Unit 10 - Data Structures

Objectives
1. Describe the usage of data structures.
2. Develop simple data structures.
3. Apply Java collections.
4. Apply wrapper classes.
5. Apply string manipulation with String and StringBuffer classes.

Data structures
- A data structure is a collection of data that can be handled altogether using a single name, e.g., bitwise operation on an integer, an object, an array.

Common data structures
1. List (pp4, fig-10.1)
   - Maintain data items with a predefined order.
     - The order may be alphabetical order, or simply the order of the items entering the list (e.g., a queue).
   - Usually an index is assigned to each item in the list for accessing purpose.
2. Stack
   - Maintain data items in a special order - LIFO (like a pile of files in that the last file you place on the pile is the first one you can get).
3. Tree (pp5, fig-10.2)
   - Maintain structured data that are hierarchical.
   - Each tree contains storage called nodes (represented as rectangles in diagram), and the tree starts with a root node (hence it is actually an inverted tree).
   - The nodes are interconnected with pointers (arrows) from parent node to child node.
   - If a node has at least one child node, it is called an internal node. Otherwise it is a leaf node (i.e., no child).
   - If each node of a tree can have at most 2 nodes, it is called a binary tree.
   - A binary tree is a good data structure for storing and searching a collection of orderable data items (e.g., numbers, names, etc.).
     - e.g. How to store the numbers 45, 60, 57, 23, 10, 70, 31 in a tree structure that can facilitate searching? (pp6, fig-10.3)
4. Set
   - It is a collection of unique elements.
   - A set is like a list but no elements can be duplicated.
5. Map
   - The objects maintained by a list object can be accessed by using an index (i.e., a value of primitive type int in Java).
   - Actually objects in a collection (value objects) can be specified or accessed by using another object (key objects). The relationship between the key and value is a mapping and this structure is called a map.
     - e.g., a Student object can be accessed by using the student name (which is a String object in Java) in a collection.
   - A map is like a list but its index can be any object instead of an integer.

Collections
- Java provides implementation of these frequently used data structures in the java.util package.
- These data structures have the common purpose to maintain a collection of data items (although by different means). So most of them are defined as subclasses of the interface Collection.
- The Collection interface defines the baseline functionality a data structure or collection must have.
- Then specific Java interfaces, like List and Set, are implemented on top to define the baseline behaviours each kind of data structure should have.
- Concrete classes of data structure are built from these interfaces to define all abstract methods in a way appropriate to the data structure itself.

The Collection interface
- The Collection interface is the base class for all classes that implement data structures.
- Commonly used (abstract) methods:
  
  ```java
  public boolean add(Object obj)
  public void clear()
  public boolean contains(Object obj)
  public boolean isEmpty()
  public boolean remove(Object obj)
  public int size()
  public Object[] toArray()
  ```

The List interface
- List interface is a subinterface of Collection interface.
- All implementation classes of the list data structure are the subclasses of this List interface. They can maintain a collection of objects and the order among them. And it is possible to access each object by an index.
- A list can has any number of objects, and its storage/size can be extended whenever needed (unlike array viz. is fixed).
- Additional commonly used (abstract) methods:
  
  ```java
  public void add(int index, Object element)
  public Object remove(int index)
  public Object get(int index)
  public void set(int index, Object element)
  public int indexOf(Object element)
  public int lastIndexOf(Object element)
  ```

Implementation class of the List interface
- There are several implementation classes of the List interface in the java.util package, e.g. ArrayList, LinkedList, Vector.
- All these classes have implemented all the abstract methods defined in Collection and List.
- The add() method of the ArrayList class implicitly appends the object to the end of the list.
- Hence the implicit predefined ordering of an ArrayList object is the entry sequence.
- Order of adding the elements to it.
- Read the example TestArrayList.java
  
  ```java
  public void clear()
  public boolean contains(Object obj)
  public boolean isEmpty()
  public boolean remove(Object obj)
  public int size()
  public Object[] toArray()
  ```
Commonly used (abstract) methods:
Ø
- It maintains the mapping between these 2 sets, so that by
- Map object maintains 2 sets of data items:
- There are several implementation classes of the
- Set interface
- All these classes have implemented all the abstract
- Methods defined in Collection and Set.
- There is no order

Implementation class of the Set interface
Ø
- Set interface is a subinterface of Collection interface.
- All implementation classes of the set data structure are the
- They can maintain a collection of unique objects with no

Choosing suitable collection types and implementations

The Map interface
Ø
- Map object maintains 2 sets of data items:
- It maintains the mapping between these 2 sets, so that by
- Commonly used (abstract) methods:

Implementation class of the Map interface
Ø
- There are several implementation classes of the Map
- All these classes have implemented all the abstract
- Read the example TestMap.java

import java.util.*;
public class TestHashMap {  
    public static void main(String[] args) {  
        // Check program parameter
        if (args.length < 1) { // Show usage
            System.out.println("Usage: java TestHashMap <name>");
        } else {  
            // Prepare a List object (a ArrayList object)
            List objList = new ArrayList();
            objList.add(0, "Hello"); // Add element to the List object
            objList.add(1, args[0]);
            objList.add(2, "It is now ");
            objList.add(3, new Date());
            objList.add(4, ", ");
            // Show the objects maintained by the List one by one
            for (int i=0; i < objList.size(); i++) {
                System.out.print(objList.get(i));
            }
        }
    }
}

import java.util.*;
public class TestHashSet1 {  
    public static void main(String[] args) {  
        // Create a Set object
        Set objSet = new HashSet();
        objSet.add("Hello"); // Add objects to the set
        objSet.add("World");
        objSet.add(Hello); // this entry will be ignored
        objSet.add(0); // this entry will be ignored
        // Obtain an array object from the Set object
        Object[] elements = objSet.toArray();
        // Show the elements maintained by the Set object
        for (int i = 0; i < elements.length; i++) {
            System.out.println(elements[i]);
        }
    }
}

The Set interface
(pp14)

- Set interface is a subinterface of Collection interface.
- All implementation classes of the set data structure are the subclasses of this Set interface.
- They can maintain a collection of unique objects with no ordering among them.

Implementation class of the Set interface

- There are several implementation classes of the Set interface in java.util, e.g. TreeSet, HashSet.
- All these classes have implemented all the abstract methods defined in Collection and Set.
- Read the example TestHashSet1.java

import java.util.*;
public class TestHashSet1 {  
    public static void main(String[] args) {  
        // Create a Set object
        Set objSet = new HashSet();
        objSet.add("Hello");
        objSet.add("World");
        // Add objects to the set
        objSet.add("Hello");
        objSet.add("World");
        objSet.add(Hello); // this entry will be ignored
        objSet.add(0); // this entry will be ignored
        // Obtain an array object from the Set object
        Object[] elements = objSet.toArray();
        // Show the elements maintained by the Set object
        for (int i = 0; i < elements.length; i++) {
            System.out.println(elements[i]);
        }
    }
}

import java.util.*;
public class TestTreeMap {  
    public static void main(String[] args) {  
        // Create a TreeMap object
        Map objMap = new TreeMap();
        objMap.put("HK", "Hong Kong");
        objMap.put("KLN", "Kowloon");
        objMap.put("NT", "New Territories");
        // Show the contents of the Map objects
        System.out.println("Keys are: "+ objMap.keySet());
        System.out.println("Values are: "+ objMap.values());
    }
}

import java.util.*;
import java.util.*;
public class TestTreeSet {  
    public static void main(String[] args) {  
        // Create a TreeSet object
        Collection objSet = new TreeSet();
        objSet.add(0, "Hello");
        objSet.add(1, args[0]);
        objSet.add(2, "It is now ");
        objSet.add(3, new Date());
        objSet.add(4, ", ");
        // Show the objects maintained by the TreeSet one by one
        for (int i=0; i < objSet.size(); i++) {
            System.out.print(objSet.get(i));
        }
    }
}

The Map interface
(pp20)

- Map object maintains 2 sets of data items:
  - a key set maintaining all key objects and
  - a value set maintaining all value objects.
- It maintains the mapping between these 2 sets, so that by providing a key object the corresponding value object can be retrieved.
- Commonly used (abstract) methods:

Implementation class of the Map interface

- There are several implementation classes of the Map interface in java.util, eg. HashMap, TreeMap.
- All these classes have implemented all the abstract methods defined in Map.
- Read the example TestHashMap.java

Choosing suitable collection types and implementations

if ( need to maintain key/value mappings )
    use Map object
else if ( allows duplicated objects )
    if ( fixed number of data items )
        use Array object
else
    use List object
else
    use Set object

public class TestHashSet {  
    public static void main(String[] args) {  
        // Create a Set object
        Set objSet = new HashSet();
        objSet.add("Hello"); // this entry will be ignored
        objSet.add("World");
        // Add objects to the set
        objSet.add("Hello");
        objSet.add("World");
        objSet.add(Hello); // this entry will be ignored
        objSet.add(0); // this entry will be ignored
        // Obtain an array object from the Set object
        Object[] elements = objSet.toArray();
        // Show the elements maintained by the Set object
        for (int i = 0; i < elements.length; i++) {
            System.out.println(elements[i]);
        }
    }
}

public boolean hasNext( )
public Object next( )
public void remove( )

public class TestTreeSet {  
    public static void main(String[] args) {  
        // Create a TreeSet object
        Collection objSet = new TreeSet();
        objSet.add(0, "Hello");
        objSet.add(1, args[0]);
        objSet.add(2, "It is now ");
        objSet.add(3, new Date());
        objSet.add(4, ", ");
        // Show the objects maintained by the TreeSet one by one
        for (int i=0; i < objSet.size(); i++) {
            System.out.print(objSet.get(i));
        }
    }
}

public boolean hasNext( )
public Object next( )
public void remove( )

