)

- All **Maps** store a group of object pairs called **entries**. Each map **entry** has:
  - **key** - identifies values uniquely (maps cannot have duplicate keys)
  - **value** - accessed by their keys (each key maps to at most one value)
- So, the **key** MUST be used to obtain a particular value from the **Map**.
Map Interface

- The **Map** interface defines many methods.
- Querying methods:
  - `size()` - returns the number of elements in the map.
  - `isEmpty()` - returns whether or not there are any elements in the map.
  - `containsKey(Object key)` - returns whether or not the given object is a key of the map (uses `.equals()` method for comparison).
  - `containsValue(Object val)` - returns whether or not the given object is a value in the map (uses `.equals()` method for comparison).
  - `getKey(Object key)` - returns the `Object` in the map that is associated with the given key.
  - `values(Object key)` - returns a `Collection` containing all the values in the map.
  - `keySet(Object key)` - returns a `Set` of all the keys in the map.
  - `entrySet(Object key)` - returns a `Set` of all the key/value pairs in the map (i.e., `Map.Entry` objects represent key.value pairs).
Map Interface (cont)

Modifying methods (changes the collection in some way):

- **put(Object key, Object val)** - adds a new entry in the map with the given key and value. The key MUST be unique, otherwise this method replaces the value that was previously there by the given value.
- **putAll(Map m)** - adds all entries that are in the given map to the receiver map.
- **remove(Object key)** - removes the given key/value entry from the map based on the key only.
- **clear()** - empties out the map by removing all elements.
HashMap class

- **HashMaps** are not fixed-size, so they can grow as needed (just like with ArrayLists).

- Items are added to the **HashMap** by specifying the key AND the value.
  - The key is used as a unique identifier that distinguishes its associated value from any other value.
  - Both the keys and values can be arbitrary objects (but no null key for Hashtables).

- To create a general **HashMap** that can store arbitrary keys and values we can use this constructor:

  ```java
  HashMap table = new HashMap();
  ```
Example

```java
Map<String,String> phoneBook = new HashMap<>();

phoneBook.put("Arthur","555-8813");
phoneBook.put("Mark", "555-2238");
//returns null
phoneBook.remove("555-8813");
//returns 555-8813
phoneBook.remove("Arthur");
phoneBook.isEmpty(); //returns false
phoneBook.put("Norm", "555-3789");
//returns true
phoneBook.containsKey("Norm");
phoneBook.put("Mark", "555-2200");
phoneBook.values();
//returns [555-3789, 555-2200]
phoneBook.keySet();
//returns [Mark, Norm]
phoneBook.entrySet();
//returns [Mark=555-2200, Norm=555-3789]
```
The basic difference between a HashMap and a TreeMap is that the latter sorts its keys.

TreeMap implements the SortedMap interface and thus its keys are automatically sorted using the Comparable or Comparator interfaces.

```java
Map<String, Integer> hmap = new HashMap<String, Integer>();
hmap.add("Smith", 30);
hmap.add("Borg", 20);
hmap.add("Grech", 24);

Map<String, Integer> tMap = new TreeMap<String, Integer>(hmap);
System.out.println(tMap);
```