CS310 - Advanced Data Structures and Algorithms

API, Collections

September 17, 2019
API and Encapsulation

- **Application Programming Interface (API):** a group of function specifications that are meant to provide some service or services to programs, clients of the API.

- **Data encapsulation:** Hiding the data of the implementation of the API functions so well that the client can't get at it by normal programming methods.

- They **have** to use the API functions to get their work done.
Example: C string library functions: strcpy, strcmp
Example: The Flips.java example from section 1.2 in S&W: Client code for Counter.java
API and Encapsulation

Implementation details and class variables

API

Nurit Haspel
Example – bank account.

Construct an account:
```java
BankAccnt ba = new BankAccnt('JJ', 234, 100);
```

Modify balance:
```java
ba.withdraw(20); ba.deposit(10);
```

Access balance:
```java
ba.balance; // Incorrect!!
ba.getBalance(); // Correct!!
```

The API is the method specification. An API lies between two bodies of code, the client code and the class implementation.
Java interface for clear implementation:

```java
public interface Account {
    // withdraw amt from this BA
    int withdraw(int amount);
    // deposit amt to this BA
    void deposit(int amount);
    // return balance for this BA
    int getBalance();
}
```
Any class that has balance going up and down can implement this interface:

```java
public class BankAccount implements Account {
    // constructor - create a BA
    public BankAccount(String nm, int _id, int bal) {...}
    // Account API functions
    public int withdraw(int amt) {...}
    public void deposit(int amt) {...}
    public int getBalance() {...}
    // Fields - all private
    private int id;
    private String name;
    private int balance;
}
```
Client code:

```java
// client code example:
public class TestBankAccount {
    static void main(String [] args) {
        BankAccount ba = BankAccount('JJ', 234, 100);
        ba.withdraw(20);
        ba.deposit(10);
    }
}
```
Advantages of Encapsulation

- Clear statement of functionality in use – what is intended/provided by the class.
- Partition of responsibility/code. Important if many programmers.
- Can share General Purpose objects across many apps. Saves coding. At worst have to make minor improvements later.
- Safety of contents. Primitive functions can check arguments, etc.,
- Debugability. No mystery changes to data – can breakpoint the functions that change data.
A collection is a container of objects.
A collection may be ordered or unordered.
It may or may not allow duplicates.
The Java Collection Interface lays down the foundation
List, Stacks, Queues, Priority Queues (CS210), Sets, Maps.
We will do a short review.
// Collection interface; the root of all 1.6 collections.
public interface Collection<AnyType> extends Iterable<AnyType>,
    java.io.Serializable
{
    int size(); // How many items are in this collection.
    boolean isEmpty(); // Is this collection empty?
    boolean contains(Object x); // is X in collection?
    boolean add(AnyType x); // Adds x to collection.
    boolean remove(Object x); // Removes x from collection.
    void clear(); // Change collection size to zero.
    // Obtains an Iterator object to traverse collection.
    Iterator<AnyType> iterator( );
    // Obtains a primitive array view of the collection.
    Object [] toArray();
    // Obtains a primitive array view of the collection.
    <OtherType> OtherType[] toArray(OtherType [] arr );
}
The Collections Interface

- Note how all the elements for a Collection are Objects, since a type parameter can only take on Object types.
- The only non-Object types are int, double, char, etc., the primitive types.
- However, each of these has a corresponding “wrapper” Object type: Integer, Double, etc., and autoboxing makes it easy to use these collections.
In Java, the collection object is fully encapsulated.

We can’t see the collection itself.

We are allowed to get references to the objects in the collection.

Each of them should be individually encapsulated.
// Iterator interface
public interface Iterator<AnyType> extends java.util.Iterator<AnyType>
{
    // Are there any items not iterated over
    boolean hasNext();
    // Obtains the next (as yet unseen) item in the collection
    AnyType next();
    // Remove the last item returned by next.
    // Can only be called once after next
    void remove();
}
A List is an ordered sequence of elements: $a_0, a_1, a_2, \ldots, a_{n-1}$.

Lists are discussed in Section 1.3 of S&W.
// List interface.
public interface List<AnyType> extends Collection<AnyType>
{
    // Returns the item at position idx
    AnyType get( int idx );

    // Changes the item at position idx.
    AnyType set( int idx, AnyType newVal );

    // Obtains a ListIterator object used to traverse the collection bi-directionally.
    ListIterator<AnyType> listIterator( int pos );
}
// ListIterator interface for List interface.
public interface ListIterator<AnyType> extends Iterator<AnyType>
{
    // Tests if there are more items in the collection when iterating in reverse.
    boolean hasPrevious();

    // Obtains the previous item in the collection when traversing in reverse.
    AnyType previous();

    // Remove the last item returned by next or previous.
    // Can only be called once after next or previous.
    void remove();
}
The two most important classes that implement the List interface are LinkedList and ArrayList.