public class TestTreeMap {  
    public static void main(String[] args) {  
        // Create a TreeMap object
        Map objMap = new TreeMap();
        objMap.put("HK", "Hong Kong");
        objMap.put("KLN", "Kowloon");
        objMap.put("NT", "New Territories");
        // Show the contents of the Map objects
        System.out.println("Keys are: "+ objMap.keySet());
        System.out.println("Values are: "+ objMap.values());
    }
}

public boolean hasNext( )
public Object next( )
public void remove( )

public class TestTreeSet {  
    public static void main(String[] args) {  
        // Create a TreeSet object
        Collection objSet = new TreeSet();
        objSet.add(0, "Hello");
        objSet.add(1, args[0]);
        objSet.add(2, "It is now ");
        objSet.add(3, new Date());
        objSet.add(4, ", ");
        // Show the objects maintained by the TreeSet one by one
        for (int i=0; i < objSet.size(); i++) {
            System.out.print(objSet.get(i));
        }
    }
}

Itersors

To access each item in a collection, we can first convert it to an array of objects by using the method toArray(). Then a for-loop is applied on the array to iterate through all elements

Alternatively, the Collection interface provides a better and more elegant way to iterate through all items. This is called an iterator.

The iterator of a collection can be obtained by calling the iterator() method of the collection. This method returns an object implementing the Iterator interface.

Methods of the Iterator interface:

public boolean hasNext( )
public Object next( )
public void remove( )

Looping through a collection using an Iterator:

Collection collection = ...;
Iterator iterator = collection.iterator();
while (iterator.hasNext()) {
    Object item = iterator.next();
    // process the item
}

Comparing with the approach using an array:

Collection collection = ...;
Object[] items = collection.toArray();
for (int i=0; i<items.length; i++) {
    // process the item at items[i]
}
Designing a data structure: queue

A queue is a FIFO data structure (compare to a stack which is LIFO). It contains a storage to hold a collection of data items, and the storage is managed by 2 behaviours: enqueue and dequeue.

A queue is a collection and hence should have an attribute length to represent its size. And we should be able to query the length of a queue.

Similar to other collection classes, we should first build a Queue interface to define the baseline behaviour of this data structure.

We can then build a concrete class of queue by implementing the Queue interface.

A queue contains a list of objects, and hence we can implement a concrete queue class using the concrete LinkedList class. eg. LinkedQueue.java (pp27)

1. We can then build a concrete class of queue by implementing the Queue interface.
2. A queue contains a list of objects, and hence we can implement a concrete queue class using the concrete LinkedList class.

To revert the process (i.e. conversion from the wrapper object back to its primitive value), use the method primitiveValue() : eg.

\[
\text{int } i = \text{myInteger.intValue()};
\]

\[
\text{boolean } b = \text{myBoolean.booleanValue()};
\]

\[
\text{char } c = \text{myCharacter.charValue()};
\]

\[
\text{double } d = \text{myDouble.doubleValue()};
\]

Note that all wrapper objects are immutable and final

To construct an object from a primitive value, we can use the corresponding wrapper class constructor:

\[
\text{public WrapperClassName(Primitive Type value)}
\]

eg

\[
\text{Integer intObj = new Integer("10");}
\]

\[
\text{Double doubleObj = new Double("3.14");}
\]

\[
\text{To convert a wrapper object back to a string, use the toString() method (defined for all wrapper classes).}
\]

\[
\text{int } x = \text{new Integer("1234")}, \text{int Value}()
\]

\[
\text{Alternatively for Number wrapper objects, we can make use of the parsePrimitive() class method of the wrapper classes.}
\]

\[
\text{int } x = \text{Integer.parseInt("1234");}
\]

String class (_pp33)

String objects are immutable (i.e. once created, there is no way to change it unless creating another string).

Consider the following code, how many string objects are created? [ TestString.java (pp44 fig-10.25)]

\[
\text{String str = "";}
\]

\[
\text{for (int i=0; i<30000; i++)}
\]

\[
\text{str += i;}
\]

In this example, there is a large overhead of performing string concatenation. Hence if we need to perform string concatenation intensively, it can be a performance problem.

Java provides a StringBuffer class to solve this problem.

StringBuffer class (pp30)

In contrast to the String objects, StringBuffer objects are mutable. If we need to repeatedly concatenate strings, the StringBuffer object is more preferred for performance issues.

To create a StringBuffer object, we can use 3 constructors:

\[
\text{public StringBuffer()}\]

\[
\text{public StringBuffer(int length)}\]

\[
\text{public StringBuffer(String str)}\]

To change the content of a StringBuffer object, we can use these methods:

append(...), delete(int, int), deleteCharAt(int), insert(int….), replace(int, int, String), reverse()

There are 2 ways to convert a StringBuffer object to a String object:

\[
\text{String str = aStringBuffer.toString()}\]

\[
\text{String str = new String(aStringBuffer)}\]

What you should have done ...

Received all books
Read up to Unit 10

What to do next?
Finish any unfinished unit
Read the specimen exam paper (when it is released)
Review all course materials and prepare for the exam

About the last tutorial ...
No slide will be prepared
Discussion on exam issues and the specimen exam paper