CSSE2002 exam material

John Owen

1 Important bits from the `Object` class

`boolean equals(Object obj)`

Indicates whether some other object is "equal to" this one.

Usually, you override `equals()` iff your class is immutable and it hasn't already been overridden by a superclass.

Rules `equals()` must abide by:

- Reflexivity: `x.equals(x)`.
- Symmetry: `x.equals(y) == y.equals(x)`.
- Transitivity: `x.equals(y) && y.equals(z) == x.equals(z)`.
- `x.equals(null)` is false.

If you override `equals()`, you must also override `hashCode()`.

`int hashCode()`

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by HashMap. The general contract of hashCode is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

`String toString()`

Returns a string representation of the object.

2 Good implementation of `hashCode()`

2.1 General case

First, for every field $\mathtt{f_i}$ ($\mathtt{f_0}$ is the first field and so on), calculate its hash code $\mathtt{h_i}$ using

$$\mathtt{h_i} = \begin{cases} \texttt{(}\mathtt{f_i}\texttt{ ? 1 : 0)} & \text{if } \mathtt{f_i} \text{ is a } \texttt{boolean} \\ \texttt{(int) }\mathtt{f_i} & \text{if } \mathtt{f_i} \text{ is a } \texttt{byte}, \texttt{char}, \texttt{short}, \texttt{long}, \text{ or } \texttt{int} \\ \texttt{Float.floatToIntBits(}\mathtt{f_i}\texttt{)} & \text{if } \mathtt{f_i} \text{ is a } \texttt{float} \text{ or } \texttt{double} \\ \mathtt{f_i}\texttt{ == null ? 0 : }\mathtt{f_i}\texttt{.hashCode()} & \text{if } \mathtt{f_i} \text{ is a } \texttt{Object} \\ \texttt{Arrays.hashCode(}\mathtt{f_i}\texttt{)} & \text{if } \mathtt{f_i} \text{ is an array.} \end{cases}$$

To calculate the hash code, we use $\texttt{hashCode} = \sum_{\mathtt{i}} \mathtt{h_i} 31^{\mathtt{i}}$.

2.2 Example implementation

```
public class Employee {
    int        employeeId;
    String     name;
    Department dept;

    // ...

    public int hashCode() {
        return employeeId + name.hashCode() * 31 + dept.hashCode() * 31 ^ 2;
    }
}
```

3 Important bits from the `List<E>` interface

`boolean add(E e)`
: Appends the specified element to the end of this list (optional operation).

`void add(int index, E element)`
: Inserts the specified element at the specified position in this list (optional operation).

`boolean addAll(Collection<?  extends E> c)`
: Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).

`boolean addAll(int index, Collection<?  extends E> c)`
: Inserts all of the elements in the specified collection into this list at the specified position (optional operation).

`void clear()`
: Removes all of the elements from this list (optional operation).

`boolean contains(Object o)`
: Returns true if this list contains the specified element.

`boolean containsAll(Collection<?> c)`
: Returns true if this list contains all of the elements of the specified collection.

`boolean equals(Object o)`
: Compares the specified object with this list for equality.

`E get(int index)`
: Returns the element at the specified position in this list.

`int hashCode()`
: Returns the hash code value for this list.

`int indexOf(Object o)`
: Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.

`boolean isEmpty()`
: Returns true if this list contains no elements.

`Iterator<E> iterator()`
: Returns an iterator over the elements in this list in proper sequence.

`int lastIndexOf(Object o)`
: Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.

`ListIterator<E> listIterator()`
: Returns a list iterator over the elements in this list (in proper sequence).

`ListIterator<E> listIterator(int index)`
: Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.

`E remove(int index)`
: Removes the element at the specified position in this list (optional operation).

`boolean remove(Object o)`
: Removes the first occurrence of the specified element from this list, if it is present (optional operation).

`boolean removeAll(Collection<?> c)`
: Removes from this list all of its elements that are contained in the specified collection (optional operation).

`void replaceAll(UnaryOperator<E> operator)`
: Replaces each element of this list with the result of applying the operator to that element.

`boolean retainAll(Collection<?> c)`
: Retains only the elements in this list that are contained in the specified collection (optional operation).

`E set(int index, E element)`
: Replaces the element at the specified position in this list with the specified element (optional operation).

`int size()`
: Returns the number of elements in this list.