- They have different performance for large lists.
- Both have extra methods over and above the List interface.
Here is a 4 member list:

- We can get(0), ..., get(3) and access any particular object ref.
- We can set(0, b) and replace the object at 0 with b.
- What happens if we set(4, b)?
- To grow the list we need to use add(Object x), but where does it go??
- This is fast because the LinkedList tracks the end-of-list.
A ListIterator starting from 0, has a next method that returns element 0 on first call, element 1 on second, etc.

Should test with hasNext before doing a next.

If hasNext returns false, the iterator is at end of list (EOL).

It starts at beginning of list, so there are 5 different iterator states for 4 elements:

\[ \langle A0 \rangle \langle A1 \rangle \langle A2 \rangle \langle A3 \rangle \]

↑ original iterator, before element 0

↑ after first next, returning A0
  (just before element 1)

↑ after 2nd next, returning A1

↑ after 3rd next, returning A2

↑ after last element
You may think of an iterator as sitting between two elements, so to speak.

At each point in time, an iterator is positioned just after the element that would be returned by previous(), and just before the element that would be returned by next().

A ListIterator can go both ways.

When we talk about numerical position in a list, it’s normally about the position of an element, not directly the iterator.
Mental Model of a ListIterator

- With the `listIterator(int pos)` method, the `pos` determines an element position and the method returns an iterator positioned prior to that element, or at EOL if `pos == size of list`.

- `listIterator(0)` gives an iterator positioned before the very first element, etc.

- A special case is that extra position after all the elements: this is attained by using `N` as an arg, the number of elements in the list.

- **Remember: an iterator has `N+1` possible positions for `N` elements.**
What happens if next returns A1, then another next returns A2, and then a previous is done – is A1 returned?
What happens if next returns A1, then another next returns A2, and then a previous is done – is A1 returned?

No! We’ve just gone past A2 one way, and now we go back across it again, so A2 gets returned again.
```java
class TestArrayList {
    public static void main( String[] args ) {
        ArrayList<Integer> lst = new ArrayList<Integer>();
        lst.add(2);
        lst.add(4);
        ListIterator<Integer> itr1 = lst.listIterator(0);
        System.out.print("Forward: ");
        while (itr1.hasNext())
            System.out.print(itr1.next() + " ");
        System.out.println();
        System.out.print("Backward: ");
        while (itr1.hasPrevious())
            System.out.print(itr1.previous() + " ");
        System.out.println();
        System.out.print("Backward: ");
        ListIterator<Integer> itr2 = lst.listIterator(lst.size());
        while (itr2.hasPrevious())
            System.out.print(itr2.previous() + " ");
        System.out.println();
        System.out.print("Forward: ");
        for (Integer x : lst)
            System.out.print(x + " ");
        System.out.println();
    }
}
```
Here the ListIterator<Integer> goes all the way down the list to EOL, then back along the list, so the turn-around occurs at EOL.

Another ListIterator<Integer> starts from “lst.size()”, which would be 4 for our list.

This is an artificial element number denoting the EOL position of the iterator.

Again, there are n+1 different iterator states for n elements, and these are numbered from 0 to n.
An iterator sits between elements.

When calling remove, which nearby element gets removed?

The object removed is the last one returned by next or previous, and only one remove per movement-action is allowed.

What happens if you next, remove, and then next again?

You access the element just after the removed element. Because we’ve moved past the deleted element already, the iterator position is clear.

If you next, remove, previous, you should get the previous-to-removed. And so on, using the model above.
Example: Using two iterators to remove duplicates from a LinkedList doesn't work!

