JAVA™ 2

SUN™ CERTIFIED PROGRAMMER & DEVELOPER

7

Objects and Collections

CERTIFICATION OBJECTIVES

- Overriding hashCode() and equals()
- Collections
- Garbage Collection

✓ Two-Minute Drill

Q&A Self Test

CERTIFICATION OBJECTIVE

Overriding hashCode() and equals() (Exam Objective 9.2)

Distinguish between correct and incorrect implementations of hashcode methods.

You're an object. Get used to it. You have state, you have behavior, you have a job. (Or at least your chances of getting one will go up after passing the exam.) If you exclude primitives, everything in Java is an object. Not just object, but Object with a capital 'O'. Every exception, every event, *every array* extends from java.lang.Object. We've already talked about it in Chapter 6 when we looked at overriding `equals()`, but there's more to the story, and that *more* is what we'll look at now.

For the exam, you don't need to know every method in Object, but you *will* need to know about the methods listed in Table 7-1.

Chapter 9 covers `wait()`, `notify()`, and `notifyAll()`. The `finalize()` method is covered later in this chapter. So in this section we'll look at just the `hashCode()` and `equals()` methods. Oh, that leaves out `toString()`, doesn't it. OK, we'll cover that right now because it takes two seconds.

TABLE 7-1 Methods of Class Object Covered on the Exam

Method	Description
`boolean equals(Object obj)`	Decides whether two objects are meaningfully equivalent.
`void finalize()`	Called by the 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.

The toString() Method Override `toString()` when you want a mere mortal to be able to read something meaningful about the objects of your class. Code can call `toString()` on your object when it wants to read useful details about your object. For example, when you pass an object reference to the `System.out.println()` method, the object's `toString()` method is called, and the return of `toString()` is what you see displayed as follows:

```
public class HardToRead {
   public static void main (String [] args) {
     HardToRead h = new HardToRead();
     System.out.println(h);
   }
}
```

Running the HardToRead class gives us the lovely and meaningful,

```
% java HardToRead
HardToRead@a47e0
```

The preceding output is what you get when you *don't* override the `toString()` method of class Object. It gives you the class name (at least *that's* meaningful) followed by the @ symbol, followed by the unsigned hexadecimal representation of the object's hashcode.

Seeing this perhaps motivates you to override the `toString()` method in your classes, for example,

```
public class BobTest {
   public static void main (String[] args) {
     Bob f = new Bob("GoBobGo", 19);
     System.out.println(f);
   }
}

class Bob {
   int shoeSize;
   String nickName;
   Bob(String nickName, int shoeSize) {
     this.shoeSize = shoeSize;
     this.nickName = nickName;
   }
   public String toString() {
      return ("I am a Bob, but you can call me " + nickName +
              ". My shoe size is " + shoeSize);
   }
}
```

This ought to be a bit more readable:

```
% java BobTest
I am a Bob, but you can call me GoBobGo. My shoe size is 19
```

Some people affectionately refer to `toString()` as "the spill-your-guts method," because the most common implementations of `toString()` simply spit out the object's state (in other words, the current values of the important instance variables).

So that's it for `toString()`. Now we'll tackle `equals()` and `hashCode()`.

Overriding equals()

You learned about the `equals()` method in Chapter 6, where we looked at the wrapper classes. We discussed how comparing two object references using the == operator evaluates true only when both references refer to the same object (because == simply looks at the bits in the variable, and they're either identical or they're not). You saw that the String class and the wrapper classes have overridden the `equals()` method (inherited from class Object), so that you could compare two different objects (of the same type) to see if their *contents* are meaningfully equivalent. If two different Integer instances both hold the *int* value 5, as far as you're concerned they *are* equal. The fact that the value 5 lives in two separate objects doesn't matter.

When you really need to know if two *references* are identical, use ==. But when you need to know if the *objects themselves* (not the references) are equal, use the `equals()` method. For each class you write, you must decide if it makes sense to consider two different instances equal. For some classes, you might decide that two objects can *never* be equal. For example, imagine a class Car that has instance variables for things like make, model, year, configuration—you certainly don't want your car suddenly to be treated as the very *same* car as someone with a car that has *identical attributes*. Your car is your car and you don't want your neighbor Billy driving off in it just because, "hey, it's really the same car; the `equals()` method said so." 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 the same attributes. So in the case of a Car you might not ever need, or want, to override the `equals()` method. Of course, you know that can't be the end of the story.

What It Means if You *Don't* Override equals()

There's a potential limitation lurking here: if you don't override the `equals()` method, *you won't be able to use the object as a key in a hashtable.* The `equals()` method in Object uses only the == operator for comparisons, so unless you override `equals()`, two objects are considered equal *only if the two references refer to the same object.*

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 Moe and 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.

The bottom line is this: if you want objects of your class to be used as keys for a hashtable (or as elements in any data structure that uses equivalency for searching for—and/or retrieving—an object), then *you must override `equals()` so that two different instances can be considered the same.* So how would we fix the car? You might override the `equals()` method so that it compares the unique VIN (Vehicle Identification Number) as the basis of comparison. That way, you can use one instance when you add it to a Collection, and essentially *re-create* an identical instance when you want to do a search based on that object as the key. Of course, overriding the `equals()` method for Car also allows the potential that more than one object representing a single unique car can exist, which might not be safe in your design. Fortunately, the String and wrapper classes work well as keys in hashtables—they override the `equals()` method. So rather than using the actual *car instance* as the key into the car/owner pair, you could simply use a String that represents the unique identifier for the car. That way, you'll never have more than one instance representing a specific car, but you can still use the car—or rather*, one of the car's attributes*—as the search key.

Implementing an equals() Method

So let's say you decide to override `equals()` in your class. It might look something like this:

```
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;
     }
  }
}
```

Let's look at this code in detail. In the main method of EqualsTest, we create two Moof instances, passing the same value (8) to the Moof constructor. Now look at the Moof class and let's see what it does with that constructor argument—it assigns the value to the *moofValue* instance variable. Now imagine that you've decided two Moof objects are the same if their *moofValue* is identical. So you override the `equals()` method and compare the two *moofValues*. It *is* that simple. But let's break down what's happening in the `equals()` method:

```
1. public boolean equals(Object o) {
2.   if ((o instanceof Moof) && (((Moof)o).getMoofValue()
           == this.moofValue)) {
3.      return true;
```

```
4.      } else {
5.        return false;
6.      }
7.   }
```

First of all, you *must* observe all the rules of overriding, and in line 1 we are indeed declaring a valid override of the `equals()` method we inherited from Object.

Line 2 is where all the action is. Logically, we have to do *two* things in order to make a valid equality comparison:

1. *Be sure that the object being tested is of the correct type!* It comes in polymorphically as type Object, so you need to do an `instanceof` test on it. Having two objects of different class types be considered equal is usually *not* a good idea, but that's a design issue we won't go into here. Besides, you'd *still* have to do the `instanceof` test just to be sure that you could cast the object argument to the correct type *so that you can access its methods* or variables in order to actually *do* the comparison. Remember, if the object doesn't pass the `instanceof` test, then you'll get a runtime `ClassCastException` if you try to do, for example, this:

   ```
   public boolean equals(Object o) {
      if (((Moof)o).getMoofValue() == this.moofValue){
         // the preceding line compiles, but it's BAD!
         return true;
        } else {
          return false;
        }
     }
   ```

   The `(Moof)o` cast will fail if *o* doesn't refer to something that IS-A Moof.

2. *Compare the attributes we care about* (in this case, just *moofValue*). Only the developers can decide what makes two instances equal. (For performance you're going to want to check the fewest number of attributes.)

By the way, in case you were a little surprised by the whole `((Moof)o).getMoofValue()` syntax, we're simply casting the object reference, *o*, just-in-time as we try to call a method that's in the Moof class but not in Object. Remember *without* the cast, you can't compile because the compiler would see the object referenced by *o* as simply, well, an Object. And since the Object class doesn't have a `moofvalue()` method, the compiler would squawk (technical

term). But then as we said earlier, even *with* the cast the code fails at runtime if the object referenced by *o* isn't something that's castable to a Moof. So don't ever forget to use the `instanceof` test first. Here's another reason to appreciate the short circuit && operator—if the `instanceof` test fails, we'll never *get* to the code that does the cast, so we're always safe at runtime with the following:

```
if ((o instanceof Moof) && (((Moof)o).getMoofValue()
      == this.moofValue)) {
    return true;
} else {
    return false;
}
```

exam Watch

Remember that the `equals()`, `hashCode()`, and `toString()` methods are all public. The following would not be a valid override of the `equals()` method, although it might appear to be if you don't look closely enough during the exam:

```
class Foo {
   boolean equals(Object o) { }
  }
}
```

And watch out for the argument types as well. The following method is an overload, but not an override of the equals() method:

```
class Boo {
   public boolean equals(Boo b) { }
}
```

Be sure you're very comfortable with the rules of overriding so that you can identify whether a method from Object is being overridden, overloaded, or illegally redeclared in a class. The `equals()` method in class Boo changes the argument from Object to Boo, so it becomes an overloaded method and won't be called unless it's from your own code that knows about this new, different method that happens to also be named equals.

So that takes care of `equals()`.

Whoa… not so fast. If you look at the Object class in the Java API documentation, you'll find what we call a *contract* specified in the `equals()` method. A Java *contract* is a set of rules that *should* be followed, or rather *must be followed if you want to provide a "correct" implementation as others will expect it to be.* Or to put it another way, if you

don't follow the contract, you may still compile and run, but your code (or someone else's) may break at runtime in some unexpected way.

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)` should 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 nonnull reference value `x`, `x.equals(null)` should return `false`.

