CS310 – Advanced Data Structures and Algorithms

Spring 2020 – API, Collections

February 13, 2020
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
Example – bank account.

Construct an account:
BankAccnt ba = new BankAccnt(‘JJ’, 234, 100);

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

Access balance:
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:

- All its methods are public, and it can have constants as well.
- It can’t be instantiated, i.e., we can’t use ”new Account()”
- A class is said to implement this interface if it provides definitions (i.e. code) for all these methods.

```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 Account interface. Here BankAccount is a subtype of Account, i.e., BankAccount ISA Account

```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: a BankAccount ISA Account

// client code example, v1:
public class TestBankAccount {
    public static void main(String[] args) {
        BankAccount ba = new BankAccount('"JJ"', 234, 100);
        ba.withdraw(20);
        ba.deposit(10);
    }
}

// client code example, v2: ba can have type Account here
public class TestBankAccount {
    public static void main(String[] args) {
        Account ba = new BankAccount('"JJ"', 234, 100);
        ba.withdraw(20);
        ba.deposit(10);
    }
}

// Note: Those marks around JJ are supposed to be double quotes
Type Hierarchies and ISA

- A class A ISA B if A is the same type as B, or its type extends-B or A’s class implements B, an interface.
- Similarly an interface A ISA B if A is the same interface as B or extends B as an interface.
- Note that it is an asymmetric relationship: A ISA B does not mean B ISA A
- If A ISA B, then we say A is type-compatible with B.
- We can assign $b = a$, where a is an A and b is a B object.
- i.e., a ref b can refer to a B or any type that ISA B
- Object references can be tested for type compatibility with instanceOf: if “a instanceOf B” is true, then $b = a$ will work.
- But we try to write code that doesn’t use instanceOf: the types should be clear from context.
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.
What about BankAccount equality testing?

- Note that BankAccount.java has no equals method
- What happens if ba.equals(ba1) is executed?
- Answer: since ba ISA Object, Object.equals() is called.
- How does Object.equals work?
- Answer: it just compares the two refs, ba and ba1, for equality.

...so it returns true only if ba and ba1 are exactly the same object, not if they just agree on all fields.

- If you want field-testing equality, you need to override Object.equals by implementing it in BankAccount.java.
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, Sets, Maps.
See Weiss Chap. 6 for textbook coverage, or online JDK API.
The JDK Collection Interface is not covered in S&W.
But many of the individual collection abstractions are: lists, stacks, queues, priority queues, symbol tables (i.e. Maps), Sets as specialized STs.
The Collection Interface

// Collection interface; the root of all JDK 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 Collection Interface vs. a Collection class

- So far we have seen the interface for Collection
- We need an implementing class to actually use it, for example ArrayList, the stretchy array collection
- Then we can put

```java
ArrayList<BankAccount> myList = new ArrayList<BankAccount>();
```

- `myList` is now an object of type `ArrayList<BankAccount>`, and also of type `Collection<BankAccount>`, since an `ArrayList` ISA `Collection`
- ...and we know it has methods size, isEmpty, contains, add, ... because it is a Collection
- ...and it can have additional methods too, as we will see
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.
import java.util.ArrayList;
public class TestArrayList {

    public static void main( String [] args ) {
        ArrayList<String> array = new ArrayList<String>();
        array.add("apple");
        array.add("banana");
        // Loop through the collection by index
        for ( int i = 0; i < array.size(); i++ ) {
            System.out.println( array.get( i ) );
        }
        // Using the fact that Collections are Iterable
        for (String s: array) {
            System.out.println(s);
        }
    }
}
Encapsulation of Collection Objects

- In Java, the collection object (ArrayList or whatever) 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();
}

Note that the Collection interface has a method named "iterator" that provides an iterator for the collection. For example, we can get an iterator for myList by calling mylist.iterator(). This returns an Iterator<BankAccount>, helpful for iterating through the BankAccounts in the list.
import java.util.ArrayList;
import java.util.Iterator;
public class TestArrayList2 {

    public static void main( String[] args ) {
        ArrayList<String> array = new ArrayList<String>();
        array.add("apple");
        array.add("banana");
        // Loop through the collection by using an Iterator
        Iterator<String> itr = array.iterator();
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }
    }
}
A List is an ordered sequence of elements: \( a_0, a_1, a_2, \ldots, a_{n-1} \).

Simple Lists are discussed in Section 1.3 of S&W.

JDK Lists are fully encapsulated, so we never deal directly with the references from item to item.

As you may have guessed, a List ISA Collection
// 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 );
}

// For example, since ArrayList ISA List we can write--
List<BankAccount> myList = new ArrayList<BankAccount>();

Don't forget all those Collection methods every List gets by the fact that a List ISA Collection. Also the Object methods equals and hashCode since a List ISA Object.
// 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();
}

Lists have more powerful iterators than non-List Collections. We can iterate up and down the list. We can start from either end by using the pos argument of List.listIterator(pos).
import java.util.ArrayList;
import java.util.ListIterator;
public class TestArrayList3 {
    public static void main( String [ ] args )
    {
        ArrayList<String> array = new ArrayList<String>();
        array.add("apple");
        array.add("banana");
        // Loop through the collection by using a ListIterator
        ListIterator<String> itr = array.listIterator();
        while (itr.hasNext()) {
            System.out.println(itr.next() );
        }
        // going backwards:
        ListIterator<String> itr2 = array.listIterator(array.size());
        while (itr2.hasPrevious()) {
            System.out.println(itr2.previous() );
        }
        // The listIterator is still alive--
        System.out.println(itr2.next());
    }
}
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.
- But by far their most important methods are in the Collection plus List interfaces.
- Because of this, it is common to use List type instead of ArrayList:

```java
List<BankAccount> myList = new ArrayList<BankAccount>();
```
Here is a 4 member list:

\[ A_0 \leftrightarrow A_1 \leftrightarrow A_2 \leftrightarrow A_3 \leftrightarrow \text{NULL} \]

- 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:

\[
< A_0 > < A_1 > < A_2 > < A_3 >
\]

↑ 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 ListlIterator 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.
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.
Question

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.
class TestArrayList4
{
    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( );
    }
}

N. Haspel, E. O’Neil  CS310 – Advanced Data Structures and Algorithms
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.
Two Iterators: ConcurrentModificationException!

List<Order> list = new LinkedList<Order>(); // or ArrayList
// add some elements to the list: 100, 200, 100, 400
Iterator<Order> itr = list.iterator();
System.out.println("about to do next() in outer loop... list is" + list);
Order o1 = itr.next();
System.out.println("outer loop o1 iterator 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(); // this works OK
    }
}
System.out.println("Now advance outer loop o1 iterator...");
if (itr.hasNext())
    itr.next(); // this throws!!
Output:
about to do next() in outer loop... list is[100, 200, 100, 400]
outer loop o1 iterator working on 100
inner loop o2 =200
inner loop o2 =100
removing o2 =100
inner loop o2 =400
Now try to advance outer loop o1 iterator...
Exception in thread "main" java.util.ConcurrentModificationException at java.base/java.util.LinkedList.ListItr.checkForComodification(LinkedList.java:970) at java.base/java.util.LinkedListListItr.next(LinkedList.java:892) at ListRemove.main(ListRemove.java:34)
Note that for this example, the Order object needs an equals method.
Otherwise, the code would not find duplicates at all!
We can just test the order id between two Orders.
But don’t forget we need to return false if the other object isn’t an Order.
See the code on p. 103 for an example of equals.
When you code equals, you should also code hashCode.
Use @Override to make compiler check method really overrides.
public class Order {
    private int id; // unique identifier, basis of equality
    // other fields
    public Order(int i) { id = i; }
    public int getId() { return id; }
    @Override
    public boolean equals(Object other) {
        if (this == other) return true;
        if (other == null) return false;
        if (this.getClass() != other.getClass())
            return false;
        Order o = (Order)other;
        return id == o.id;
    }
    @Override
    public int hashCode() { return Integer.valueOf(id).hashCode(); }
    public String toString() { return ""+id; }
}
Safely Removing Duplicates

- Drop the outer iterator, just get the value of the element 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?

N. Haspel, E. O’Neil

CS310 – Advanced Data Structures and Algorithms
• Drop the outer iterator, just get the value of the element 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 = 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 we can only insert (push), retrieve (top or peek), and delete (pop) elements at one end.

A Queue is a specialized list where we insert at one end (enqueue), retrieve and delete (dequeue) at the other.
There is a Stack class in the JDK, but its performance is not good.

The JDK docs suggest use of ArrayDeque (interface Deque) instead

Deque ISA Collection, so this is a "wide" interface

We can create a Stack class and implement it with a LinkedList to stick to its textbook model.
There is a Queue interface in the JDK

The JDK docs list various implementing concrete classes including ArrayDeque

a Queue ISA Collection, so this is a ”wide” interface

In fact, a Queue ISA Deque ISA Collection

so a Queue is implemented easily with an ArrayDeque.

We can create a Queue class and implement it with a LinkedList to stick to its textbook model.
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.

Additionally, HashSet uses hashCode() and TreeSet uses compareTo().
import java.util.Set;
import java.util.TreeSet;
public class TestTreeSet {
    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");
        System.out.println(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
import java.util.Set;
import java.util.HashSet;
public class TestHashSet {
    public static void main(String[] args) {
        Set<String> s = new HashSet<String>;
        s.add("joe");
        s.add("bob");
        s.add("hal");
        System.out.println(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`. 
public class Order {
    private int id; // unique identifier, basis of equality
    // other fields
    public Order(int i) { id = i; }
    public int getId() { return id; }
    @Override
    public boolean equals(Object other) {
        // code on pg. 103, adapted
        if (this == other) return true;
        if (other == null) return false;
        if (this.getClass() != other.getClass())
            return false;
        Order o = (Order)other;
        return id == o.id;
    }
    @Override
    public int hashCode() {
        return Integer.valueOf(id).hashCode();
    }
}
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.

In Order.java, equals and hashCode are consistent, so that if a.equals(b), then a.hashCode() == b.hashCode.

But there is no compareTo method

So we can use HashSet<Order> but not TreeSet<Order>
Let’s fix up Order so we can use TreeSet<Order>...

We need Order to be Comparable, actually Comparable<Order>

compareTo is easier to implement than equals because its method has the generic type

Again we use @Override to check our work
public class Order implements Comparable<Order> {
    private int id; // unique identifier, basis of equality
    // other fields
    public Order(int i) { id = i; }
    public int getId() { return id; }
    @Override
    public boolean equals(Object other) {
        // as before
    }
    @Override
    public int hashCode() { return Integer.valueOf(id).hashCode(); }
    @Override
    public int compareTo(Order other) {
        return Integer.valueOf(id).compareTo(other.id);
    }
    public String toString() { return ""+id; }
}
import java.util.Set;
import java.util.TreeSet;
public class TestTreeSet {
    public static void main(String[] args) {
        Set<Order> s = new TreeSet<Order>();
        s.add(new Order(600));
        s.add(new Order(100));
        s.add(new Order(200));
        s.add(new Order(100));
        System.out.println(s); // only 3 orders show--Why? What order do they print?
    }
}
Recall BankAccount, a simple class with no equals, hashCode, compareTo

– Can we set up a Set\(<\text{BankAccount}\)> without changing BankAccount?

We can’t use TreeSet because there’s no compareTo

But all classes have equals and hashCode from Object

So we can use HashSet\(<\text{BankAccount}\)>

We just need to realize it could have two BankAccount objects with equal field values
import java.lang.Integer;
class SimpleStudent implements Comparable<SimpleStudent> {
    private int id; private String name;
    public SimpleStudent(String n, int i) { name = n; id = i; }
    // getters and setters for name and id go here
    public boolean equals(Object rhs) { // more compact code,
        but still correct
        if (rhs == null || getClass() != rhs.getClass())
            return false;
        SimpleStudent other = (SimpleStudent) rhs;
        return id == other.id;
    }
    public int compareTo(SimpleStudent other)
    {
        return
            Integer.valueOf(id).compareTo(Integer.valueOf(other.id));
    }
    public int hashCode()
    {
        return Integer.valueOf(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 can still be compared.

But we can’t drop only one of the two! The behavior will be unpredictable.

And don’t forget, we need equals for HashSet and TreeSet.
In many cases the identifier is one instance variable (or field).

This instance variable 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 true, hashCodes should be the same, Not necessarily the other way around.

Sometimes (uncommonly) more than one instance variable is needed.
public final class PhoneNumber {
    // 3 fields, all important to define phone no.
    private final int area, exch, ext;

    public PhoneNumber(int area, int exch, int ext) {
        this.area = area; this.exch = exch; this.ext = ext;
    }
    // getters for area, exch, ext go here
    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; // or use another prime here
    }
}
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. S&W calls it a symbol table or ST.
Actually S&W’s ST is a class, not an interface, but its API is what we’re talking about.
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 (i.e. STs) are lookup tables.

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

Mapping creates a pair of \( \langle \text{DomainType}, \text{RangeType} \rangle \).

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 Map Interface’s Nested Interface

/**
 * This is part of Map.java, but didn’t fit in last slide
 * 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.
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”
import java.util.Map;
import java.util.HashMap;

public class TestMap {
    public static void main(String[] args) {
        Map<Integer,String> map = new HashMap<Integer,String>();
        map.put(92, "A");
        map.put(79, "B");
        map.put(68, "C");
        System.out.println(map);
    }
}

C:\cs310\TestMap>java TestMap
{68=C, 92=A, 79=B}

This shows that HashMap has a toString() to print out its contents. So does TreeSet
import java.util.Map;
import java.util.Set;
import java.util.HashMap;

public class TestMap1 {
    public static void main(String[] args) {
        Map<Integer, String> map = new HashMap<Integer, String>();
        map.put(92, "A");
        map.put(79, "B");
        map.put(68, "C");
        System.out.println(map);
        Set<Map.Entry<Integer, String>> entries = map.entrySet();
        System.out.println(entries);
        for (Map.Entry<Integer, String> e : entries) {
            System.out.println(e);
        }
    }
}
Map Example, showing entrySet, with output

... (cide from last slide)
    System.out.println(map);
    Set<Map.Entry<Integer,String>> entries = map.entrySet();
    System.out.println(entries);
    for (Map.Entry<Integer,String> e: entries) {
        System.out.println(e);
    }

C:\cs\cs310\TestMap>java TestMap1
{68=C, 92=A, 79=B} for map, a Map<Integer, String>
[68=C, 92=A, 79=B] for entries, a Set<Map.Entry<...>>
68=C
92=A
79=B

This shows that the HashMap provides the entrySet Set and its Map.Entry elements with toString(), for our convenience in displaying these data structures.
Ways of Thinking About Maps

- As holding conversions, like codes to grades, social security number to name.
- As generalized arrays. An array maps 0 to a[0], 1 to a[1], etc., very restricted map.
- As math functions: $y = f(x)$ is a map: each $x$ to $f(x)$.
- As a “database” with key lookup: SSN to employee record, ISBN to book record, name to inventory record.
Maps are Sets with a Value attached to each element

- So the elements, i.e. the key objects, need the same equals/hashCode/compareTo treatment as in HashSet/TreeSet
  - We need equals and hashCode for key objects of HashMap
  - We need equals and compareTo for key objects of TreeMap
- The value objects don’t need any of these methods implemented
- Note: if we ignore the values, a Map provides a ready-made set
So far, we've looked at "toy" applications: just put things in Set or Map, then read them back.

Let's look at S&W for more app ideas:

- Set: Dedup, p. 490: read words from a text file, report each once only.
- Set: WhiteFilter, pg. 491: read special words from one file, report on their use in a text.
- Map: FrequencyCounter, pg. 372: find #occurances of each word in a text.
- Map: LookupCSV, pg. 495: read key-value pairs (one value for each key) from a file, then lookup keys for user.
We need to convert S&W code to JDK classes

- StdIn.readString, StdIn.isEmpty() : we use Scanner
- S&W HashSet: we use HashSet (or TreeSet), which ISA Set
- S&W ST: we use HashMap (or TreeMap), which ISA Map
- Look at the handout on FrequencyCounter.java to see a full example
### Performance of JDK Sets

<table>
<thead>
<tr>
<th>method</th>
<th>JDK HashSet(&lt;E&gt;)</th>
<th>JDK TreeSet(&lt;E&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>size(), isEmpty()</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>contains</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>add</td>
<td>$O(1)^*$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>clear</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

* assuming good hash function and large-enough hash table
* Hash Table resize costs $O(n)$
* So best to initialize hash table size to estimated needed size
* new HashSet(1000000) for example
# Performance of JDK Maps

<table>
<thead>
<tr>
<th>method</th>
<th>JDK HashMap(&lt;E&gt;)</th>
<th>JDK TreeMap(&lt;E&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>size(), isEmpty()</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>containsKey</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>get</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>put</td>
<td>$O(1)^*$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>clear</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>keySet</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

* assuming good hash function and large-enough hash table

- Can use new HashMap(1000000) for example
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
There is a Queue interface in the JDK, a subinterface of Collection.

The JDK docs list various implementing concrete classes including ArrayDeque, previously listed here, and PriorityQueue.

So a PriorityQueue ISA Collection, so it has a "wide" interface.

Thus a PriorityQueue ISA Queue ISA Collection.

The element class needs to implement compareTo to compare priorities of elements, or, for the JDK class, a Comparator can be provided.
Operations of a Priority Queue

- `insert(item)` insert into proper place in PQ
- `FindMin()` return min element by priority
- `DeleteMin()` delete and return min element by priority
- `isEmpty()` and/or `size()` also needed

<table>
<thead>
<tr>
<th></th>
<th>S&amp;W MinPQ(p. 309)</th>
<th>JDK PriorityQueue&lt;E&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td><code>insert(item)</code></td>
<td><code>add(item)</code>, <code>offer(item)</code>, <code>same</code></td>
</tr>
<tr>
<td>FindMin</td>
<td><code>Key min()</code></td>
<td><code>Key peek()</code></td>
</tr>
<tr>
<td>DeleteMin</td>
<td><code>Key delMin()</code></td>
<td><code>Key poll()</code></td>
</tr>
</tbody>
</table>
### Performance of Operations of a Priority Queue

- 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>JDK/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.

- Note that since Insert and DeleteMin are $O(n)$, can sort $n$ items in $O(n \log n)$ time by Insets, then DeleteMins.
PriorityQueue Examples

- Recall SimpleStudent, with equals, hashCode, and compareTo based on id, an integer
- We could add students to a priority queue, then loop over DeleteMins
- The first one out would be the smallest id, then the next, ...
- But this could be done by a TreeSet, or an array sort, not that interesting
- Suppose we add GPA to SimpleStudent, a new instance variable, and want students in GPA order
With GPA added to SimpleStudent, it’s not so simple, call the result Student

Student still has equals, hashCode, and compareTo based on id

It has a new instance variable, gpa, and we want students in GPA order

We can set up a JDK PriorityQueue with priority given by the student’s GPA

The JDK PriorityQueue allows us to provide a Comparator (comparing two elements by GPA) to do this

Comparators are discussed in S&W pp. 338-339.
public static void main(String[] args) {
    // new Java 8 features--easy Comparators!
    // Java8 Comparator method "comparing" builds a
    // specific-field comparator, here on GPA--
    Comparator<Student> byGpa = 
        Comparator.comparing(Student::getGpa);
    // or use a lambda function--see code online
    PriorityQueue<Student> pq =
        new PriorityQueue<Student> (byGpa);
    pq.add(new Student("Ann",100, 2.3));
    pq.add(new Student("Ling", 104, 4.0));
    pq.add(new Student("Dave",104, 3.1));
    System.out.println(pq); // note duplicates (equal id)
    // Use DeleteMin to pull them out in GPA order--
    while (!pq.isEmpty()) {
        Student x = pq.poll();
        System.out.println(x);
    }
}
// Java8 Comparator method "comparing" builds a
   specific-field Comparator, here on GPA--
Comparator<Student> byGpa =
    Comparator.comparing(Student::getGpa);
PriorityQueue<Student> pq =
    new PriorityQueue<Student> (byGpa);
pq.add(new Student("Ann",100, 2.3));
pq.add(new Student("Ling", 104, 4.0));
pq.add(new Student("Dave",104, 3.1));
System.out.println(pq); // note duplicates (equal id)
// Use DeleteMin to pull them out in GPA order--
while (!pq.isEmpty()) {
    Student x = pq.poll();
    System.out.println(x);
}
-------
[(Ann 100 2.3), (Ling 104 4.0), (Dave 104 3.1)]
(Ann 100 2.3)          <-- Student.toString() output
(Dave 104 3.1)
(Ling 104 4.0)