**Collections and Generics:**

**ToString():**

1) When we pass the object reference to the system.out.println the object ToString method is called. If we don’t override the ToString method it gives the class name followed by @ symbol followed by unsigned hexadecimal representation of objects hashcode.

Example:

public class HardToRead {

public static void main (String [] args) {

HardToRead h = new HardToRead();

System.out.println(h);

}

}

Out put:

% java HardToRead

HardToRead@a47e0

**Equals() and ==:**

When you want need to know if the object references are identical use ==

When you want to know if the objects themselves are equal use equals method.

Before overriding the equals() method u must decide if it makes sense to consider two different instances equal.

For some classes you might decide that the two objects can never be equal.

For example, Imagine a car class, You certainly don’t want your car suddenly treated as the very same car as someone with a car that as identical attributes.

So no two cars should ever be considered exactly equal. If two references refer to one car, then you know that both are talking about one car, not two cars that have

same attributes. So in the case of a Car you might not ever need, or want, to override the equals() method.

What it means if you don’t override the equals():

1) If you don’t override a class’s equals method you won’t be able to use those objects as key in a hash table.

2) equals method in Object class uses == operators for comparisons, so unless you override equals(), two objects are considered equal only if the two references refer to the same object.

3) Let's look at what it means to not be able to use an object as a hashtable key.Imagine you have a car, a very specific car (say, John's red Subaru Outback as opposed to Mary's purple Mini) that you want to put in a HashMap (a type of hashtable we'll look at later in this chapter), so that you can search on a particular car and retrieve the corresponding Person object that represents the owner. So you add the car instance as the key to the HashMap (along with a corresponding Person object as the value). But now what happens when you want to do a search? You want to say to the HashMap collection, "Here's the car, now give me the Person object that goes with this car." But now you're in trouble unless you still have a reference to the exact object you used as the key when you added it to the Collection. *In other* *words, you can't make an identical Car object and use it for the search.*

4) Logically we have to make two things in order to make a valid equality comparison.

a) Be sure that the object being tested is of the correct type, so you need to do an instance of test on it. Just to make sure that you could cast the object argument to correct type so that you can access its methods or variables in order to actually do the comparison. If the object doesn’t pass the instance of test you’ll get a runtime ClassCastException.

b) Only compare the attributes we care about.

**The equals() Contract:**

Pulled straight from the Java docs, the equals() contract says

■ It is **reflexive**. For any reference value x, x.equals(x) should return true.

■ It is **symmetric**. For any reference values x and y, x.equals(y) should

return true if and only if y.equals(x) returns true.

■ It is **transitive**. For any reference values x, y, and z, if x.equals(y) returns

true and y.equals(z) returns true, then x.equals(z) must return true.

■ It is **consistent**. For any reference values x and y, multiple invocations of

x.equals(y) consistently return true or consistently return false, provided

no information used in equals comparisons on the object is modified.

■ For any non-null reference value x, x.equals(null) should return false.

**Overriding hashcode():**

Some of the collection classes uses the hashcode to increase the performance of large collections.

For Example refer the java book page number 550.

* The goal is to get a wide and random distribution of objects across the bucket and the contract requires only that two equal objects have equal hashcodes.
* In order for an object to be located, the search object and the object in the collection must have both identical hashcode values and return true for the equals method.

Hashcode contract:

■ 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, thencalling 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 hashtables

**Important Note:**

Transient variables can really mess with your equals() andhashCode() implementations. Keep variables non-transient or, if they mustbe marked transient, don't use them to determine hashcodes or equality.

**Collections Framework:**

Interfaces need to know

Collection

Set SortedSet NavigableSet

List

Map SortedMap NavigableMap

Queue

Concrete implementation classes needs to be know:

Map Set Lists Queues Utilities

HashMap HashSet ArrayList PriorityQueue Collections

Hashtable LinkedHashSet Vector Arrays

TreeMap TreeSet LinkedList

LinkedHashMap

Three overloaded use of word collection:

collection (lowercase *c*), which represents any of the data structures in which objects are stored and iterated over.

■ Collection (capital *C*), which is actually the java.util.Collection interface from which Set, List, and Queue extend. (That's right, extend, not implement. There are no direct implementations of Collection.)

■ Collections (capital *C* and ends with *s*) is the java.util.Collections class that holds a pile of static utility methods for use with collections.

Collections come in four basic flavors:

■ **Lists** *Lists* of things (classes that implement List).

■ **Sets** *Unique* things (classes that implement Set).

■ **Maps** Things with a *unique* ID (classes that implement Map).

■ **Queues** Things arranged by the order in which they are to be processed.

There are sub flavors within those four flavors

Sorted, unsorted, ordered, unordered.

The Methods in Collections Class

Collections.copy(v,arrayList); to copy from one collection to other collection.

# List list = Collections.nCopies(5,"A"); Create List containing n Copies of Specified Object Example

# Collections.min(arrayList); TO find min value

# Collections.max(hashSet); to find max value

# Collections.enumeration(arrayList); To get enumeration over collections .

# Tommorw : Subset of collection