And you're so not off the hook yet. We haven't looked at the `hashCode()` method, but `equals()` and `hashCode()` are bound together by a joint contract that specifies *if two objects are considered equal using the `equals()` method, then they must have identical hashcode values.* So to be truly safe, your rule of thumb should be *if you override `equals()`, override `hashCode()` as well.* So let's switch over to `hashCode()` and see how that method ties in to `equals()`.

Overriding hashCode()

The hashcode value of an object is used by some collection classes (we'll look at the collections later in this chapter). Although you can think of it as kind of an object ID number, it isn't necessarily unique. Collections such as HashMap and HashSet use the hashcode value of an object to determine where the object should be *stored* in the collection, and the hashcode is used again to help *locate* the object in the collection. For the exam you do *not* need to understand the deep details of how the collection classes that use hashing are implemented, but you *do* need to know which collections use them (but, um, they all have *hash* in the name so you should be good

there). You must also be able to recognize an *appropriate* or *correct* implementation of `hashCode()`. This does not mean *legal* and does not even mean *efficient.* It's perfectly legal to have a terribly inefficient hashcode method in your class, as long as it doesn't violate the contract specified in the Object class documentation (we'll look at that contract in a moment). So for the exam, if you're asked to pick out an appropriate or correct use of hashcode, don't mistake appropriate for *legal* or *efficient.*

Understanding Hashcodes

In order to understand what's appropriate and correct, we have to look at how some of the collections use hashcodes.

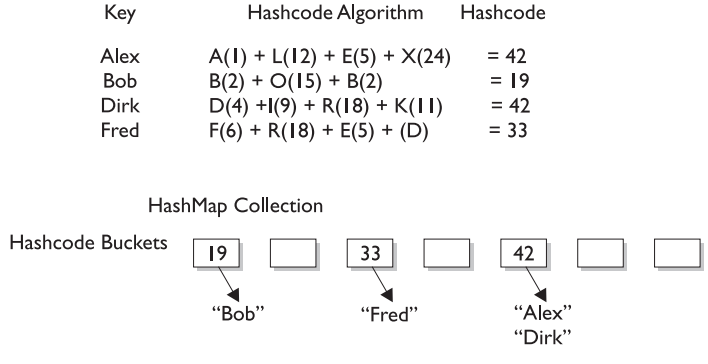
Imagine a set of buckets lined up on the floor. Someone hands you a piece of paper with a name on it. You take the name and calculate an integer code from it by using A is 1, B is 2, etc., and adding the numeric values of all the letters in the name together. *A specific name will always result in the same code;* for example, see Figure 7-1.

We don't introduce anything random, we simply have an algorithm that will always run the same way given a specific input, so the output will always be identical for any two identical inputs. So far so good? Now the way you *use* that code (and we'll call it a *hashcode* now) is to determine which bucket to place the piece of paper into (imagine that each bucket represents a different code number you might get). Now imagine that someone comes up and shows you a name and says, "Please retrieve the piece of paper that matches this name." So you look at the name they show you, and run the same hashcode-generating algorithm. The hashcode tells you in which bucket you should look to find the name.

You might have noticed a little flaw in our system, though. *Two different names might result in the same value.* For example, the names *Amy* and *May* have the same

FIGURE 7-1

A simplified hashcode example



letters, so the hashcode will be identical for both names. That's acceptable, but it does mean that when someone asks you (the bucket-clerk) for the *Amy* piece of paper, you'll still have to search through the target bucket reading each *name* until we find *Amy* rather than *May*. The code tells you only which *bucket* to go into, but not how to locate the name once we're *in* that bucket.

on the job

In real-life hashing, it's not uncommon to have more than one entry in a bucket. Good hashing retrieval is typically a two-step process:

1. ***Find the right bucket.***
2. ***Search the bucket for the right element.***

So for efficiency, your goal is to have the papers distributed as evenly as possible across all buckets. Ideally, you might have just one name per bucket so that when someone asked for a paper you could simply calculate the hashcode and just grab the *one* paper from the correct bucket (without having to go flipping through different papers in that bucket until you locate the exact one you're looking for). The least efficient (but still functional) hashcode generator would return the *same* hashcode (say, 42) *regardless* of the name, so that *all* the papers landed in the same bucket while the others stood empty. The bucket-clerk would have to keep going to that one bucket and flipping painfully through each one of the names in the bucket until the right one was found. And if *that's* how it works, they might as well not use the hashcodes at all but just go to the one big bucket and start from one end and look through each paper until they find the one they want.

This distributed-across-the-buckets example is similar to the way hashcodes are used in collections. When you put an object in a collection that uses hashcodes, the collection uses the hashcode of the object to decide in which bucket/slot the object should land. Then when you want to *fetch* that object (or, for a hashtable, retrieve the associated value for that object), you have to give the collection a reference to an object which *the collection compares to the objects it holds in the collection*. As long as the object (stored in the collection, like a paper in the bucket) you're trying to search for has the *same hashcode* as the object you're using for the search (the name you *show* to the person working the buckets), then the object will be found. But…and this is a Big One, imagine what would happen if, going back to our name example, you showed the bucket-worker a name and they calculated the code based on only *half* the letters in the name instead of *all* of them. They'd never find the name in the bucket because they wouldn't be looking in the correct bucket!

Now can you see why if two objects are considered equal, their hashcodes must also be equal? Otherwise, you'd never be able to find the object since the default hashcode method in class Object virtually always comes up with a unique number for each object, *even if the equals method is overridden in such a way that two or more objects are considered equal.* It doesn't matter how equal the objects are if their hashcodes don't reflect that. So one more time: *If two objects are equal, their* ***hashcodes*** *must be equal as well.*

Implementing hashCode()

What the heck does a *real* hashcode algorithm look like? People get their PhDs on hashing algorithms, so from a computer science viewpoint, it's beyond the scope of the exam. The part we care about here is the issue of *whether you follow the contract.* And to follow the contract, think about what you do in the `equals()` method. *You compare attributes.* Because that comparison almost always involves instance variable values (remember when we looked at two Moof objects and considered them equal if their *int moofValues* were the same?). Your `hashCode()` implementation should use the same instance variables. Here's an example:

```
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);
  }
}
```

Because the `equals()` method considers two objects equal if they have the same *x* value, we have to be sure that objects with the same *x* value will return identical hashcodes.

exam Watch

A `hashCode()` that returns the same value for all instances whether they're equal or not is still a legal—even appropriate—`hashCode()` method! For example,

```
public int hashCode() {
    return 1492;
  }
```

would not violate the contract. Two objects with an x value of 8 will have the same hashcode. But then again, so will two unequal objects, one with an x value of 12 and the other a value of -920. This `hashCode()` method is horribly inefficient, remember, because it makes all objects land in the same bucket, but even so, the object can still be found as the collection cranks through the one and only bucket—using `equals()`—trying desperately to finally, painstakingly, locate the correct object. In other words, the hashcode was really no help at all in speeding up the search, even though search speed is hashcode's intended purpose! Nonetheless, this one-hash-fits-all method would be considered appropriate and even correct because it doesn't violate the contract. Once more, correct does not necessarily mean good.

Typically, you'll see `hashCode()` methods that do some combination of ^-ing (XOR-ing) the instance variables, along with perhaps multiplying them by a prime number. In any case, while the goal is to get a wide and random distribution of objects across buckets, the contract (and whether or not an object can be found) requires only that two equal objects have equal hashcodes. The exam does *not* expect you to rate the efficiency of a `hashCode()` method, but you must be able to recognize which ones will and will not work (work meaning "will cause the object to be found in the collection").

Now that we know that two equal objects must have identical hashcodes, is the reverse true? Do two objects with identical hashcodes have to be considered equal? Think about it—you might have lots of objects land in the same bucket because their hashcodes are identical, but unless they *also pass the `equals()` test,* they won't come up as a match in a search through the collection. This is exactly what you'd get with our very inefficient everybody-gets-the-same-hashcode method. It's legal and correct, just slooooow.

So 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. So there's just no way out of overriding both methods *to be absolutely certain that your objects can be used in Collections that use hashing.*

The hashCode() Contract

Now coming to you straight from the fabulous Java API documentation for class Object, may we present (drum roll) the `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, 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 hashtables.

And what this means to you is…

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`	

So let's look at what *else* might cause a `hashCode()` method to fail. What happens if you include a `transient` variable in your `hashCode()` method? While that's legal (compiler won't complain), under some circumstances an object you put in a collection won't be found. The exam doesn't cover object serialization, so we won't go into any details here. Just keep in mind that serialization saves an object so that it can be reanimated later by deserializing it back to full objectness.

But danger Will Robinson—remember that transient variables are not saved when an object is serialized. A bad scenario might look like this:

```
class SaveMe implements Serializable{
  transient int x;
  int y;
   SaveMe(int xVal, int yVal) {
      x = xVal;
      y = yVal;
   }
  public int hashCode() {
     return (x ^ y); //Legal, but not correct to
                     // use a transient variable
  }
  public boolean equals(Object o) {
     SaveMe test = (SaveMe)o;
     if (test.y == y && test.x == x) { // Legal, not correct
       return true;
     } else {
        return false;
     }
  }
}
```

Here's what could happen using code like the preceding example:

- Give an object some state (assign values to its instance variables).
- Put the object in a HashMap, using the object as a key.
- Save the object to a file using object serialization without altering any of its state.
- Retrieve the object from the file through deserialization.
- Use the deserialized (brought back to life on the heap) object to get the object out of the HashMap.

Oops. The object in the collection and the *supposedly* same object brought back to life are no longer identical. The object's transient variable will come back with a default value rather than the value the variable had at the time it was saved (or put into the HashMap). So using the preceding SaveMe code, if the value of *x* is 9 when the instance is put in the HashMap, then since *x* is used in the calculation of the hashcode, when the value of *x* changes the hashcode changes too. And when that same instance of SaveMe is brought back from deserialization, `x == 0`, *regardless*

of the value of x at the time the object was serialized. So the new hashcode calculation will give a *different* hashcode, and the `equals()` method fails as well since *x* is used as one of the indicators of object equality.

Bottom line: transient variables can really mess with your equals and hashcode implementations. Either keep the variable nontransient or, if it *must* be marked transient, then don't use it in determining an object's hashcode or equality.

CERTIFICATION OBJECTIVE

Collections (Exam Objective 9.1)

Make appropriate selection of collection classes/interfaces to suit specific behavior requirements.

Can you imagine trying to write object-oriented applications without using data structures like hashtables or linked lists? What would you do when you needed to maintain a sorted list of, say, all the members in your *Simpsons* fan club? Obviously you can do it yourself; Amazon.com must have thousands of algorithm books you can buy. But with the kind of schedules programmers are under today ("Here's a spec. Can you have it all built by tomorrow morning?"), it's almost too painful to consider.

The Collections Framework in Java, which took shape with the release of JDk1.2 (the first *Java 2* version) and expanded in 1.4, gives you lists, sets, and maps to satisfy most of your coding needs. They've been tried, tested, and tweaked. Pick the best one for your job and you'll get—at the least—reasonably good performance. And when you need something a little more custom, the Collections Framework in the java.util package is loaded with interfaces and utilities.

So What Do You Do with a Collection?

There are a few basic operations you'll normally use with collections:

- *Add objects* to the collection.
- *Remove objects* from the collection.
- *Find out if an object (or group of objects)* is in the collection.

- *Retrieve an object* from the collection (without removing it).
- *Iterate* through the collection, looking at each element (object) one after another.

Key Interfaces and Classes of the Collections Framework

For the exam, you won't need to know much detail about the collections, but you *will* need to know the purpose of the each of the key interfaces, and you'll need to know *which* collection to choose based on a stated requirement. The collections API begins with a group of interfaces, but also gives you a truckload of concrete classes. The core interfaces you need to know for the exam (and life in general) are the following six:

Collection	Set	Sorted Set
List	Map	Sorted Map

Figure 7-2 shows the interface and class hierarchy for collections.

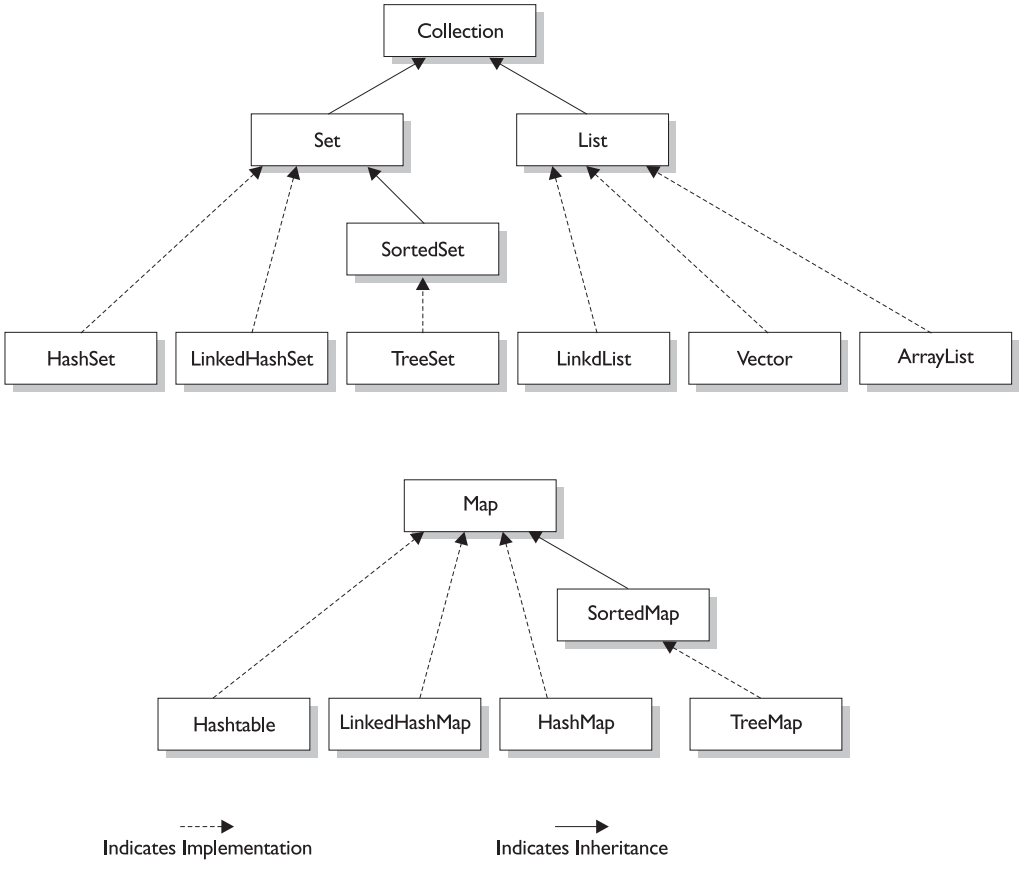
The core concrete implementation classes you need to know for the exam are the following ten (there are others, but the exam doesn't specifically cover them):

Map Implementations	Set Implementations	List Implementations
HashMap	HashSet	ArrayList
Hashtable	LinkedHashSet	Vector
TreeMap	TreeSet	LinkedList
LinkedHashMap		

Not all collections in the Collections Framework actually implement the Collection interface. In other words, *not all collections pass the IS-A test for Collection*. Specifically, none of the Map-related classes and interfaces extend from Collection. So while SortedMap, Hashtable, HashMap, TreeMap, and LinkedHashMap are all thought of as *collections*, none are actually extended from Collection-with-a-capital-C. To make things a little more confusing, there are really *three* overloaded uses of the word "collection":

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

FIGURE 7-2 The collections class and interface hierarchy



- Collection (capital 'C'), which is actually the java.util.Collection interface from which Set and List extend. (That's right, *extend,* not implement. There are *no* direct implementations of Collection.)
- Collections (capital 'C' and ends with 's'), which is actually the java.util.Collections *class* that holds a pile of static utility methods for use with collections.

exam Watch

You can so easily mistake "Collections" for "Collection"—be careful. Keep in mind that Collections is a class, with static utility methods, while Collection is an interface with declarations of the methods common to most collections including `add`, `remove`, `contains`, `size`, and `iterator`.

Collections come in three 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)

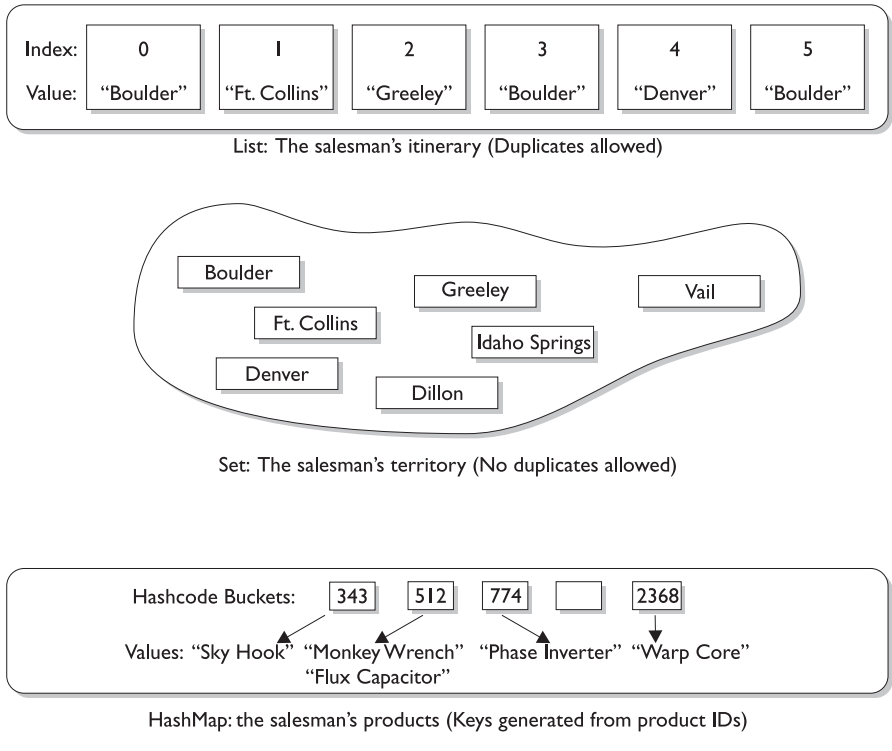
Figure 7-3 illustrates the structure of a List, a Set, and a Map.
But there are subflavors within those three types:

Sorted	Unsorted	Ordered	Unordered

An implementation class can be unsorted and unordered, ordered but unsorted, or both ordered *and* sorted. But an implementation can never be sorted but unordered, because sorting is a specific type of ordering, as you'll see in a moment. For example,

FIGURE 7-3

Lists, Sets, and Maps



a HashSet is an unordered, unsorted set, while a LinkedHashSet is an ordered (but not sorted) set that maintains the order in which objects were inserted.

Maybe we need to be explicit about the difference between sorted and ordered, but first we have to discuss the idea of *iteration*. When you think of iteration, you may think of iterating over an array using, say, a *for* loop to access each element in the array in order ([0], [1], [2], etc.). Iterating through a collection usually means walking through the elements one after another starting from the first element. Sometimes, though, even the concept of *first* is a little strange—in a Hashtable there really *isn't* a notion of first, second, third, and so on. In a Hashtable, the elements are placed in a (as far as you're concerned) chaotic order based on the hashcode of the key. But *something* has to go first when you iterate; thus, when you iterate over a Hashtable there will indeed be an order. But as far as you can tell, it's completely arbitrary and can change in an apparently random way with further insertions into the collection.

Ordered When a collection is ordered, it means you can iterate through the collection in a specific (not-random) order. A Hashtable collection is not ordered. Although the Hashtable itself has internal logic to determine the order (based on hashcodes and the implementation of the collection itself), *you* won't find any order when you iterate through the Hashtable. An ArrayList, however, keeps the order established by the elements' index position (just like an array). LinkedHashSet keeps the order established by insertion, so the last element inserted is the last element in the LinkedHashSet (as opposed to an ArrayList where you can insert an element at a specific index position). Finally, there are some collections that keep an order referred to as the *natural order* of the elements, and those collections are then not just ordered, but also *sorted*. Let's look at how natural order works for sorted collections.

Sorted You know how to sort alphabetically—*A* comes before *B*, *F* comes before *G*, etc. For a collection of String objects, then, the *natural order* is alphabetical. For Integer objects, the natural order is by numeric value. And for Foo objects, the natural order is, um, we don't know. There *is no natural order* for Foo unless or until the Foo developer provides one, through an interface that defines how instances of a class can be compared to one another. For the exam, you don't need to know *how* to define natural order for your classes, only that you know there *is* such a thing as natural order and that it's used in sorted collections.

So, a sorted collection means *a collection sorted by natural order*. And *natural order is defined by the class of the objects being sorted* (or a supertype of that class, of course).

If you decide that Foo objects should be compared to one another (and thus sorted) using the value of their *bar* instance variables, then a sorted collection will order the Foo objects according to the rules in the Foo class for how to use the *bar* instance variable to determine the order. Again, you don't need to know *how* to define natural order, but keep in mind that *natural order* is not the same as an ordering determined by insertion, access, or index. A collection that keeps an order (such as insertion order) is not really considered sorted *unless it uses natural order* or, optionally, the ordering rules that you specify in the constructor of the sorted collection.

Figure 7-4 highlights the key distinctions between ordered and sorted collections.

FIGURE 7-4

What it means to be ordered or sorted

```
import java.util.*;
public class Ordered {
  public static void main(String [] args) {

    LinkedHashSet lhs = new LinkedHashSet();
    lhs.add ("Chicago");
    lhs.add ("Detroit");
    lhs.add ("Atlanta");
    lhs.add ("Denver");

    Iterator it = lhs.iterator();
    while (it.hasNext() ) (
      System.out.println("city " + it.next());
    }
  }
}
```

Output is
Chicago
Detroit
Atlanta
Denver

A LinkedHashSet, ***ordered*** by order of insertion

```
import java.util.*;
public class Sorted {
  public static void main(String [] args) {

    TreeSet ts = new TreeSet();
    ts.add ("Chicago");
    ts.add ("Detroit");
    ts.add ("Atlanta");
    ts.add ("Denver");

    Iterator it = ts.iterator();
    while (it.hasNext() ) (
      System.out.println("city " + it.next());
    }
  }
}
```

Output is
Atlanta
Chicago
Denver
Detroit

A TreeSet, ***sorted*** alphabetically

Now that we know about ordering and sorting, we'll look at each of the three interfaces, and then we'll dive into the concrete implementations of those interfaces.

List

A List cares about the index. The one thing that List has that nonlists don't have is a set of methods related to the index. Those key methods include things like `get(int index)`, `indexOf()`, `add(int index, Object obj)`, etc. (You don't need to memorize the method signatures.) All three List implementations are ordered by index position—a position that *you* determine either by setting an object at a specific index or by adding it *without* specifying position, in which case the object is added to the end. The three List implementations are described in the following section.

ArrayList Think of this as a growable array. It gives you *fast iteration* and *fast random access*. To state the obvious: it is an ordered collection (by index), but not sorted. You might want to know that as of version 1.4, ArrayList now implements the new RandomAccess interface—a marker interface (meaning it has no methods) that says, "this list supports fast (generally constant time) random access." Choose this over a LinkedList when you need fast iteration but aren't as likely to be doing a lot of insertion and deletion.

Vector Vector is a holdover from the earliest days of Java; Vector and Hashtable were the two original collections, the rest were added with Java 2 versions 1.2 and 1.4. A Vector is basically the same as an ArrayList, but `Vector()` methods are synchronized for thread safety. You'll normally want to use ArrayList instead of Vector because the synchronized methods add a performance hit you might not need. And if you *do* need thread safety, there are utility methods in class Collections that can help. Vector is the only class other than ArrayList to implement RandomAccess.

LinkedList A LinkedList List is ordered by index position, like ArrayList, except that the elements are doubly-linked to one another. This linkage gives you new methods (beyond what you get from the List interface) for adding and removing from the beginning or end, which makes it an easy choice for implementing a stack or queue. Keep in mind that a LinkedList may iterate more slowly than an ArrayList, but it's a good choice when you need fast insertion and deletion.

Set

A Set cares about uniqueness—it doesn't allow duplicates. Your good friend the `equals()` method determines whether two objects are identical (in which case only one can be in the set). The three Set implementations are described in the following sections.

HashSet A HashSet is an unsorted, unordered Set. It uses the hashcode of the object being inserted, so the more efficient your `hashCode()` implementation the better access performance you'll get. Use this class when you want a collection with no duplicates and you don't care about order when you iterate through it.

LinkedHashSet A LinkedHashSet is an ordered version of HashSet that maintains a doubly-linked List across all elements. Use this class instead of HashSet when you care about the iteration order; when you iterate though a HashSet the order is unpredictable, while a LinkedHashSet lets you iterate through the elements in the order in which they were inserted. Optionally, you can construct a LinkedHashSet so that it maintains the order in which elements were last *accessed*, rather than the order in which elements were inserted. That's a pretty handy feature if you want to build a least-recently-used cache that kills off objects (or flattens them) that haven't been used for awhile. (LinkedHashSet is a new collection class in version 1.4.)

TreeSet The TreeSet is one of two sorted collections (the other being TreeMap). It uses a Red-Black tree structure (but you knew that), and guarantees that the elements will be in ascending order, according to the *natural order* of the elements. Optionally, you can construct a TreeSet with a constructor that lets you give the collection your *own* rules for what the *natural order* should be (rather than relying on the ordering defined by the elements' class).

Map

A Map cares about unique identifiers. You *map* a unique *key* (the ID) to a specific *value*, where both the *key* and the *value* are of course *objects*. You're probably quite familiar with Maps since many languages support data structures that use a *key/value* or *name/value* pair. *Where* the keys land in the Map is based on the key's hashcode, so, like HashSet, the more efficient your `hashCode()` implementation, the better access performance you'll get. The Map implementations let you do things like search for a value based on the key, ask for a collection of just the values, or ask for a collection of just the keys.

HashMap The HashMap gives you an unsorted, unordered Map. When you need a Map and you don't care about the order (when you iterate through it), then HashMap is the way to go; the other maps add a little more overhead. HashMap allows one null key in a collection and multiple null values in a collection.

Hashtable Like Vector, Hashtable has been in from prehistoric Java times. For fun, don't forget to note the naming inconsistency: HashMap vs. Hashtable. Where's the capitalization of "t"? Oh well, you won't be expected to spell it. Anyway, just as Vector is a synchronized counterpart to the sleeker, more modern ArrayList, *Hashtable is the synchronized counterpart to HashMap*. Remember that you don't synchronize a *class*, so when we say that Vector and Hashtable are synchronized, we just mean that the key *methods* of the class are synchronized. Another difference, though, is that while HashMap lets you have null values as well as one null key, a *Hashtable doesn't let you have anything that's null.*

LinkedHashMap Like its Set counterpart, LinkedHashSet, the LinkedHashMap collection maintains insertion order (or, optionally, access order). Although it will be somewhat slower than HashMap for adding and removing elements, you can expect faster iteration with a LinkedHashMap. (LinkedHashMap is a new collection class as of version 1.4.)

TreeMap You can probably guess by now that a TreeMap is a *sorted* Map. And you already know that this means "sorted by the *natural order* of the elements." But like TreeSet, TreeMap lets you pass your *own* comparison rules in when you construct a TreeMap, to specify how the elements should be compared to one another when they're being ordered.

exam Watch

Look for incorrect mixtures of interfaces with classes. You can easily eliminate some answers right away if you recognize that, for example, a Map can't be the collection class you choose when you need a name/value pair collection, since Map is an interface and not a concrete implementation class. The wording on the exam is explicit when it matters, so if you're asked to choose an interface, choose an interface rather than a class that implements that interface. The reverse is also true—if you're asked to choose an implementation class, don't choose an interface type.

Whew! That's all the collection stuff you'll need for the exam, and Table 7-2 puts it in a nice little summary.

TABLE 7-2 Collection Interface Concrete Implementation Classes

Class	Map	Set	List	Ordered	Sorted
HashMap	X			No	No
Hashtable	X			No	No
TreeMap	X			Sorted	By *natural order* or custom comparison rules
LinkedHashMap	X			By insertion order or last access order	No
HashSet		X		No	No
TreeSet		X		Sorted	By *natural order* or custom comparison rules
LinkedHashSet		X		By insertion order or last access order	No
ArrayList			X	By index	No
Vector			X	By index	No
LinkedList			X	By index	No

exam Watch

Be sure you know how to interpret Table 7-2 in a practical way. For the exam, you might be expected to choose a collection based on a particular requirement, where that need is expressed as a scenario. For example, which collection would you use if you needed to maintain and search on a list of parts, identified by their unique alphanumeric serial where the part would be of type Part? Would you change your answer at all if we modified the requirement such that you also need to be able to print out the parts in order, by their serial number? For the first question, you can see that since you have a Part class, but need to search for the objects based on a serial number, you need a Map. The *key* ***will be the serial number as a String, and the*** *value* ***will be the Part instance. The default choice should be HashMap, the quickest Map for access. But now when we amend the requirement to include getting the parts in order of their serial number, then we need a TreeMap—which maintains the*** *natural order* ***of the keys. Since the*** *key* ***is a String, the*** *natural order* ***for a String will be a standard alphabetical sort. If the requirement had been to keep track of which part was last accessed, then we'd probably need a LinkedHashMap.***

But since a LinkedHashMap loses the natural order (replacing it with last-accessed order), if we need to list the parts by serial number, we'll have to explicitly sort the collection, using a utility method.

Now that you know how to compare, organize, access, and sort objects, there's only one thing left to learn in this sequence: how to get *rid* of objects. The last objective in this chapter looks at the garbage collection system in Java. You simply won't believe how many garbage collection questions are likely to show up on your exam, so pay close attention to this last section. Most importantly, you'll need to know what is and is not guaranteed and what you're responsible for when it comes to memory management in Java.

CERTIFICATION OBJECTIVE

Garbage Collection (Exam Objectives 3.1, 3.2, 3.3)

State the behavior that is guaranteed by the garbage collection system.
Write code that explicitly makes objects eligible for garbage collection.
Recognize the point in a piece of source code at which an object becomes eligible for garbage collection.

Overview of Memory Management and Garbage Collection

This is the section you've been waiting for! It's finally time to dig into the wonderful world of memory management and garbage collection.

Memory management is a crucial element in many types of applications. Consider a program that reads in large amounts of data, say from somewhere else on a network, and then writes that data into a database on a hard drive. A typical design would be to read the data into some sort of collection in memory, perform some operations on the data, and then write the data into the database. After the data is written into the database, the collection that stored the data temporarily must be emptied of old data or deleted and re-created before processing the next batch. This operation might be performed thousands of times, and in languages like C or C++ that do not offer automatic garbage collection, a small flaw in the logic that manually empties or deletes the collection data structures can allow small amounts of memory to be

improperly reclaimed or lost. Forever. These small losses are called *memory leaks*, and over many thousands of iterations they can make enough memory inaccessible that programs will eventually crash. Creating code that performs manual memory management cleanly and thoroughly is a nontrivial and complex task, and while estimates vary, it is arguable that manual memory management can *double* the development effort for a complex program.

Java's garbage collector provides an automatic solution to memory management. In most cases it frees you from having to add any memory management logic to your application. The downside to automatic garbage collection is that you can't completely control when it runs and when it doesn't.

Overview of Java's Garbage Collector

Let's look at what we mean when we talk about garbage collection in the land of Java. From the 30,000 ft. level, garbage collection is the phrase used to describe automatic memory management in Java. Whenever a software program executes (in Java, C, C++, Lisp, etc.), it uses memory in several different ways. We're not going to get into Computer Science 101 here, but it's typical for memory to be used to create a stack, a heap, in Java's case constant pools, and method areas. The *heap* is that part of memory where Java objects live, and it's the one and only part of memory that is in any way involved in the garbage collection process.

exam Watch

A heap is a heap is a heap. For the exam it's important to know that you can call it the heap, you can call it the garbage collectible heap, you can call it Johnson, but there is one and only one heap.

So, all of garbage collection revolves around making sure that the heap has as much free space as possible. For the purpose of the exam, what this boils down to is deleting any objects that are no longer *reachable* by the Java program running. We'll talk more about what *reachable* means, but let's drill this point in. *When the garbage collector runs, its purpose is to find and delete objects that cannot be reached.* If you think of a Java program as in a constant cycle of creating the objects it needs (which occupy space on the heap), and then discarding them when they're no longer needed, creating new objects, discarding them, and so on, the missing piece of the puzzle is the garbage collector. When it runs, it looks for those discarded objects and deletes them from memory so that the cycle of using memory and releasing it can continue. Ah, the great circle of life.

When Does the Garbage Collector Run?

The garbage collector is under the control of the JVM. The JVM decides when to run the garbage collector. From within your Java program you can ask the JVM to run the garbage collector, but there are no guarantees, under any circumstances, that the JVM will comply. Left to its own devices, the JVM will typically run the garbage collector when it senses that memory is running low. Experience indicates that when your Java program makes a request for garbage collection, the JVM will usually grant your request in short order, *but there are no guarantees.* Just when you think you can count on it, the JVM will decide to ignore your request.

How Does the Garbage Collector Work?

You just can't be sure. You might hear that the garbage collector uses a *mark and sweep* algorithm, and for any given Java implementation that *might* be true, but the Java specification doesn't guarantee any particular implementation. You might hear that the garbage collector uses *reference counting;* once again maybe yes maybe no. The important concept to understand for the exam is *when does an object become eligible for garbage collection.* To answer this question fully we have to jump ahead a little bit and talk about *threads.* (See Chapter 9 for the real scoop on threads.) In a nutshell, every Java program has from one to many threads. Each thread has its own little execution stack. Normally, you (the programmer), cause at least one thread to run in a Java program, the one with the `main()` method at the bottom of the stack. However, as you'll learn in excruciating detail in Chapter 9, there are many really cool reasons to launch additional threads from your initial thread. In addition to having its own little execution stack, each thread has its own lifecycle. For now, all we need to know is that threads can be alive or dead. With this background information we can now say with stunning clarity and resolve that, *an object is eligible for garbage collection when no live thread can access it.*

Based on that definition, the garbage collector does some magical, unknown operations, and when it discovers an object that can't be reached by any live thread it will consider that object as eligible for deletion, and it might even delete it at some point. (You guessed it, it also might *not* ever delete it.) When we talk about *reaching* an object, we're really talking about having a *reachable* reference variable that refers to the object in question. If our Java program has a reference variable that refers to an object, and that reference variable is available to a live thread, then that object is considered *reachable.* We'll talk more about how objects can become unreachable in the following section.

exam Watch

Can a Java application run out of memory? Yes. The garbage collection system attempts to remove objects from memory when they are not used. However, if you maintain too many live objects (objects referenced from other live objects), the system can run out of memory. Garbage collection cannot ensure that there is enough memory, only that the memory that is available will be managed as efficiently as possible.

Writing Code That Explicitly Makes Objects Eligible for Collection

In the previous section, we learned the theories behind Java garbage collection. In this section, we show how to make objects eligible for garbage collection using actual code. We also discuss how to attempt to force garbage collection if it is necessary, and how you can perform additional cleanup on objects before they are removed from memory.

Nulling a Reference

As we discussed earlier, an object becomes eligible for garbage collection when there are no more reachable references to it. Obviously, if there are no reachable references, it doesn't matter what happens to the object. For our purposes it is just floating in space, unused, inaccessible, and no longer needed.

The first way to remove a reference to an object is to set the reference variable that refers to the object to `null`. Examine the following code:

```
1. public class GarbageTruck {
2.   public static void main(String [] args) {
3.     StringBuffer sb = new StringBuffer("hello");
4.     System.out.println(sb);
5.     // The StringBuffer object is not eligible for collection
6.     sb = null;
7.     // Now the StringBuffer object is eligible for collection
8.   }
9. }
```

The StringBuffer object with the value `hello` is assigned the reference variable *sb* in the third line. To make it eligible, we set the reference variable *sb* to `null`, which removes the single reference that existed to the StringBuffer object. Once line 6 has run, our happy little `hello` StringBuffer object is doomed, eligible for garbage collection.

Reassigning a Reference Variable

We can also decouple a reference variable from an object by setting the reference variable to refer to another object. Examine the following code:

```
class GarbageTruck {
   public static void main(String [] args) {
      StringBuffer s1 = new StringBuffer("hello");
      StringBuffer s2 = new StringBuffer("goodbye");
      System.out.println(s1);
      // At this point the StringBuffer "hello" is not eligible
      s1 = s2; // Redirects s1 to refer to the "goodbye" object
      // Now the StringBuffer "hello" is eligible for collection
   }
}
```

Objects that are created in a method also need to be considered. When a method is invoked, any local variables created exist only for the duration of the method. Once the method has returned, the objects created in the method are eligible for garbage collection. There is an obvious exception, however. If an object is returned from the method, its reference might be assigned to a reference variable in the method that called it; hence, it will not be eligible for collection. Examine the following code:

```
import java.util.Date;
public class GarbageFactory {
   public static void main(String [] args) {
      Date d = getDate()
      doComplicatedStuff();
      System.out.println("d = " + d);
   }

   public static Date getDate() {
      Date d2 = new Date();
      String now = d2.toString();
      System.out.println(now);
      return d2;
   }
}
```

In the preceding example, we created a method called `getDate()` that returns a Date object. This method creates two objects: a Date and a String containing the date information. Since the method returns the Date object, it will *not* be eligible for collection even after the method has completed. The String object, though, will be eligible, even though we did not explicitly set the *now* variable to `null`.

Isolating a Reference

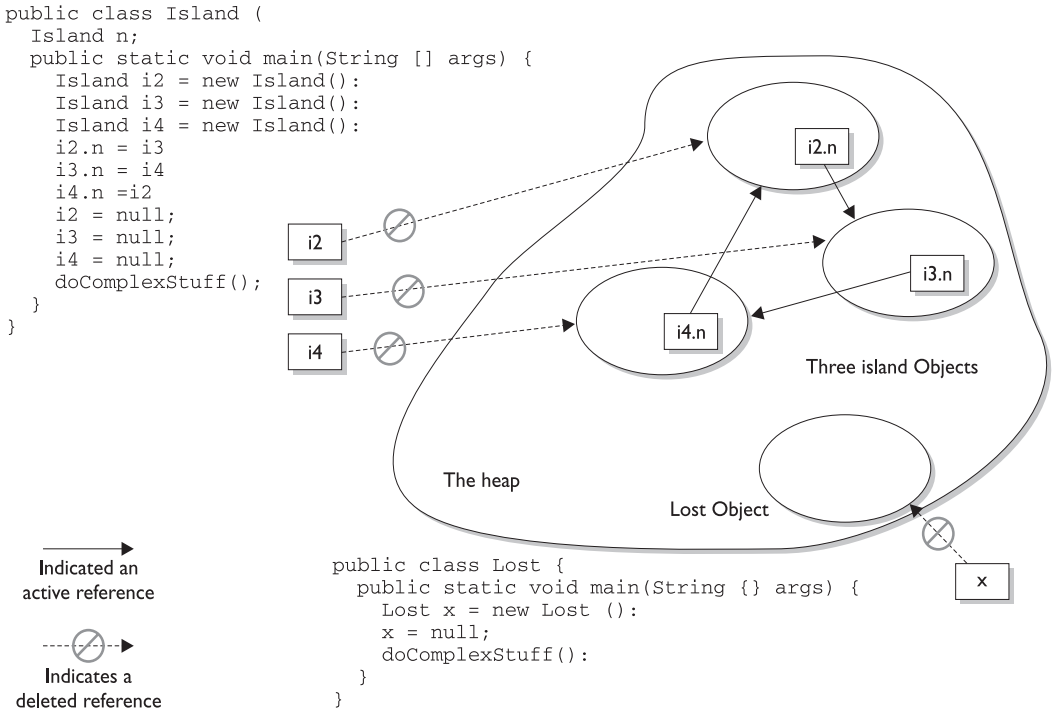
There is another way in which objects can become eligible for garbage collection, *even if they still have valid references!* We think of this scenario as *islands of isolation*. A simple example is a class that has an instance variable that is a reference variable to another instance of the same class. Now imagine that two such instances exist and that they refer to each other. If all other references to these two objects are removed, then even though each object still has a valid reference, there will be no way for any live thread to access either object. When the garbage collector runs, it will discover any such *islands* of objects and will remove them. As you can imagine, such islands can become quite large, theoretically containing hundreds of objects. Examine the following code:

```
public class Island {
  Island i;
  public static void main(String [] args) {

    Island i2 = new Island();
    Island i3 = new Island();
    Island i4 = new Island();

    i2.i = i3;   // i2 refers to i3
    i3.i = i4;   // i3 refers to i4
    i4.i = i2;   // i4 refers to i2

    i2 = null;
    i3 = null;
    i4 = null;

    // do complicated, memory intensive stuff
  }
}
```

When the code reaches `// do complicated,` the three Island objects (previously known as i2, i3, and i4) have instance variables so that they refer to each other, but their links to the outside world (i2, i3, and i4) have been nulled. These three objects are eligible for garbage collection.

This covers everything you will need to know about making objects eligible for garbage collection. Study Figure 7-5 to reinforce the concepts of objects without references and *islands of isolation*.

FIGURE 7-5 Objects eligible for garbage collection



Forcing Garbage Collection

The first thing that should be mentioned here is, contrary to this section's title, *garbage collection cannot be forced*. However, Java provides some methods that allow you to request that the JVM perform garbage collection. For example, if you are about to perform some time-sensitive operations, you probably want to minimize the chances of a delay caused by garbage collection. But you must remember that the methods that Java provides are *requests*, and not demands; the virtual machine will do its best to do what you ask, but there is no guarantee that it will comply.

exam Watch

In reality, it is possible only to suggest to the JVM that it perform garbage collection. However, there are no guarantees the JVM will actually remove all of the unused objects from memory. It is essential that you understand this concept for the exam.

The garbage collection routines that Java provides are members of the Runtime class. The Runtime class is a special class that has a single object (a Singleton) for each main program. The Runtime object provides a mechanism for communicating directly with the virtual machine. In order to get the Runtime instance, you can use the method `Runtime.getRuntime()`, which returns the Singleton. Alternatively, for the method we are going to discuss, you can call the same method on the System class, which has static methods that can do the work of obtaining the Singleton for you. The simplest way to ask for garbage collection (remember—just a request) is

```
System.gc();
```

Theoretically, after calling `System.gc()`, you will have as much free memory as possible. We say *theoretically* because this routine does not always work that way. First, the JVM you are using may not have implemented this routine; the language specification allows this routine to do nothing at all. Second, another thread (again, see Chapter 9) may perform a substantial memory allocation right after you run the garbage collection.

This is not to say that `System.gc()` is a useless method—it's much better than nothing. You just can't rely on `System.gc()` to free up enough memory so that you don't have to worry about the garbage collector being run. The certification exam is interested in guaranteed behavior, not probable behavior.

Now that we are somewhat familiar with how this works, let's do a little experiment to see if we can see the effects of garbage collection. The following program lets us know how much total memory the JVM has available to it and how much free memory it has. It then creates 10,000 Date objects. After this, it tells us how much memory is left and then calls the garbage collector (which, if it decides to run, should halt the program until all unused objects are removed). The final free memory result should indicate whether it has run. Let's look at the program:

```
 1.  import java.util.Date;
 2.  public class CheckGC {
 3.     public static void main(String [] args) {
 4.        Runtime rt = Runtime.getRuntime();
 5.        System.out.println("Total JVM memory: " + rt.totalMemory());
 6.        System.out.println("Before Memory = " + rt.freeMemory());
 7.        Date d = null;
 8.        for(int i = 0;i<10000;i++) {
 9.           d = new Date();
10.           d = null;
11.        }
12.        System.out.println("After Memory = " + rt.freeMemory());
13.        rt.gc();   // an alternate to System.gc()
14.        System.out.println("After GC Memory = " + rt.freeMemory());
```

```
15.     }
16.  }
```

Now, let's run the program and check the results:

```
Total JVM memory: 1048568
Before Memory = 703008
After Memory = 458048
After GC Memory = 818272
```

As we can see, the VM actually did decide to garbage collect (i.e. delete) the eligible objects. In the preceding example, we suggested to the JVM to perform garbage collection with 458,048 bytes of memory remaining, and it honored our request. This program has only one user thread running, so there was nothing else going on when we called `rt.gc()`. Keep in mind that the behavior when `gc()` is called may be different for different JVMs, so there is no guarantee that the unused objects will be removed from memory. About the only thing you *can* guarantee is that if you *are* running very low on memory, the garbage collector will run before it throws an `OutOfMemoryException`.

Cleaning Up Before Garbage Collection—the Finalize() Method

Java provides you a mechanism to run some code just before your object is deleted by the garbage collector. This code is located in a method named `finalize()` that all classes inherit from class Object. On the surface this sounds like a great idea; maybe your object opened up some resources, and you'd like to close them before your object is deleted. The problem is that, as you may have gathered by now, you can't count on the garbage collector to ever delete an object. *So, any code that you put into your class's overridden `finalize()` method is **not** guaranteed to run.* The `finalize()` method for any given object might run, but you can't count on it, so don't put any essential code into your `finalize()` method. In fact, we recommend that in general you don't override `finalize()` at all.

Tricky Little Finalize() Gotcha's

There are a couple of concepts concerning `finalize()` that you need to remember.

- For any given object, `finalize()` will be called only once by the garbage collector.
- Calling `finalize()` can actually result in saving an object from deletion.

Let's look into these statements a little further. First of all, remember that `finalize()` is a method, and any code that you can put into a *normal* method you can put into `finalize()`. For example, in the `finalize()` method you could write code that passes a reference to the object in question back to another object, effectively *uneligiblizing* the object for garbage collection. If at some point later on this same object becomes eligible for garbage collection again, the garbage collector can still process this object and delete it. The garbage collector, however, will remember that, *for this object*, `finalize()` already ran, and it will not run `finalize()` again.

Now that we've gotten down and dirty with garbage collection, verify that the following scenarios and solutions make sense to you. If they don't, reread the last part of this chapter. *While awake.*

SCENARIO & SOLUTION

I want to allocate an object and make sure that it never is deallocated. Can I tell the garbage collector to ignore an object?	No. There isn't a mechanism for marking an object as undeletable. You can instead create a static member of a class, and store a reference to the object in that. Static members are considered live objects.
My program is not performing as well as I would expect. I think the garbage collector is taking too much time. What can I do?	First, if it really is the garbage collector (and it probably isn't), then the code is creating and dropping many references to many temporary objects. Try to redesign the program to reuse objects or require fewer temporary objects.
I am creating an object in a method and passing it out as the method result. How do I make sure the object isn't deleted before the method returns?	The object won't be deleted until the last reference to the object is dropped. If you return the object as a method return value, the method that called it will contain a reference to the object.
How do I drop a reference to an object if that object is referred to in a member of my class?	Set the member to `null`. Alternatively, if you set a reference to a new object, the old object loses one reference. If that is the last reference, the object becomes eligible for deletion.
I want to keep objects around as long as they don't interfere with memory allocation. Is there any way I can ask Java to warn me if memory is getting low?	Prior to Java 1.2, you would have to check the amount of free memory yourself and guess. Java 1.2 introduced soft references for just this situation. This is not part of the Java 2 exam, however.

FROM THE CLASSROOM

Advanced Garbage Collection in Java 2

Up to this point, we have been discussing the original Java memory management model. With Java 2, the original model was augmented with *reference classes*. Reference classes, which derive from the abstract class *Reference*, are used for more sophisticated memory management. (You will not need to know the advanced management model for the exam.) The Reference class is the superclass for the *WeakReference, SoftReference*, and *PhantomReference* classes found in the java.lang.ref package.

By default, you as a programmer work with strong references. When you hear people talking about references (at parties, on the bus), they are usually talking about *strong references*. This was the classic Java way of doing things, and it is what you have unless you go out of your way to use the Reference classes. Strong references are used to prevent objects from being garbage collected; a strong reference from a reachable object is enough to keep the referred-to object in memory.

Let's look at the other three types of references:

- **Soft references** The Java language specification states that soft references can be used to create memory-sensitive caches. For example, in an image program, when you make a change to the image (say, an Image object), the old Image object can stick around in case the user wants to undo the change. This old object is an example of a *cache*.
- **Weak references** These are similar to soft references in that they allow you to refer to an object without forcing the object to remain in memory. Weak references are different from soft references, however, in that they do not request that the garbage collector attempt to keep the object in memory. Unlike soft references, which may stick around for a while even after their strong references drop, weak references go away pretty quickly.
- **Phantom references** These provide a means of delaying the reuse of memory occupied by an object, even if the object itself is finalized. A *phantom* object is one that has been finalized, but whose memory has not yet been made available for another object.

Objects are placed into one of several categories, depending on what types of references can be used to get to the object. References are ordered as follows: strong, soft, weak, and phantom. Objects are then known as *strongly reachable*, *softly reachable*, *weakly reachable*, *phantom reachable*, or *unreachable.*

- **Strongly reachable** If an object has a strong reference, a soft reference, a weak

reference, and a phantom reference all pointing to it, then the object is considered strongly reachable and will not be collected.

- **Softly reachable** An object without a strong reference but with a soft reference, a weak reference, and a phantom reference, will be considered softly reachable and will be collected only when memory gets low.
- **Weakly reachable** An object without a strong or soft reference but with a weak or phantom reference, is considered weakly reachable and will be collected at the next garbage collection cycle.
- **Phantom reachable** An object without a strong, soft, or weak reference but with a phantom reference, is considered phantom reachable and will be finalized, but the memory for that object will not be collected.
- **Unreachable** What about an object without a strong, soft, weak, or phantom reference? Well, that object is considered unreachable and will already have been collected, or will be collected as soon as the next garbage collection cycle is run.

— Bob Hablutzel

CERTIFICATION SUMMARY

As you know by now, when we come to this point in the chapter (the end), we like to pause for a moment and review all that we've done. We began by looking at the `hashCode()` and `equals()` methods, with a quick review of another important method in class Object, `toString()`.You learned that overriding `toString()` is your opportunity to create a meaningful summary (in the form of a String) of the state of any given instance in your classes. The `toString()` method is automatically called when you ask `System.out.println()` to print an object.

Next you reviewed the purpose of == (to see if two reference variables refer to the *same object*) and the `equals()` method (to see if two objects are *meaningfully equivalent*). You learned the downside of not overriding `equals()`—you may not be able to find the object in a collection. We discussed a little bit about how to write a good `equals()` method—don't forget to use `instanceof` and refer to the object's significant attributes. We reviewed the contracts for overriding `equals()` and `hashCode()`. We learned about the theory behind hashcodes, the difference

between legal, appropriate, and efficient hashcoding. We also saw that even though wildly inefficient, it's legal for a `hashCode()` method to always return the same value.

Next we turned to collections, where we learned about *Lists, Sets, and Maps,* and the difference between ordered and sorted collections. We learned the key attributes of the common collection classes, and when to use which. Finally, we dove into garbage collection, Java's automatic memory management feature. We learned that the heap is where objects live and where all the cool garbage collection activity takes place. We learned that in the end, the JVM will perform garbage collection whenever it wants to. You (the programmer) can request a garbage collection run, but you can't force it. We talked about garbage collection only applying to objects that are *eligible*, and that *eligible* means "inaccessible from any live thread." Finally, we discussed the rarely useful `finalize()` method, and what you'll have to know about it for the exam. All in all one fascinating chapter.

TWO-MINUTE DRILL

Here are some of the key points from Chapter 7.

Overriding hashCode() and equals()

- ❑ The critical methods in class Object are `equals()`, `finalize()`, `hashCode()`, and `toString()`.
- ❑ `equals()`, `hashCode()`, and `toString()` are public (`finalize()` is protected).
- ❑ Fun facts about `toString()`:
  - ❑ Override `toString()` so that `System.out.println()` or other methods can see something useful.
  - ❑ Override `toString()` to return the essence of your object's state.
- ❑ Use == to determine if two reference variables refer to the same object.
- ❑ Use `equals()` to determine if two objects are *meaningfully equivalent*.
- ❑ If you don't override `equals()`, your objects *won't* be useful hashtable/hashmap keys.
- ❑ If you don't override `equals()`, two different objects can't be considered *the same*.
- ❑ Strings and wrappers override `equals()` and make good hashtable/hashmap keys.
- ❑ When overriding `equals()`, use the `instanceof` operator to be sure you're evaluating an appropriate class.
- ❑ When overriding `equals()`, compare the objects' *significant* attributes.
- ❑ Highlights of the `equals()` contract:
  - ❑ *Reflexive:* `x.equals(x)` is `true`.
  - ❑ *Symmetric:* If `x.equals(y)` is `true`, then `y.equals(x)` must be `true`.
  - ❑ *Transitive:* If `x.equals(y)` is `true`, and `y.equals(z)` is `true`, then `z.equals(x)` is `true`.

- *Consistent:* Multiple calls to `x.equals(y)` will return the same result.
- *Null:* If `x` is not `null`, then `x.equals(null)` is `false`.

- If `x.equals(y)` is `true`, then `x.hashCode() == y.hashCode()` must be `true`.
- If you override `equals()`, override `hashCode()`.
- Classes HashMap, Hashtable, LinkedHashMap, and LinkedHashSet use hashing.
- A *legal* `hashCode()` override compiles and runs.
- An *appropriate* `hashCode()` override sticks to the contract.
- An *efficient* `hashCode()` override distributes keys randomly across a wide range of buckets.
- To reiterate: if two objects are equal, their hashcodes must be equal.
- It's *legal* for a `hashCode()` method to return the same value for all instances (although in practice it's very inefficient).
- Highlights of the `hashCode()` contract:
  - *Consistent:* Multiple calls to `x.hashCode()` return the same integer.
  - If `x.equals(y)` is `true`, then `x.hashCode() == y.hashCode()` must be `true`.
  - If `x.equals(y)` is `false`, then `x.hashCode() == y.hashCode()` can be either `true` or `false`, but `false` will tend to create better efficiency.
- Transient variables aren't appropriate for `equals()` and `hashCode()`.

Collections

- Common collection activities include adding objects, removing objects, verifying object inclusion, retrieving objects, and iterating.
- Three meanings for "collection":
  - collection—Represents the data structure in which objects are stored
  - Collection—`java.util.Collection`—Interface from which Set and List extend
  - Collections—A class that holds static collection utility methods

- ❑ Three basic *flavors* of collections include *Lists, Sets, Maps*:
  - ❑ Lists of things: Ordered, *duplicates allowed,* with an *index*
  - ❑ Sets of things: May or may not be ordered and/or sorted, *duplicates not allowed*
  - ❑ Maps of things with keys: May or may not be ordered and/or sorted, *duplicate keys not allowed*
- ❑ Four basic *subflavors* of collections include *Sorted, Unsorted, Ordered, Unordered*.
- ❑ Ordered means iterating through a collection in a specific, nonrandom order.
- ❑ Sorted means iterating through a collection in a *natural* sorted order.
- ❑ Natural means alphabetic, numeric, or programmer-defined, whichever applies.
- ❑ Key attributes of common collection classes:
  - ❑ ArrayList: Fast iteration and fast random access
  - ❑ Vector: Like a somewhat slower ArrayList, mainly due to its synchronized methods
  - ❑ LinkedList: Good for adding elements to the ends, i.e., stacks and queues
  - ❑ HashSet: Assures no duplicates, provides no ordering
  - ❑ LinkedHashSet: No duplicates; iterates by insertion order or last accessed (new with 1.4)
  - ❑ TreeSet: No duplicates; iterates in *natural* sorted order
  - ❑ HashMap: Fastest updates (key/value pairs); allows one null key, many null values
  - ❑ Hashtable: Like a slower HashMap (as with Vector, due to its synchronized methods). No null values or null keys allowed
  - ❑ LinkedHashMap: Faster iterations; iterates by insertion order or last accessed, allows one null key, many null values (new with 1.4)
  - ❑ TreeMap: A sorted map, in *natural* order

Garbage Collection

- ❑ In Java, garbage collection provides some automated memory management.
- ❑ All objects in Java live on the *heap*.

- The *heap* is also known as the *garbage collectible heap*.
- The purpose of garbage collecting is to find and delete objects that can't be reached.
- Only the *JVM* decides exactly when to run the garbage collector.
- You (the programmer) can only *recommend* when to run the garbage collector.
- You can't know the G.C. algorithm; *maybe* it uses mark and sweep, *maybe* it's generational and/or iterative.
- Objects must be considered *eligible* before they can be garbage collected.
- An object is eligible when no live thread can *reach* it.
- To reach an object, a live thread must have a live, reachable reference variable to that object.
- Java applications can run out of memory.
- Islands of objects can be garbage collected, even though they have references.
- To reiterate: garbage collection can't be forced.
- Request garbage collection with `System.gc();` (recommended).
- Class Object has a `finalize()` method.
- The `finalize()` method is guaranteed to run *once and only once* before the garbage collector deletes an object.
- Since the garbage collector makes no guarantees, *`finalize()` may never run.*
- You can uneligibilize an object from within `finalize()`.

SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. Read all of the choices carefully, as there may be more than one correct answer. Choose all correct answers for each question.

HashCode and equals() (Exam Objective 9.2)

1. Given the following,

```
11.  x = 0;
12.  if (x1.hashCode() != x2.hashCode() )  x = x + 1;
13.  if (x3.equals(x4) )  x = x + 10;
14.  if (!x5.equals(x6) ) x = x + 100;
15.  if (x7.hashCode() == x8.hashCode() )  x = x + 1000;
16.  System.out.println("x = " + x);
```

and assuming that the equals () and hashCode() methods are property implemented and x1 – x8 are all of the same type, if the output is "`x = 1111`", which of the following statements will always be true?

A. `x2.equals(x1)`

B. `x3.hashCode() == x4.hashCode()`

C. `x5.hashCode() != x6.hashCode()`

D. `x8.equals(x7)`

2. Given the following,

```
class Test1 {
  public int value;
  public int hashCode() { return 42; }
}
class Test2 {
  public int value;
  public int hashcode() { return (int)(value^5); }
}
```

which statement is true?

A. class Test1 will not compile.

B. The Test1 `hashCode()` method is more efficient than the Test2 `hashCode()` method.

C. The Test1 `hashCode()` method is less efficient than the Test2 `hashCode()` method.

D. class Test2 will not compile.

E. The two `hashcode()` methods will have the same efficiency.

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

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

B. If the `equals()` method returns `false`, the `hashCode()` comparison != must 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`.

4. Which class does not override the `equals()` and `hashCode()` methods, inheriting them directly from class Object?

A. java.lang.String

B. java.lang.Double

C. java.lang.StringBuffer

D. java.lang.Character

E. java.util.ArrayList

5. What two statements are true about properly overridden `hashCode()` and `equals()` methods?

A. `hashCode()` doesn't have to be overridden if `equals()` is.

B. `equals()` doesn't have to be overridden if `hashCode()` is.

C. `hashCode()` can always return the same value, regardless of the object that invoked it.

D. If two different objects that are not meaningfully equivalent both invoke `hashCode()`, then `hashCode()` can't return the same value for both invocations.

E. `equals()` can be true even if it's comparing different objects.

Using Collections (Exam Objective 9.1)

6. Which collection class allows you to grow or shrink its size and provides indexed access to its elements, but whose methods are not synchronized?

A. java.util.HashSet
B. java.util.LinkedHashSet
C. java.util.List
D. java.util.ArrayList
E. java.util.Vector

7. Which collection class allows you to access its elements by associating a key with an element's value, and provides synchronization?

A. java.util.SortedMap
B. java.util.TreeMap
C. java.util.TreeSet
D. java.util.HashMap
E. java.util.Hashtable

8. Given the following,

```
12.    TreeSet map = new TreeSet();
13.    map.add("one");
14.    map.add("two");
15.    map.add("three");
16.    map.add("four");
17.    map.add("one");
18.    Iterator it = map.iterator();
19.    while (it.hasNext() ) {
20.        System.out.print( it.next() + " " );
21.     }
```

what is the result?

A. one two three four
B. four three two one
C. four one three two
D. one two three four one
E. one four three two one
F. The print order is not guaranteed.

9. Which collection class allows you to associate its elements with key values, and allows you to retrieve objects in FIFO (first-in, first-out) sequence?

A. java.util.ArrayList
B. java.util.LinkedHashMap
C. java.util.HashMap
D. java.util.TreeMap
E. java.util.LinkedHashSet
F. java.util.TreeSet

Garbage Collection (Exam Objectives 3.1, 3.2, 3.3)

10. Given the following,

```
1.  public class X {
2.    public static void main(String [] args) {
3.      X x = new X();
4.      X x2 = m1(x);
5.      X x4 = new X();
6.      x2 = x4;
7.      doComplexStuff();
8.    }
9.    static X m1(X mx) {
10.     mx = new X();
11.     return mx;
12.   }
13. }
```

After line 6 runs. how many objects are eligible for garbage collection?

A. 0
B. 1
C. 2
D. 3
E. 4

11. Which statement is true?

A. All objects that are eligible for garbage collection will be garbage collected by the garbage collector.
B. Objects with at least one reference will never be garbage collected.

C. Objects from a class with the `finalize()` method overridden will never be garbage collected.

D. Objects instantiated within anonymous inner classes are placed in the garbage collectible heap.

E. Once an overridden `finalize()` method is invoked, there is no way to make that object ineligible for garbage collection.

12. Given the following,

```
1.   class X2 {
2.     public X2 x;
3.     public static void main(String [] args) {
4.       X2 x2 = new X2();
5.       X2 x3 = new X2();
6.       x2.x = x3;
7.       x3.x = x2;
8.       x2 = new X2();
9.       x3 = x2;
10.      doComplexStuff();
11.    }
12.  }
```

after line 9 runs, how many objects are eligible for garbage collection?

A. 0

B. 1

C. 2

D. 3

E. 4

13. Which statement is true?

A. Calling `Runtime.gc()` will cause eligible objects to be garbage collected.

B. The garbage collector uses a mark and sweep algorithm.

C. If an object can be accessed from a live thread, it can't be garbage collected.

D. If object 1 refers to object 2, then object 2 can't be garbage collected.

14. Given the following,

```
12.  X3 x2 = new X3();
13.  X3 x3 = new X3();
14.  X3 x5 = x3;
15.  x3 = x2;
16.  X3 x4 = x3;
```

```
17.  x2 = null;
18.  //  insert code
```

what two lines of code, inserted independently at line 18, will make an object eligible for garbage collection? (Choose two.)

A. `x3 = null;`

B. `x4 = null;`

C. `x5 = null;`

D. `x3 = x4;`

E. `x5 = x4;`

15. Given the following,

```
12.  void doStuff3() {
13.    X x = new X();
14.    X y = doStuff(x);
15.    y = null;
16.    x = null;
17.  }
18.  X doStuff(X mx) {
19.     return doStuff2(mx);
20.  }
```

at what point is the object created in line 13 eligible for garbage collection?

A. After line 15 runs

B. After line 16 runs

C. After line 17 runs

D. The object is not eligible.

E. It is not possible to know for sure.

SELF TEST ANSWERS

Strings (Exam Objective 9.2)

1. ☑ **B.** By contract, if two objects are equivalent according to the `equals()` method, then the `hashCode()` method must evaluate them to be ==.
   ☒ **A** is incorrect because if the `hashCode()` values are not equal, the two objects *must not* be equal. **C** is incorrect because if `equals()` is not true there is no guarantee of any result from `hashCode()`. **D** is incorrect because `hashCode()` will often return == even if the two objects do not evaluate to `equals()` being true.

2. ☑ **C.** The so-called "hashing algorithm" implemented by class Test1 will always return the same value, 42, which is legal but which will place all of the hash table entries into a single bucket, the most inefficient setup possible.
   ☒ **A** and **D** are incorrect because these classes are legal. **B** and **E** are incorrect based on the logic described above.

3. ☑ **A** and **D. A** is a restatement of the `equals()` and `hashCode()` contract. **D** is true because if the `hashCode()` comparison returns ==, the two objects might or might not be equal.
   ☒ **B, C,** and **E** are incorrect. **B** and **C** are incorrect because the `hashCode()` method is very flexible in its return values, and often two dissimilar objects can return the same hash code value. **E** is a negation of the `hashCode()` and `equals()` contract.

4. ☑ **C.** java.lang.StringBuffer is the only class in the list that uses the default methods provided by class Object.
   ☒ **A, C, D, E,** and **F** are incorrect based on the logic described above.

5. ☑ **C** and **E** are correct.
   ☒ **A, B,** and **D** are incorrect. **A** and **B** are incorrect because by contract `hashCode()` and `equals()` can't be overridden unless both are overridden. **D** is incorrect; `hashCode()` will often return the same value when hashing dissimilar objects.

Using Collections (Exam Objective 9.1)

6. ☑ **D.** All of the collection classes allow you to grow or shrink the size of your collection. ArrayList provides an index to its elements. The newer collection classes tend not to have synchronized methods. Vector is an older implementation of ArrayList functionality and has synchronized methods; it is slower than ArrayList.
   ☒ **A, B, C,** and **E** are incorrect based on the logic described above; **C,** *List* is an interface.

7. ☑ **E.** Hashtable is the only class listed that provides synchronized methods. If you need synchronization great; otherwise, use HashMap, it's faster.
   ☒ **A, B, C,** and **D** are incorrect based on the logic described above.

8. ☑ **C.** TreeSet assures no duplicate entries; also, when it is accessed it will return elements in *natural order*, which typically means alphabetical.
☒ **A, B, D, E,** and **F** are incorrect based on the logic described above.

9. ☑ **B.** LinkedHashMap is the collection class used for caching purposes. FIFO is another way to indicate caching behavior. To retrieve LinkedHashMap elements in cached order, use the `values()` method and iterate over the resultant collection.
☒ **A, C, D, E,** and **E** are incorrect based on the logic described above.

Garbage Collection (Exam Objectives 3.1, 3.2, 3.3)

10. ☑ **B.** By the time line 6 has run, the only object without a reference is the one generated as a result of line 4. Remember that "Java is pass by value," so the reference variable `x` is not affected by the `m1()` method.
☒ **A, C, D,** and **E** are incorrect based on the logic described above.

11. ☑ **D.** All objects are placed in the garbage collectible heap.
☒ **A** is incorrect because the garbage collector makes no guarantees. **B** is incorrect because *islands* of isolated objects can exist. **C** is incorrect because `finalize()` has no such mystical powers. **E** is incorrect because within a `finalize()` method, an object's reference can be passed back to a live thread.

12. ☑ **C.** This is an example of the *islands of isolated objects*. By the time line 9 has run, the objects instantiated in lines 4 and 5 are referring to each other, but no live thread can reach either of them.
☒ **A, B, D,** and **E** are incorrect based on the logic described above.

13. ☑ **C.** This is a great way to think about when objects can be garbage collected.
☒ **A** and **B** assume guarantees that the garbage collector never makes. **D** is wrong because of the now famous *islands of isolation* scenario.

14. ☑ **C** and **E.** By the time line 18 is reached, `x2` is null, `x3` and `x4` refer to the object created in line 12, and `x5` refers to the object created in line 13. Any kind of redirection of `x5` will leave the second object without a reference.
☒ **A, B,** and **D** are incorrect because the first object has two references; changing one of the references will not cause the first object to become unreachable.

15. ☑ **E** is correct. A copy of a reference to the line 13 object is passed to the `doStuff2()` method. We don't know what goes on in that method; it's possible that the reference is passed to other live objects.
☒ **A, B, C,** and **D** are incorrect based on the logic described above.