Algorithm: scan LinkedList list with itr, from list.iterator()
For each itr-position, with element o1, initialize a ListIterator (o2) at that position.
Scan rest of list with o2, removing elements that equal o1.
The exception is ConcurrentModificationException.
Removing Duplicates From a List With Two Iterators

List<Order> list = new LinkedList<Order>(); // or ArrayList
// add some elements to the list: Foo, Bar, Bar, Bam
Iterator<Order> itr = list.iterator();
int position = 0;
System.out.println("about to do next() in outer loop... list is" + list);
// throws ConcurrentModificationException after first remove
Order o1 = itr.next();
System.out.println("working on " + o1);
ListIterator<Order> listItr = list.listIterator(position + 1);
while (listItr.hasNext()) {
    Order o2 = listItr.next();
    System.out.println("inner loop o2 =" + o2);
    if (o1.equals(o2)) {
        System.out.println("removing o2 =" + o2);
        listItr.remove();
    }
}
position++;

Nurit Haspel
CS310 - Advanced Data Structures and Algorithms
Output:
List with Foo, Bar, Bar, Bam:
about to do next() in outer loop... list is[Foo, Bar, Bar, Bam]
working on Foo
inner loop o2 =Bar
inner loop o2 =Bar
inner loop o2 =Bam
about to do next() in outer loop... list is[Foo, Bar, Bar, Bam]
working on Bar
inner loop o2 =Bar
removing o2 =Bar
inner loop o2 =Bam
about to do next() in outer loop... list is[Foo, Bar, Bam]
Exception in thread "main" java.util.ConcurrentModificationException
at java.util.AbstractList$Itr.checkForComodification(Unknown Source)
at java.util.AbstractList$Itr.next(Unknown Source)
at ListTest.main(ListTest.java:20)
Safely Removing Duplicates

- Drop the outer iterator, just get the value of the into a variable.
- Run an internal loop with an iterator, removing all elements that are equal to that value.
- What to do with huge lists, when using get and/or remove in inner loop means $O(n^2)$ or worse?

You can use HashSet $h = \text{new HashSet(list)}$; //Set means no dups, $O(n)$ 
Then put result back in a list.

Another way: toArray, then sort, then pick off unique values, $O(n \log n)$.
• Drop the outer iterator, just get the value of the into a variable.

• Run an internal loop with an iterator, removing all elements that are equal to that value.

• What to do with huge lists, when using get and/or remove in inner loop means $O(n^2)$ or worse?

• Abandon lists!

• You can use HashSet $h = \text{new HashSet(list)}$; //Set means no dups, $O(n)$

• Then put result back in a list.

• Another way: toArray, then sort, then pick off unique values, $O(n \log n)$
Performance of LinkedList vs. ArrayList

For ArrayList of size n
- Get, set are very fast, $O(1)$
- Append-type add is fast most of the time. If it involves array expansion, it is expensive, $O(n)$.
- Delete is expensive unless if it is at the end.

For LinkedList of size n
- Get, set depends on the index position
- get(1) is done by two next’s down the list from the beginning of the list, and get(n-2) is done by two previous next’s from the end of the list
- Most expensive is get(n/2)
- Delete/add is easy once the right spot in the list is located. Remove in an iterator is $O(1)$, but the larger task may involve $O(n)$ next’s to get the iterator positioned.
A Stack is a specialized List where can only insert (push), retrieve (top), and delete (pop) elements at one end.

A Queue is a specialized list where insert at one end, retrieve and delete at the other.
A set contains a number of elements, with no duplicates and no order.

A = \{1, 5, 3, 96\}, or B = \{17, 5, 1, 96\}, C=\{ “Mary”, “contrary”, “quite” \}.

Incorrect - \{ “Mary”, “contrary”, “quite”, “Mary” \}.

In Java, the Set interface is the Collection interface.

The API isn’t sensitive to the lack of duplicates, only the implementation. The implementations in the JDK are the TreeSet and HashSet.

They check for duplicates by using the equals method of the elements.

HashSet uses hashCode() and TreeSet uses compareTo().
public static void main(String[] args) {
    Set<String> s = new TreeSet<String>();
    // For reverse order:
    TreeSet<String>(Collections.reverseOrder());
    s.add("joe");
    s.add("bob");
    s.add("hal");
    printCollection(s);
}
Sets of JDK Element Type

- Note that a pure set is supposed to be without order, and here we are seeing order imposed by the TreeSet.
- It’s an extra feature, so the TreeSet gives us a SortedSet.
- We can just ignore the order if we want.
- The TreeSet gives a high-performance implementation, competitive with HashSet.
Sets of type String, Integer, etc. are very easy to use, JDK classes all have appropriate equals, hashCode, and compareTo (they implement Comparable<E>).

Ex: Simple set app using element type String.

If we add “hal” again, no difference in resulting Set.

```java
public static void main(String[] args)
{
    Set<String> s = new HashSet<String>();
    s.add("joe");
    s.add("bob");
    s.add("hal");
    printCollection(s);
}
```
If we use our own class for the element type, we have to make sure equals, and hashCode or compareTo are in good enough shape to work properly when called by HashSet or TreeSet on the element objects.

Consistency requirements: equals and hashCode must be consistent, so that if a.equals(b), then a.hashCode() == b.hashCode().

Also equals and compareTo must be consistent, so that if a.equals(b), then a.compareTo(b) == 0.
class BaseClass {
    public BaseClass(int i)
    {
        x = i;
    }
    public boolean equals(Object rhs)
    {
        if(rhs == null || !rhs.instanceof(BaseClass))
            return false;
        return x == ((BaseClass) rhs)).x;
    }
    private int x;
}

class DerivedClass extends BaseClass {
    public DerivedClass(int i, int j)
    {
        super(i); y = j;
    }
    public boolean equals(Object rhs)
    {
        return super.equals(rhs) &&
        y == ((DerivedClass) rhs).y;
    }
    private int y;
}

What could go wrong and how do we fix it?
class BaseClass {
    public BaseClass(int i) {
        x = i;
    }
    public boolean equals(Object rhs) {
        if (rhs == null || getClass() != rhs.getClass())
            return false;
        return x == ((BaseClass) rhs).x;
    }
    private int x;
}

class DerivedClass extends BaseClass {
    public DerivedClass(int i, int j) {
        super(i); y = j;
    }
    public boolean equals(Object rhs) {
        return super.equals(rhs) &&
            y == ((DerivedClass) rhs).y;
    }
    private int y;
}
Question: is BaseClass in good shape to make HashSet<BaseClass>?

Answer: No, it overrides equals, but not hashCode. If it overrode neither, it would be OK, but once equals is overridden, you need to override hashCode too, and consistently.

Fix: drop equals(), or add:

```java
public int hashCode()
{   return Integer(x).hashCode();
} 
```
Question: is BaseClass in good shape to make HashSet<BaseClass>?

Answer: No, it overrides equals, but not hashCode.

If it overrode neither, it would be OK, but once equals is overridden, you need to override hashCode too, and consistently.

Fix: drop equals(), or add:

```java
public int hashCode()
{
    return Integer(x).hashCode();
}
```
Question: is BaseClass in good shape to make “new TreeSet<BaseClass>()”? 

Fix: add "implements Comparable<BaseClass>", and method compareTo:

```java
public int compareTo(BaseClass b) { return Integer(x).compareTo(b.x); }
```
Question: is BaseClass in good shape to make “new TreeSet<BaseClass>()”? 

Answer: No, it doesn’t implement Comparable<BaseClass>.

Fix: add “implements Comparable<BaseClass>”, and method compareTo:

public int compareTo(BaseClass b)
{ return Integer(x).compareTo(b.x); }
/** Students are ordered on basis of id only. */
class SimpleStudent implements Comparable<SimpleStudent> {
    public SimpleStudent(String n, int i) { name = n; id = i; }
    public boolean equals(Object rhs) {
        if (rhs == null || getClass() != rhs.getClass())
            return false;
        SimpleStudent other = (SimpleStudent) rhs;
        return id.equals(other.id);
    }
    public int compareTo(SimpleStudent other) {
        return id.compareTo(other.id);
    }
    public int hashCode() {
        return id.hashCode();
    }
}
- The unique identifier here is the id.
- Students can have the same name and still be considered different objects.
- We can drop equals and hashCode from the class implementation.
- In this case, objects will be compared.
- But we can’t drop only one of the two! The behavior will be unpredictable.
In most cases the identifier is one class member.

This class member is often a simple data type with a built-in `equals`, `hashCode` and `compareTo`

Notice consistency requirement: `compareTo == 0` for equals objects.

if equals returns yes, `hashCode` should be the same, Not necessarily the other way around.

Sometimes (rarely) more than one field is needed.
public final class PhoneNumber { // Example from S&W, Section 1.2
    private final int area, exch, ext;

    public PhoneNumber(int area, int exch, int ext) {
        this.area = area; this.exch = exch; this.ext = ext;
    }

    public boolean equals(Object other) {
        if (other == this) return true;
        if (other == null) return false;
        if (other.getClass() != this.getClass()) return false;
        PhoneNumber that = (PhoneNumber) other;
        return (this.area == that.area) && (this.exch == that.exch) && (this.ext == that.ext);
    }

    public int hashCode() {
        return 31 * (area + 31 * exch) + ext;
    }
}
Say we had a Set of Student records and we wanted to look up a student (by name or student ID).

No efficient way to pull out an entire record in a Set.

We could iterate over the whole Set, and find the match, but that’s $O(N)$, and we want something faster.

The Map interface allows us to do that.
Maps – Definition

- Given two sets, Domain and Range, with a relation from one to another.
- Like a math function, each domain element has associated with it exactly one range element.
- Two arrows can land on the same range element, but one domain element cannot have two arrows out of it.
The action of following the arrow is often known as a “lookup” action.

For ex., employee records are looked up by social-security no. and/or by employee name.

Social security numbers or employee names are the Domain, Employee objects are the Range.

In programming, Maps are lookup tables.

We are mapping integers to employee objects or Strings to employee objects.

Mapping creates a pair of $<\text{DomainType}, \text{RangeType}>$.

The DomainType is a key, the RangeType is a value.
A simple example: mapping numbers to letter grades

- 92 → ‘A’
- 79 → ‘B’
- 68 → ‘C’

is an integer and the RangeType is string (could be a character but we want to account for all letter grades like ”A-” etc).

Each of these lines can be called a “key/value pair”, or just “pair”.

(92, ”A”) is a pair of the grade 92 (the key) and the string ”A” (the value)

The whole mapping is the set of these 3 pairs.

\[ M = \{ (92, “A”), (79, “B”), (68, “C”) \} \] – a map as a set of pairs, or “associations”
A mapping is a collection like other collections we are studying, lists, stack, queues, and sets.

However, in Java a Map has its own interface separate from Collection.

Note that not every collection of pairs makes a proper map: M qualifies as a map only if the collection of keys has no duplicates, i.e., constitutes a Set.

The collection of values can have repetitions, so it is not a Set, just a collection.
// Map interface.
public interface Map<KeyType, ValueType> extends Serializable {
    // Returns the number of keys in this map.
    int size();
    // Tests if this map is empty.
    boolean isEmpty();
    // Tests if this map contains a given key.
    boolean containsKey(KeyType key);
    // Returns the value in the map associated with the key.
    ValueType get(KeyType key);
    // Adds the key value pair to the map, overriding the
    // original value if the key was already present.
    ValueType put(KeyType key, ValueType value);
    // Remove the key and its value from the map.
    ValueType remove(KeyType key);
    // Removes all key value pairs from the map.
    void clear();
    // Returns the keys in the map.
    Set<KeyType> keySet();
    // Returns the values in the map. There may be duplicates.
    Collection<ValueType> values();
    // Return a set of Map.Entry objects corresponding to
    Set<Entry<KeyType, ValueType>> entrySet();
}
/**
 * The interface used to access the key/value pairs in a map.
 * From a map, use entrySet().iterator to obtain a iterator
 * over a Set of pairs. The next() method of this iterator
 * yields objects of type Map.Entry.
 */

public interface Entry<KeyType,ValueType> extends Serializable {
    // Obtains this pair’s key.
    KeyType getKey( );
    // Obtains this pair’s value.
    ValueType getValue( );
    // Change this pair’s value.
    ValueType setValue( ValueType newValue );
}
We can add a key/value pair to a Map, and this operation is called put in Java, and we can lookup the associated range element (value) of any given domain element (key), and this action is called get.

Put is more pushy than “set” for Lists – it can put another fact into the collection rather than just changing one that’s there.

Like sets, Java supports two main implementations: TreeMap and HashMap.
Ways of Thinking About Maps

- As holding conversions, like codes to grades, social security number to name.
- As generalized arrays.
- As math functions: \( y = f(x) \) is a map.
- As a “database” with key lookup: SSN to employee record, ISBN to book record, name to inventory record.
Often data are processed in a specific order, the priority

Examples
- In OS process scheduling, a process comes with a priority, and high priority processes are executed earlier than low priority ones
- In network routing, high priority traffic (VoIP, IPTV) are delivered before other traffic

We need to dynamically maintain the pending jobs
- New jobs keep coming in
- When a server becomes available, the highest-priority job is removed from PQ and serviced

Convention: non-negative, and a smaller value has a higher priority
Operations of a Priority Queue

- Insert($Q$, item, priority)
- FindMinimum($Q$)
- DeleteMinimum($Q$)

Implementation by different data structures, worst case runtime:

<table>
<thead>
<tr>
<th></th>
<th>Unsorted array</th>
<th>Sorted array</th>
<th>Linked list</th>
<th>Balanced tree</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>FindMin</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>DeleteMin</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

- The trick for arrays and lists: in addition to the data structure, keep a pointer that points to the current minimum