Hashing and Collections

Object Equality

== means that the object reference is the same. The items must refer to the same object to be considered ==.
equals() is a method in the Object class. Each subclass decides how to implement it.
For instance, the String class implements it such that if two different String objects have the same characters, they are equal.
If you don’t override equals(), the equals() in Object is used. This checks ==.

== versus equals() Review

```
String eq1 = new String("abc");
String eq2 = new String("def");
String eq3 = new String("abc");
String eq4 = eq1;
if(eq1 == eq2) {
        System.out.println("eq1 == eq2");
}
if(eq1 == eq3) {
        System.out.println("eq1 == eq3");
}
if(eq1 == eq4) {
        System.out.println("eq1 == eq4");
}
if(eq1.equals(eq2)) {
        System.out.println("eq1.equals(eq2)");
}
if(eq1.equals(eq3)) {
        System.out.println("eq1.equals(eq3)");
}
if(eq1.equals(eq4)) {
        System.out.println("eq1.equals(eq4)");
}
```

Implementing equals()

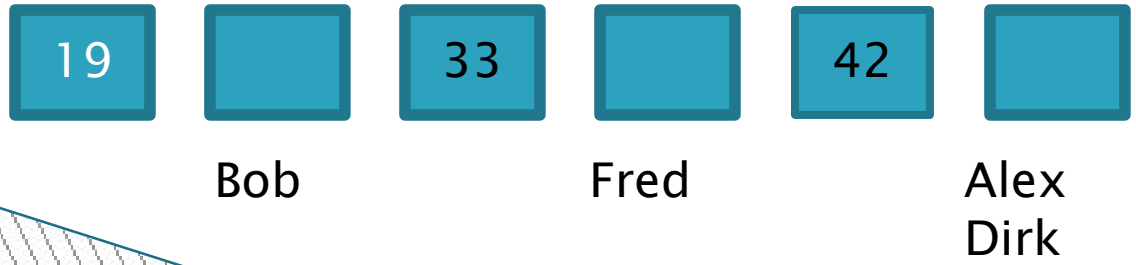
```
public class ImplementingEquals {

public static void main(String[] args) {
    Car hondaRita = new Car(1234);
    Car toyotaJaime = new Car(1234);
    Car fordBen = new Car(1423);

    if(hondaRita.equals(fordBen)) {
    System.out.println("hondaRita and fordBen are the same car.");
    }

    if(hondaRita.equals(toyotaJaime)) {
    System.out.println("hondaRita and toyotaJaime are the same car.");
    }
}
}
```

```
class Car {
// A VIN (Vehicle Identification Number) is unique for each car built
private int vinNumber;

Car(int vinNumber) {
          this.vinNumber = vinNumber;
}

public int getVinNumber() {
          return vinNumber;
}

public boolean equals(Object o) {
     if((o instanceof Car) && ((Car)o).getVinNumber() == this.vinNumber) {
          return true;
     } else {
          return false;
     }
}
}
```

Hash Codes

Collections such as HashMap and HashSet use the hashcode to determine how to store and locate an object.

Find the right bucket using hashCode().
Search the bucket for the right element using equals().



Hashcode Contract

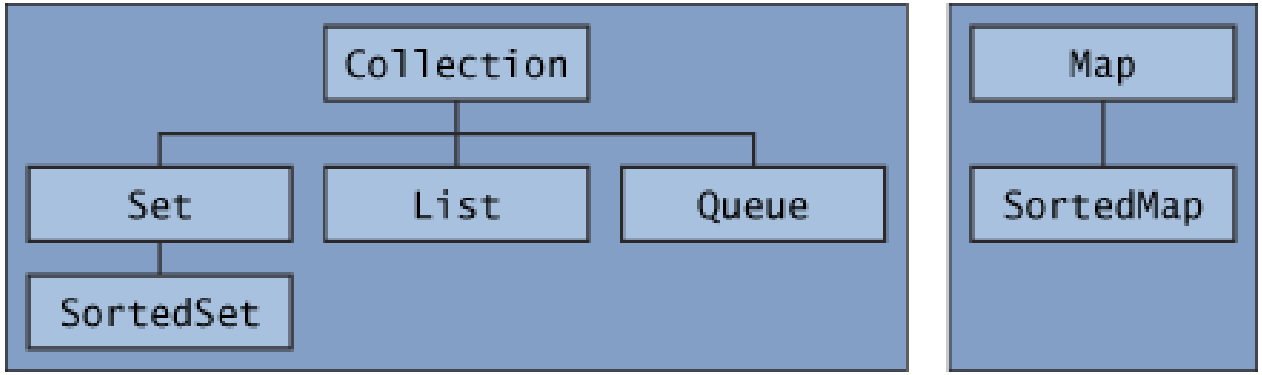
If two objects are equal, their hashcodes must be equal as well.
If called multiple times, the hashcode should return the same integer provided no values used in equals have changed.
Two objects can be unequal using equals() but still have the same hashcode.

What Are Collections For?

Add objects to the collection.
Remove objects from the collection.
Find out if an object is in the collection.
Retrieve an object from the collection.
Iterate through the collection, looking at each element one after another.

The Collections Framework

The Core Collections Interfaces



Collections Basics

collection
Any of the data structures in which objects are stored and iterated over

Collection
java.util.Collection interface from which Set, List, and Queue extend.

Collections
java.util.Collections class that has static utility methods for use with collections.

Working With Collections

Choose Collection wisely
There are 14 main Collection Types.
Think about whether you need ordering, sorting, key lookup.

Code to interface
List myList = new ArrayList();

Autoboxing
myInts.add(new Integer(42))
myInts.add(42)

HashSet

HashSet is an unordered, unsorted implementation of the Set interface
HashSets, like all Sets, cannot contain duplicate entries

```
HashSet<Mountain> hs = new HashSet<Mountain>();
hs.add(new Mountain());
System.out.println(hs);
```

HashSet

Override both equals(Object obj) and hashCode() if you are going to use any collection with the word “Hash” in it such as HashSet

```
public int hashCode(){
       return this.getName().hashCode();

}

public boolean equals(Object o){
       return(this.getName().equals(((Mountain)o).getName()));
}
```

Set Insertion

Inserting into a HashSet
Checks the objects's hashcode to determine where to insert the object

If it finds another object with the same hashCode then it calls one of the object's equals() method to see if they are *actually* equal

Exercise

1. Make a Mountain class that has private instance variables name and height and public get and set methods to access height and name
2. Override Object's equals method in the Mountain class. If a Mountain has a name that is different from another Mountain's name then the equals method returns false. If they are the same, return true.
3. Override Object’s hashcode method by taking the mountain name and applying .hashcode().
4. Add 2 mountains with the same name and 1 mountain with a different name into the HashSet and print the contents.
5. Override the toString() method, using the name of the mountain.

Sorting Collections

```
public static void main(String[] args) {
List<String> stuff = new
ArrayList<String>();
stuff.add("Denver");
stuff.add("Boulder");
stuff.add("Vail");
stuff.add("Aspen");
System.out.println("unsorted:"+stuff);
Collections.sort(stuff);
System.out.println("sorted:"+stuff);
}
```

Comparable Continued

```
@Override
public String toString() {
     return new
Integer(vinNumber).toString();
}

@Override
public int compareTo(Car c) {
     return new
Integer(vinNumber).compareTo(new
Integer(c.getVinNumber()));
}
```

Comparator Interface

Sort a collection any number of ways.
Sort instances of any class, even ones that you can’t modify.

```
class VinSort implements Comparator<Car> {

public int compare(Car one, Car two) {
    return (new
    Integer(one.getVinNumber()).compareTo(new
    Integer(two.getVinNumber())));
}
}

Inside main method--

System.out.println("unsorted cars="+cars);
VinSort vs = new VinSort();
Collections.sort(cars, vs);
System.out.println("sorted cars="+cars);
```

Standard Compare

Both methods return an int:

```
public int compare(Car one, Car two)
public int compareTo(Car c) {
```

If you compare names which are Strings or heights which are Integers, you use the compare and compareTo of these classes (see String and Integer API).

Custom Compare

You can build your own compare and compareTo for Objects that aren't comparing alphanumerically or numerically.

Return:

-1 if item one < item two

0 if item one is equal to item two

1 if item one > item two

Java will take care of sorting the collection with this information

```
//inside Color class
public enum Color {RED, BLUE, GREEN}
// inside main
Car hondaRita = new Car(1234, Color.BLUE);
// constructor example
Car(int vinNumber, Car.Color color) {
        this.vinNumber = vinNumber;
        this.color = color;
}
class ColorSort implements Comparator<Car> {
public int compare(Car one, Car two) {
    if(one.getColor().ordinal() > two.getColor().ordinal()) {
        return 1;
    } else if(one.getColor().ordinal() <
    two.getColor().ordinal()) {
        return -1;
    } else {
        return 0;
    }
}
}
```

int objOne.compareTo(objTwo)

Returns

Negative if objOne < objTwo

Zero if objOne == objTwo

Positive if objOne > objTwo

Modify class

Only one sort

Implemented in API (String, Date, etc)

int compare(objOne, objTwo)

Returns

Negative if objOne < objTwo

Zero if objOne == objTwo

Positive if objOne > objTwo

Build separate class

Many sort classes

Sort instances of third-party classes

TreeSet

TreeSet should be used when you want to keep the set sorted

TreeSet is a *little* slower because its has to maintain the order of the set on every insert

TreeSet uses each Object's compareTo() method to determine where to put each object.

Exercise

Modify your Mountain class so that the main method creates a TreeSet of mountains.

The Mountain class should implement Comparable. Compare the names of the mountain.

Add 5 mountains to a TreeSet and then output the TreeSet

HashMap

Basic Syntax for a HashMap

```
HashMap<String, Mountain> myMap = new
HashMap<String, Mountain>();
myMap.put("m1",new Mountain("Everest",
500));
Mountain m = myMap.get("m1");
System.out.println("m="+m);
```

HashMap

Create a HashMap<String,Integer>
Insert an element with the `.put() method`

```
HashMap<String, Integer> myMap = new HashMap<String,Integer>();
String stringKey = new String("Paul");
Integer numberOne = new Integer(1);
myMap.put(stringKey, numberOne);
```

The Collection Interface

```
public interface Collection<E> extends Iterable<E> {

        // Basic operations int size();
        boolean isEmpty();
        boolean contains(Object element);
        boolean add(E element); //optional
        boolean remove(Object element); //optional

         // Bulk operations
        Iterator<E> iterator();
        boolean containsAll(Collection<?> c);
        boolean addAll(Collection<? extends E> c); //optional
        boolean removeAll(Collection<?> c); //optional
        boolean retainAll(Collection<?> c); //optional
        void clear(); //optional

        // Array operations
        Object[] toArray();
        <T> T[] toArray(T[] a);
}
```