Collections

finalDesk

Methods of class Object

Method	Description
`boolean equals (Object obj)`	Decides whether two objects are meaningfully equivalent.
`void finalize()`	Called by garbage collector when the garbage collector sees that the object cannot be referenced.
`int hashCode()`	Returns a hashcode `int` value for an object, so that the object can be used in Collection classes that use hashing, including Hashtable, HashMap, and HashSet.
`final void notify()`	Wakes up a thread that is waiting for this object's lock.
`final void notifyAll()`	Wakes up *all* threads that are waiting for this object's lock.
`final void wait()`	Causes the current thread to wait until another thread calls `notify()` or `notifyAll()` on this object.
`String toString()`	Returns a "text representation" of the object.

Overriding equals()

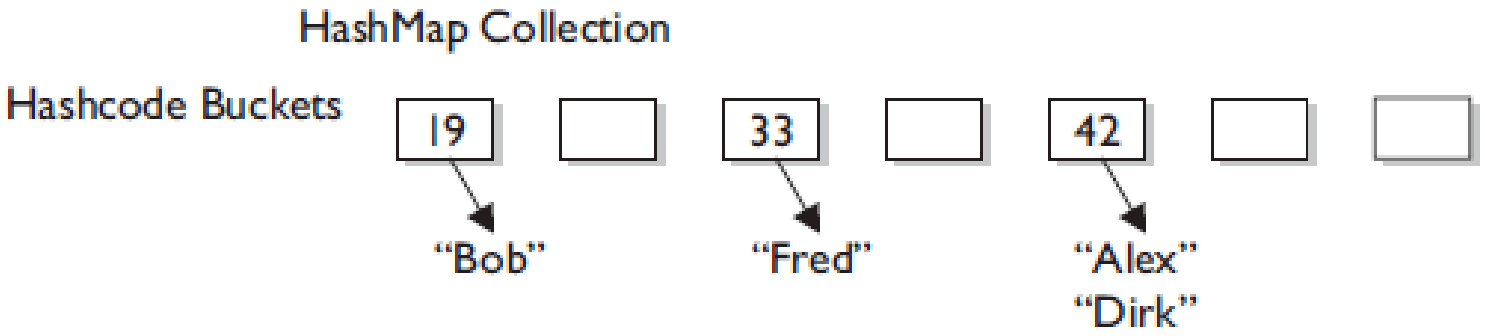
- Two object references using the == operator evaluates to true only when both references refer to the same object(looks bits of variable)
- When you need to know if the objects themselves (not the references) are equal, use the equals() method
- The equals() method in class Object uses only the == operator for comparisons

Implementing an equals() method

```
public class EqualsTest {
  public static void main (String [] args) {
     Moof one = new Moof(8);
     Moof two = new Moof(8);
     if (one.equals(two)) {
        System.out.println("one and two are equal");
     }
  }
}
class Moof {
  private int moofValue;
  Moof(int val) {
     moofValue = val;
  }
  public int getMoofValue() {
     return moofValue;
  }
  public boolean equals(Object o) {
    if ((o instanceof Moof) && (((Moof)o).getMoofValue()
         == this.moofValue)) {
      return true;
    } else {
       return false;
    }
  }
}
```

A Simplified hashCode example

Key	Hashcode Algorithm	Hashcode
Alex	A(1) + L(12) + E(5) + X(24)	= 42
Bob	B(2) + O(15) + B(2)	= 19
Dirk	D(4) + I(9) + R(18) + K(11)	= 42
Fred	F(6) + R(18) + E(5) + D(4)	= 33



Overriding hashCode()

- Hashcodes are typically used to increase the performance of large collections of data
- Although you can think of it as kind of an object ID number, it isn't necessarily unique
- In real-life hashing, it’s not uncommon to have more than one entry in a bucket. Hashing retrieval is a two-step process.
  - Find the right bucket (using hashCode())
  - Search the bucket for the right element (using equals())

Implementing hashCode()

```
class HasHash {
  public int x;
  HasHash(int xVal) { x = xVal; }

  public boolean equals(Object o) {
    HasHash h = (HasHash) o; // Don't try at home without
                             // instanceof test
    if (h.x == this.x) {
      return true;
    } else {
      return false;
    }

   }
   public int hashCode() { return (x * 17); }
 }
```

The hashCode() Contract

Condition	Required	Not Required (But Allowed)
`x.equals(y) == true`	`x.hashCode() == y.hashCode()`	
`x.hashCode() == y.hashCode()`		`x.equals(y) == true`
`x.equals(y) == false`		`No hashCode() requirements`
`x.hashCode() != y.hashCode()`	`x.equals(y) == false`	

What is Collection

- A group of elements
  - normally objects
  - related in some way
- Sometimes known as a container
- Can be manipulated as a single object
  - stored in a variable
  - passed as argument to method
  - returned as a method result
  - grouped into collections

finalDesk

Collection Interface

- `java.util.Collection<E>`
  - highest level interface
  - <E> defines the type of the objects in the collection
- `java.util.List<E>`
  - provides an ordered collection of objects
  - elements can be accessed using integer index
  - user has control over where elements are added

finalDesk

`List<E>` Implementations

- `Vector<E>`
  - Java 1.1 `Vector`
  - for compatibility
- `ArrayList<E>`
  - uses array to implement `List<E>`
  - not thread-safe, otherwise same as `Vector<E>`
- `LinkedList<E>`
  - `List<E>` interface implemented as doubly as a doubly-linked list
  - access to elements is not constant time
  - better performance for frequent add/remove operations in middle of the list

finalDesk

The `Set<E>` Interface

- `java.util.Set<E>`
- Provides a collection with no duplicate elements
  - no ordering or position information for elements
- May or may not allow null elements
  - depends on implementation

finalDesk

`Set<E>` Implementations

- Common features of `Set<E>` classes
  - no duplicate elements
  - elements retrieved by identity, not index
- `HashSet<E>`
  - `Set<E>` interface implemented using a hash table
  - doesn't guarantee to iterate the elements in any specific order
  - constant time access to elements assuming good hash function
- `TreeSet<E>`
  - `Set<E>` interface implemented as a tree

finalDesk

More Interfaces

- The `Queue<E>` Interface
- The `Iterator<E>` Interface
  - Abstract mechanism for traversing a Collection
- `Map<K,V>` Interface

finalDesk

`Map<K,V>` Implementations

- `HashTable<K,V>`
  - updated class from earlier Java versions
  - does not allow null keys and values
  - thread safe
- `HashMap<K,V>`
  - similar to `HashTable<K,V>` but allows null keys and values
  - not thread safe
- `LinkedHashMap<K,V>`
  - extends `HashMap<K,V>`
  - maintains double linked list through all elements, used for iteration
  - list normally maintains insertion order

finalDesk

Ordering Within Collections

- Classes may have Ordering criteria
- `Comparable Interface`
- `Comparator Interface`
- Sorted Collections and Maps
  - `SortedSet` Interface
  - `SortedMap` Interface

finalDesk

Exam Watch

exam Watch

We've talked a lot about sorting by natural order and using Comparators to sort. The last rule you'll need to burn in is that, whenever you want to sort an array or a collection, the elements inside must all be mutually comparable. In other words, if you have an `Object[]` and you put `Cat` and `Dog` objects into it, you won't be able to sort it. In general, objects of different types should be considered NOT mutually comparable, unless specifically stated otherwise.

Generics

- What is Generics
- Where it is used?
- Why it is important?

finalDesk

Compile-time and Runtime Types

- Some unexpected results when looking at generic types at runtime
  - based on requirement for interoperability with legacy code
  - e.g. non generic collection classes from earlier Java versions

finalDesk

Raw Types and Compatitbility

- Collection types all modified to be generic types
- Pre-Java 5 code will still work
  - generic types become raw types
- Post-Java 5 compiler will issue warnings

finalDesk

Parameterised Type Equivalence

- Parameterised types form a hierarchy
  - based on the type not the parameter
- Otherwise type safety would be compromised
  - `List<Integer>` is a `Collection<Integer>`
  - `List<Integer>` is not a `List<Object>`
- Parameterised types are equivalent to raw types

finalDesk

Question

Given:

```
public static void main(String[] args) {
  // INSERT DECLARATION HERE
  for (int i = 0; i <= 10; i++) {
    List<Integer> row = new ArrayList<Integer>();
    for (int j = 0; j <= 10; j++)
      row.add(i * j);
    table.add(row);
  }
  for (List<Integer> row : table)
    System.out.println(row);
  }
```

Which statements could be inserted at `// INSERT DECLARATION HERE` to allow this code to compile and run? (Choose all that apply.)

A. `List<List<Integer>> table = new List<List<Integer>>();`

B. `List<List<Integer>> table = new ArrayList<List<Integer>>();`

C. `List<List<Integer>> table = new ArrayList<ArrayList<Integer>>();`

D. `List<List, Integer> table = new List<List, Integer>();`

E. `List<List, Integer> table = new ArrayList<List, Integer>();`

F. `List<List, Integer> table = new ArrayList<ArrayList, Integer>();`

G. None of the above

Answer

Answer:

☑ B is correct.

☒ A is incorrect because List is an interface, so you can't say `new List()` regardless of any generic types. D, E, and F are incorrect because List only takes one type parameter (a Map would take two, not a List). C is tempting, but incorrect. The type argument `<List<Integer>>` must be the same for both sides of the assignment, even though the constructor `new ArrayList()` on the right side is a subtype of the declared type List on the left. (Objective 6.4)

Question

Which statements are true about comparing two instances of the same class, given that the `equals()` and `hashCode()` methods have been properly overridden? (Choose all that apply.)

A. If the `equals()` method returns true, the `hashCode()` comparison `==` might return false

B. If the `equals()` method returns false, the `hashCode()` comparison `==` might return true

C. If the `hashCode()` comparison `==` returns true, the `equals()` method must return true

D. If the `hashCode()` comparison `==` returns true, the `equals()` method might return true

E. If the `hashCode()` comparison `!=` returns true, the `equals()` method might return true

Answer

Answer:

☑ B and D. B is true because often two dissimilar objects can return the same hashcode value. D is true because if the `hashCode()` comparison returns ==, the two objects might or might not be equal.

☒ A, C, and E are incorrect. C is incorrect because the `hashCode()` method is very flexible in its return values, and often two dissimilar objects can return the same hash code value. A and E are a negation of the `hashCode()` and `equals()` contract. (Objective 6.2)

Contact Info

- trainers@finaldesk.com
- rishabh@finaldesk.com
- nilesh@finaldesk.com
- jignesh@finaldesk.com
- yash@finaldesk.com
- anand@finaldesk.com