CS310 – Advanced Data Structures and Algorithms

JDK Sets

Spring, 2021
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 most important implementations in the JDK are the TreeSet and HashSet.

They check for duplicates by using the compareTo (for TreeSet) or equals (for HashSet) method of the elements.

Additionally, HashSet uses hashCode() of the elements.
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
    // this method is required by "extends Iterable" above
    Iterator<AnyType> iterator();

};

The Set interface is just the Collection interface: thus the empty
curly braces in the following:

public interface Set<AnyType> extends Collection<AnyType> { };
Later we will see that the full JDK Collection/Set API also supports set union, intersection and difference.

Also methods to turn collections/sets into arrays.

JDK Sets also have real equality testing: s1.equals(s2) means the sets have the same elements based on the element equals method.

S&W has a Set API page 489, much like the "simple methods" subset on the last slide.

However, they offer no concrete classes for it, and no equality testing, or union, etc.

You are expected to use an S&W ST class and ignore the values.
import java.util.Set;
import java.util.TreeSet;
public class TestTreeSet {
    public static void main(String[] args) {
        Set<String> ss = new TreeSet<String>();
        ss.add("joe");
        ss.add("bob");
        ss.add("hal");
        for (String s:ss) {
            System.out.println(s);
        }
        // even easier --
        System.out.println(ss);
    }
}
import java.util.Set;
import java.util.TreeSet;
public class TestTreeSet {
    public static void main(String[] args) {
        Set<String> ss = new TreeSet<String>();
        ss.add("joe");
        ss.add("bob");
        ss.add("hal");
        ss.add("hal");
        for (String s:ss) {
            System.out.println(s);
        }
        // even easier--
        System.out.println(ss);
    }
}
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);
    }
}
```
Sets of User Defined Objects

- 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.

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

- TreeSet: equals and compareTo must be consistent, so that if a.equals(b), then a.compareTo(b) == 0.
If we don’t code equals and hashCode methods for our element object, its inherited methods from Object will be called.

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

Object.equals and Object.hashCode are consistent, but don’t use the object content at all, just the reference addresses.

So if we want to avoid duplicate names of Person, or duplicate ids of Order, we need to code equals and hashCode.

Coding equals is remarkably difficult. Start from known-good code (on page 103) and morph it to your needs.

→ Also see article at geeksforgeeks.org
public class Order {
    private int id; // unique identifier, basis of equality
    // other instance variables
    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; // this line is optional
        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();
    }
}

Order Objects for Set<Order>

- Note the use of @Override to make sure the method will actually be used instead of Object.equals, always lurking there.
- Note that equals uses an Object parameter, not an Order parameter: it won’t work with any other type.
- hashCode can often be delegated to the identifier in use: see how id is turned into an Integer and then that Integer’s hashCode is used.
- 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>
More on equals

- We just saw code that required \( x \) and \( y \) to both be Orders in order that \( x = y \).
- But the rules for equals (page 102) do not require identical class.
- In fact, JDK Sets \( a \) and \( b \) are equal if their elements are equal, even if \( a \) is a HashSet and \( b \) is a TreeSet.
- Of course both are Sets, an interface type.
- But an interface can’t implement equals*
- It turns out HashSet and TreeSet have a common superclass AbstractSet that implements equals for both.
- So this shows one case of the power of class inheritance (which we’re not really covering now)
- We could do the same trick for Point2D and PointXY…

*even using Java 8’s “default methods” for interfaces (which we are not covering)
import java.util.Set;
import java.util.HashSet;
public class TestHashSet {
  public static void main(String[] args) {
    Set<Order> s = new HashSet<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?
  }
}
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 parameter has the generic type that gets checked by Java.

Again we use @Override to check our work (checking our code against the Comparable interface)

Again in compareTo, we delegate the actual computation to the Integer id by using its compareTo method.
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 <BankAccount> without changing BankAccount?

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

But all classes have equals and hashCode from Object, and they are consistent.

So we can use HashSet <BankAccount>

We just need to realize it could contain two BankAccount objects with equal id values
What happens if `ba.equals(ba1)` is executed?

Answer: since `ba` ISA `Object` with no `equals` method, `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 instance variables.

If you want object data-testing equality, you need to override `Object.equals` by implementing it in `BankAccount.java`.

And then you need to implement `hashCode` too, to use `HashSet`, since it must be consistent.
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 could drop equals and hashCode from the class implementation and still use HashSet.

In this case, multiple objects could have the same id in a Set.

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