`void sort(Comparator<?  super E> c)`
: Sorts this list according to the order induced by the specified Comparator.

`List<E> subList(int fromIndex, int toIndex)`
: Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.

`Object[] toArray()`
: Returns an array containing all of the elements in this list in proper sequence (from first to last element).

4 Important bits from the `java.util.Map<K,V>` interface

`void clear()`
: Removes all of the mappings from this map (optional operation). Removes all of the mappings from this map (optional operation).

`boolean containsKey(Object key)`
: Returns true if this map contains a mapping for the specified key. Returns true if this map contains a mapping for the specified key.

`boolean containsValue(Object value)`
: Returns true if this map maps one or more keys to the specified value. Returns true if this map maps one or more keys to the specified value.

`Set<Map.Entry<K,V>> entrySet()`
: Returns a Set view of the mappings contained in this map. Returns a Set view of the mappings contained in this map.

`boolean equals(Object o)`
: Compares the specified object with this map for equality. Compares the specified object with this map for equality.

`V get(Object key)`
: Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key. Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.

`int hashCode()`
Returns the hash code value for this map. Returns the hash code value for this map.

`boolean isEmpty()`
Returns true if this map contains no key-value mappings. Returns true if this map contains no key-value mappings.

`Set<K> keySet()`
Returns a Set view of the keys contained in this map. Returns a Set view of the keys contained in this map.

`V put(K key, V value)`
Associates the specified value with the specified key in this map (optional operation). Associates the specified value with the specified key in this map (optional operation).

`void putAll(Map<? extends K,? extends V> m)`
Copies all of the mappings from the specified map to this map (optional operation). Copies all of the mappings from the specified map to this map (optional operation).

`V remove(Object key)`
Removes the mapping for a key from this map if it is present (optional operation). Removes the mapping for a key from this map if it is present (optional operation).

`boolean remove(Object key, Object value)`
Removes the entry for the specified key only if it is currently mapped to the specified value. Removes the entry for the specified key only if it is currently mapped to the specified value.

`V replace(K key, V value)`
Replaces the entry for the specified key only if it is currently mapped to some value. Replaces the entry for the specified key only if it is currently mapped to some value.

`boolean replace(K key, V oldValue, V newValue)`
Replaces the entry for the specified key only if currently mapped to the specified value. Replaces the entry for the specified key only if currently mapped to the specified value.

`int size()`
Returns the number of key-value mappings in this map. Returns the number of key-value mappings in this map.

`Collection<V> values()`
Returns a Collection view of the values contained in this map. Returns a Collection view of the values contained in this map.

5 Important bits from the `Comparable<T>` interface

`int compareTo(T o)`
Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object. The implementor must ensure `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))` for all `x` and `y`. (This implies that `x.compareTo(y)` must throw an exception iff `y.compareTo(x)` throws an exception.)

The implementor must also ensure that the relation is transitive: `(x.compareTo(y)>0 && y.compareTo(z)>0)` implies `x.compareTo(z)>0`.

Finally, the implementor must ensure that `x.compareTo(y)==0` implies that `sgn(x.compareTo(z)) == sgn(y.compareTo(z))`, for all `z`.

It is strongly recommended, but not strictly required that `(x.compareTo(y)==0) == (x.equals(y))`. Generally speaking, any class that implements the Comparable interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

In the foregoing description, the notation `sgn(expression)` designates the mathematical signum function, which is defined to return one of -1, 0, or 1 according to whether the value of expression is negative, zero or positive.

6 Important unchecked exceptions in `java.lang`

`ArithmeticException`: Arithmetic error, such as divide-by-zero.

`ArrayIndexOutOfBoundsException`: Array index is out-of-bounds.

`ArrayStoreException`: Assignment to an array element of an incompatible type.

`ClassCastException`: Attempt to cast an object of type A to something that isn't a superclass of A.

`IllegalArgumentException`: Illegal argument used to invoke a method.

`IllegalStateException`: Environment or application is in incorrect state.

`IndexOutOfBoundsException`: Some type of index is out-of-bounds.

`NullPointerException`: Invalid use of a null reference.

`NumberFormatException`: Invalid conversion of a string to a numeric format.

`StringIndexOutOfBounds`: Attempt to index outside the bounds of a string.

`UnsupportedOperationException`: An unsupported operation was encountered.

`RuntimeException`: Is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine.