7

Generics and Collections

CERTIFICATION OBJECTIVES

- Design Using Collections
- Override equals() and hashCode(), Distinguish == and equals()
- Use Generic Versions of Collections Including Set, List, and Map
- Use Type Parameters, Write Generics methods
- Use java.util to Sort and Search Use Comparable and Comparator

✓ Two-Minute Drill

Q&A Self Test

Generics are possibly the most talked about feature of Java 5. Some people love 'em, some people hate 'em, but they're here to stay. At their simplest, they can help make code easier to write, and more robust. At their most complex, they can be very, very hard to create, and maintain. Luckily, the exam creators stuck to the simple end of generics, covering the most common and useful features, and leaving out most of the especially tricky bits. Coverage of collections in this exam has expanded in two ways from the previous exam: the use of generics in collections, and the ability to sort and search through collections.

CERTIFICATION OBJECTIVE

Overriding hashCode() and equals() (Objective 6.2)

6.2 Distinguish between correct and incorrect overrides of corresponding hashCode and equals methods, and explain the difference between == and the equals method.

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 an *object*, but an Object with a capital O. Every exception, every event, every array extends from java.lang.Object. 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 was covered in Chapter 3. So in this section we'll look at just the `hashCode()` and `equals()` methods. Oh, that leaves out `toString()`, doesn't it. Okay, we'll cover that right now because it takes two seconds.

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 shown in the following example:

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 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 subject.
`String toString()`	Returns a "text representation" of the object.

```
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.

Trying to read this output might motivate 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). That's it for `toString()`. Now we'll tackle `equals()` and `hashCode()`.

Overriding equals()

You learned about the `equals()` method in earlier chapters, where we looked at the wrapper classes. We discussed how comparing two object references using the == operator evaluates to `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 isn't 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 a class's `equals()` method, you won't be able to use those objects as a key in a hashtable and you probably won't get accurate Sets, such that there are no conceptual duplicates.

The `equals()` method in class 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 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

Let's say you decide to override `equals()` in your class. It might look 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.

First, 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. For example:

```
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.

Second, compare the attributes we care about (in this case, just `moofValue`). Only the developer can decide what makes two instances equal. (For best performance, you're going to want to check the fewest number of attributes.)

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;
}
```

So that takes care of `equals()`…

Whoa…not so fast. If you look at the Object class in the Java API spec, 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, your code may still compile and run, but your code (or someone else's) may break at runtime in some unexpected way.

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) { } }
```

exam Watch

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()`.

The equals() Contract

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

- It is **reflexive**. For any reference value `x`, `x.equals(x)` should return `true`.
- It is **symmetric**. For any reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- It is **transitive**. For any reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` must return `true`.
- It is **consistent**. For any reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in equals comparisons on the object is modified.
- For any non-`null` reference value `x`, `x.equals(null)` should return `false`.

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()

Hashcodes are typically used to increase the performance of large collections of data. 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 how 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.

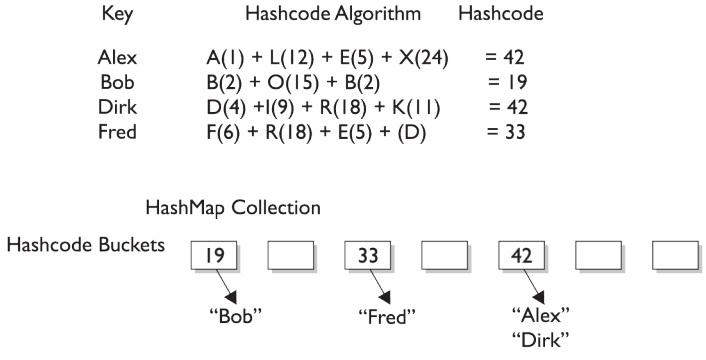
Understanding Hashcodes

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, and so on, and adding the numeric values of all the letters in the name together. A given name will always result in the same code; see Figure 7-1.

FIGURE 7-1

A simplified hashcode example



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 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 hashcode tells you only which bucket to go into, but not how to locate the name once we're in that bucket.

exam Watch

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

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

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 that 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); }
}
```

This `equals()` method says two objects are equal if they have the same `x` value, so objects with the same `x` value will have to 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 improving 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) a class's instance variables (in other words, twiddling their bits), 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. As you know, 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:

1. Give an object some state (assign values to its instance variables).
2. Put the object in a HashMap, using the object as a key.
3. Save the object to a file using serialization without altering any of its state.
4. Retrieve the object from the file through deserialization.
5. 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 to determine object equality.

Bottom line: `transient` variables can really mess with your `equals()` and `hashCode()` implementations. Keep variables non-`transient` or, if they must be marked `transient`, don't use then to determine hashcodes or equality.

CERTIFICATION OBJECTIVE

Collections (Exam Objective 6.1)

6.1 Given a design scenario, determine which collection classes and/or interfaces should be used to properly implement that design, including the use of the Comparable interface.

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, it's almost too painful to consider.

The Collections Framework in Java, which took shape with the release of JDK 1.2 and was expanded in 1.4 and again in Java 5, gives you lists, sets, maps, and queues 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—reasonable 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'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 seven:

`Collection`	`Set`	`SortedSet`
`List`	`Map`	`SortedMap`
`Queue`		

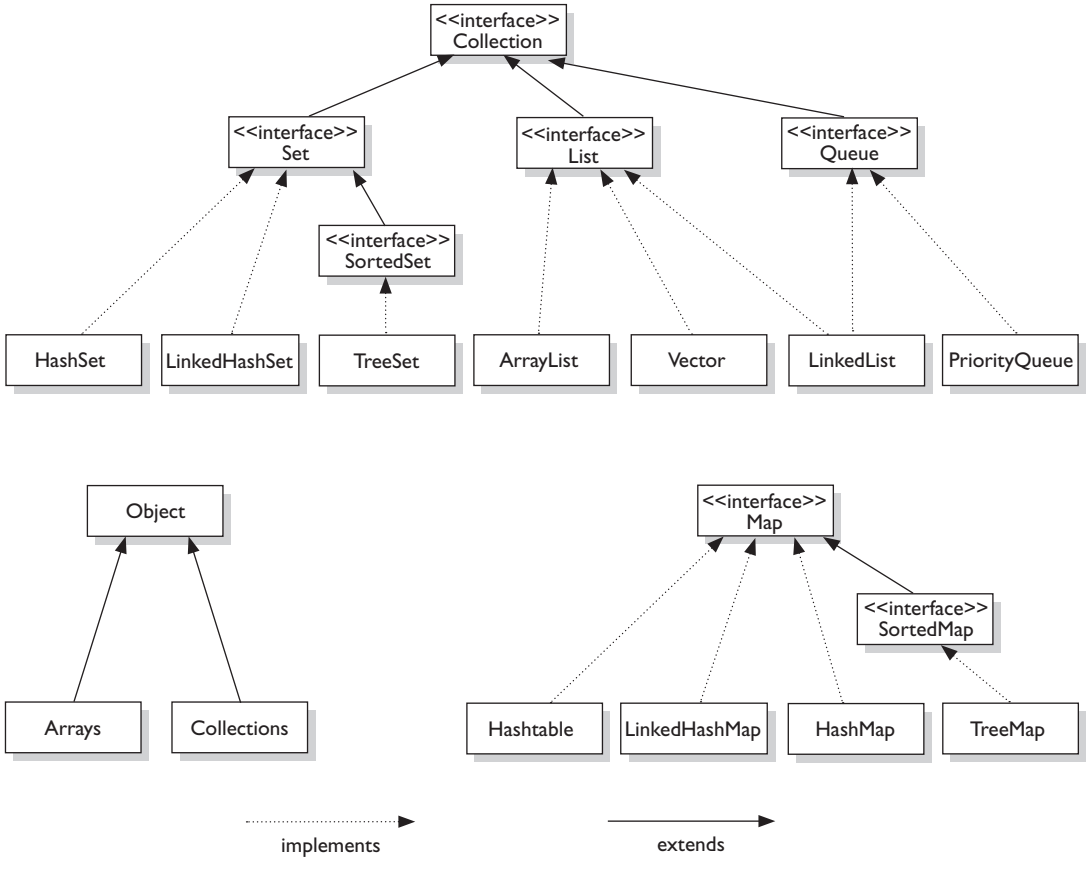
The core concrete implementation classes you need to know for the exam are the following 13 (there are others, but the exam doesn't specifically cover them):

Maps	Sets	Lists	Queues	Utilities
`HashMap`	`HashSet`	`ArrayList`	`PriorityQueue`	`Collections`
`Hashtable`	`LinkedHashSet`	`Vector`		`Arrays`
`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 (see Figure 7-2). 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.
- Collection (capital C), which is actually the java.util.Collection interface from which Set, List, and Queue extend. (That's right, extend, not implement. There are no direct implementations of Collection.)
- Collections (capital C and ends with *s*) is the java.util.Collections class that holds a pile of `static` utility methods for use with collections.

FIGURE 7-2 The interface and class hierarchy for 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()`.

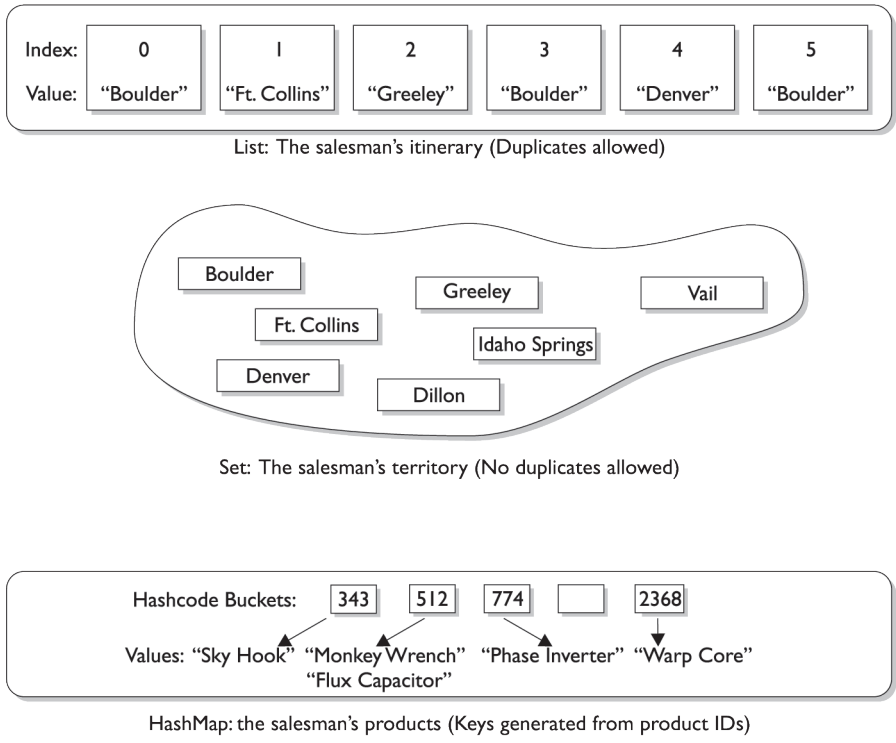
Collections come in four basic flavors:

- **Lists** *Lists* of things (classes that implement List).
- **Sets** *Unique* things (classes that implement Set).
- **Maps** Things with a *unique* ID (classes that implement Map).
- **Queues** Things arranged by the order in which they are to be processed.

Figure 7-3 illustrates the structure of a List, a Set, and a Map.

FIGURE 7-3

The structure of a List, a Set, and a Map



But there are sub-flavors within those four flavors of collections:

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, 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 should 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], and so on). 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 (seemingly) 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 apparently random ways as the collection changes.

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 A *sorted* collection means that the order in the collection is determined according to some rule or rules, known as the sort order. A sort order has nothing to do with when an object was added to the collection, or when was the last time it was accessed, or what "position" it was added at. Sorting is done based on properties of the objects themselves. You put objects into the collection, and the collection will figure out what order to put them in, based on the sort order. A collection that keeps an order (such as any List, which uses insertion order) is not really considered *sorted* unless it sorts using some king of sort order. Most commonly, the sort order used is something called the *natural* order. What does that mean?

You know how to sort alphabetically—A comes before B, F comes before G, and so on. For a collection of String objects, then, the natural order is alphabetical. For Integer objects, the natural order is by numeric value—1 before 2, and so on. 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 (*Comparable*)that defines how instances of a class can be compared to one another (does instance `a` come before `b`, or does instance `b` before `a`?). If the developer decides that Foo objects should be compared using the value of some instance variable (let's say there's one called `bar`), 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. Of course, the Foo class might also inherit a natural order from a superclass rather than define its own order, in some cases.

Aside from natural order as specified by the Comparable interface, it's also possible to define other, different sort orders using another interface: *Comparator*. We will discuss how to use both Comparable and Comparator to define sort orders later in this chapter. But for now, just keep in mind that sort order (including natural order) is not the same as ordering by insertion, access, or index.

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

List Interface

A List cares about the index. The one thing that List has that non-lists don't have is a set of methods related to the index. Those key methods include things like `get(int index)`, `indexOf(Object o)`, `add(int index, Object obj)`, and so on. 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 sections.

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 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. As of Java 5, the LinkedList class has been enhanced to implement the java.util.Queue interface. As such, it now supports the common queue methods: `peek()`, `poll()`, and `offer()`.

Set Interface

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 through a HashSet the order is unpredictable, while a LinkedHashSet lets you iterate through the elements in the order in which they were inserted.

exam Watch

When using HashSet or LinkedHashSet, the objects you add to them must override `hashCode()`***. If they don't override*** `hashCode()`***, the default*** `Object.hashCode()` ***method will allow multiple objects that you might consider "meaningfully equal" to be added to your "no duplicates allowed" set.***

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 natural order. Optionally, you can construct a TreeSet with a constructor that lets you give the collection your own rules for what the order should be (rather than relying on the ordering defined by the elements' class) by using a Comparable or Comparator.

Map Interface

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. 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. Like Sets, Maps rely on the `equals()` method to determine whether two keys are the same or different.

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. 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. HashMap allows one `null` key and multiple `null` values in a collection.

Hashtable Like Vector, Hashtable has existed 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.

TreeMap You can probably guess by now that a TreeMap is a sorted Map. And you already know that by default, this means "sorted by the natural order of the elements." Like TreeSet, TreeMap lets you define a custom sort order (via a Comparable or Comparator) when you construct a TreeMap, that specifies how the elements should be compared to one another when they're being ordered.

Queue Interface

A Queue is designed to hold a list of "to-dos," or things to be processed in some way. Although other orders are possible, queues are typically thought of as FIFO (first-in, first-out). Queues support all of the standard Collection methods and they also add methods to add and subtract elements and review queue elements.

PriorityQueue This class is new with Java 5. Since the LinkedList class has been enhanced to implement the Queue interface, basic queues can be handled with a LinkedList. The purpose of a PriorityQueue is to create a "priority-in, priority out" queue as opposed to a typical FIFO queue. A PriorityQueue's elements are ordered either by natural ordering (in which case the elements that are sorted first will be accessed first) or according to a Comparator. In either case, the elements' ordering represents their relative priority.

exam Watch

You can easily eliminate some answers right away if you recognize that, for example, a Map can't be the class to 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 a class, don't choose an interface type.

Table 7-2 summarizes the 11 of the 13 concrete collection-oriented classes you'll need to understand for the exam. (Arrays and Collections are coming right up!)

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	No
`ArrayList`			x	By index	No
`Vector`			x	By index	No
`LinkedList`			x	By index	No
`PriorityQueue`				Sorted	By to-do order

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 number 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.

CERTIFICATION OBJECTIVE

Using the Collections Framework (Objective 6.5)

6.5 Use capabilities in the java.util package to write code to manipulate a list by sorting, performing a binary search, or converting the list to an array. Use capabilities in the java.util package to write code to manipulate an array by sorting, performing a binary search, or converting the array to a list. Use the java.util.Comparator and java.lang.Comparable interfaces to affect the sorting of lists and arrays. Furthermore, recognize the effect of the "natural ordering" of primitive wrapper classes and java.lang.String on sorting.

We've taken a high-level, theoretical look at the key interfaces and classes in the Collections Framework, now let's see how they work in practice.

ArrayList Basics

The `java.util.ArrayList` class is one of the most commonly used of all the classes in the Collections Framework. It's like an array on vitamins. Some of the advantages `ArrayList` has over arrays are

- It can grow dynamically.
- It provides more powerful insertion and search mechanisms than arrays.

Let's take a look at using an `ArrayList` that contains Strings. A key design goal of the Collections Framework was to provide rich functionality at the level of the main interfaces: List, Set, and Map. In practice, you'll typically want to instantiate an `ArrayList` polymorphically like this:

```
List myList = new ArrayList();
```

As of Java 5 you'll want to say

```
List<String> myList = new ArrayList<String>();
```

This kind of declaration follows the object oriented programming principle of "coding to an interface", and it makes use of generics. We'll say lots more about generics later in this chapter, but for now just know that, as of Java 5, the <String> syntax is the way that you declare a collection's type. (Prior to Java 5 there was no way to specify the type of a collection, and when we cover generics, we'll talk about the implications of mixing Java 5 (typed) and pre-Java 5 (untyped) collections.)

In many ways, `ArrayList<String>` is similar to a `String[]` in that it declares a container that can hold only Strings, but it's more powerful than a `String[]`. Let's look at some of the capabilities that an `ArrayList` has:

```
import java.util.*;
public class TestArrayList {
  public static void main(String[] args) {
    List<String> test = new ArrayList<String>();
    String s = "hi";
    test.add("string");
    test.add(s);
    test.add(s+s);
    System.out.println(test.size());
    System.out.println(test.contains(42));
```

```
        System.out.println(test.contains("hihi"));
        test.remove("hi");
        System.out.println(test.size());
    } }
```

which produces:

```
3
false
true
2
```

There's lots going on in this small program. Notice that when we declared the `ArrayList` we didn't give it a size. Then we were able to ask the `ArrayList` for its size, we were able to ask it whether it contained specific objects, we removed an object right out from the middle of it, and then we re-checked its size.

Autoboxing with Collections

In general, collections can hold Objects but not primitives. Prior to Java 5, a very common use for the wrapper classes was to provide a way to get a primitive into a collection. Prior to Java 5, you had to wrap a primitive by hand before you could put it into a collection. With Java 5, primitives still have to be wrapped, but autoboxing takes care of it for you.

```
List myInts = new ArrayList();   // pre Java 5 declaration
myInts.add(new Integer(42));     // had to wrap an int
```

As of Java 5 we can say

```
myInts.add(42);                  // autoboxing handles it!
```

In this last example, we are still adding an `Integer` object to `myInts` (not an `int` primitive); it's just that autoboxing handles the wrapping for us.

Sorting Collections and Arrays

Sorting and searching topics have been added to the exam for Java 5. Both collections and arrays can be sorted and searched using methods in the API.

Sorting Collections

Let's start with something simple like sorting an ArrayList of Strings alphabetically. What could be easier? Okay, we'll wait while you go find ArrayList's `sort()` method…got it? Of course, ArrayList doesn't give you any way to sort its contents, but the java.util.Collections class does:

```
import java.util.*;
class TestSort1 {
  public static void main(String[] args) {
    ArrayList<String> stuff = new ArrayList<String>(); // #1
    stuff.add("Denver");
    stuff.add("Boulder");
    stuff.add("Vail");
    stuff.add("Aspen");
    stuff.add("Telluride");
    System.out.println("unsorted " + stuff);
    Collections.sort(stuff);                          // #2
    System.out.println("sorted   " + stuff);
  }
}
```

This produces something like this:

```
unsorted [Denver, Boulder, Vail, Aspen, Telluride]
sorted   [Aspen, Boulder, Denver, Telluride, Vail]
```

Line 1 is declaring an ArrayList of Strings, and line 2 is sorting the ArrayList alphabetically. We'll talk more about the Collections class, along with the Arrays class in a later section, for now let's keep sorting stuff.

Let's imagine we're building the ultimate home-automation application. Today we're focused on the home entertainment center, and more specifically the DVD control center. We've already got the file I/O software in place to read and write data between the `dvdInfo.txt` file and instances of class `DVDInfo`. Here are the key aspects of the class:

```
class DVDInfo {
  String title;
  String genre;
  String leadActor;
  DVDInfo(String t, String g, String a) {
```

```
    title = t;  genre = g;  leadActor = a;
  }
  public String toString() {
    return title + " " + genre + " " + leadActor + "\n";
  }
  // getters and setter go here
}
```

Here's the DVD data that's in the `dvdinfo.txt` file:

```
Donnie Darko/sci-fi/Gyllenhall, Jake
Raiders of the Lost Ark/action/Ford, Harrison
2001/sci-fi/??
Caddy Shack/comedy/Murray, Bill
Star Wars/sci-fi/Ford, Harrison
Lost in Translation/comedy/Murray, Bill
Patriot Games/action/Ford, Harrison
```

In our home-automation application, we want to create an instance of `DVDInfo` for each line of data we read in from the `dvdinfo.txt` file. For each instance, we will parse the line of data (remember `String.split()`?) and populate `DVDInfo`'s three instance variables. Finally, we want to put all of the `DVDInfo` instances into an ArrayList. Imagine that the `populateList()` method (below) does all of this. Here is a small piece of code from our application:

```
ArrayList<DVDInfo> dvdList = new ArrayList<DVDInfo>();
populateList();   // adds the file data to the ArrayList
System.out.println(dvdList);
```

You might get output like this:

```
[Donnie Darko sci-fi Gyllenhall, Jake
, Raiders of the Lost Ark action Ford, Harrison
, 2001 sci-fi ??
, Caddy Shack comedy Murray, Bill
, Star Wars sci-fi Ford, Harrison
, Lost in Translation comedy Murray, Bill
, Patriot Games action Ford, Harrison
]
```

(Note: We overrode `DVDInfo`'s `toString()` method, so when we invoked `println()` on the ArrayList it invoked `toString()` for each instance.)

Now that we've got a populated ArrayList, let's sort it:

```
Collections.sort(dvdlist);
```

Oops!, you get something like this:

```
TestDVD.java:13: cannot find symbol
symbol  : method sort(java.util.ArrayList<DVDInfo>)
location: class java.util.Collections
    Collections.sort(dvdlist);
```

What's going on here? We know that the Collections class has a `sort()` method, yet this error implies that Collections does NOT have a `sort()` method that can take a `dvdlist`. That means there must be something wrong with the argument we're passing (`dvdinfo`).

If you've already figured out the problem, our guess is that you did it without the help of the obscure error message shown above…How the heck do you sort instances of `DVDInfo`? Why were we able to sort instances of String? When you look up `Collections.sort()` in the API your first reaction might be to panic. Hang tight, once again the generics section will help you read that weird looking method signature. If you read the description of the one-arg `sort()` method, you'll see that the `sort()` method takes a List argument, and that the objects in the List must implement an interface called Comparable. It turns out that String implements Comparable, and that's why we were able to sort a list of Strings using the `Collections.sort()` method.

The Comparable Interface

The `Comparable` interface is used by the `Collections.sort()` method and the `java.utils.Arrays.sort()` method to sort `Lists` and arrays of objects, respectively. To implement `Comparable`, a class must implement a single method, `compareTo()`. Here's an invocation of `compareTo()`:

```
int x = thisObject.compareTo(anotherObject);
```

The `compareTo()` method returns an int with the following characteristics:

- negative — If `thisObject < anotherObject`
- zero — If `thisObject == anotherObject`
- positive — If `thisObject > anotherObject`

The `sort()` method uses `compareTo()` to determine how the List or object array should be sorted. Since you get to implement `compareTo()` for your own classes, you can use whatever weird criteria you prefer, to sort instances of your classes. Returning to our earlier example for class DVDInfo, we can take the easy way out and use the String class's implementation of `compareTo()`:

```
class DVDInfo implements Comparable<DVDInfo> {    // #1
  // existing code
  public int compareTo(DVDInfo d) {
    return title.compareTo(d.getTitle());         // #2
} }
```

In line 1 we declare that class `DVDInfo` implements `Comparable` in such a way that `DVDInfo` objects can be compared to other DVDInfo objects. In line 2 we implement `compareTo()` by comparing the two DVDInfo object's titles. Since we know that the titles are Strings, and that String implements Comparable, this is an easy way to sort our DVDInfo objects, by title. Before generics came along in Java 5, you would have had to implement Comparable something like this:

```
class DVDInfo implements Comparable {
  // existing code
  public int compareTo(Object o) {   // takes an Object rather
                                     // than a specific type
    DVDInfo d = (DVDInfo)o;
    return title.compareTo(d.getTitle());
} }
```

This is still legal, but you can see that it's both painful and risky, because you have to do a cast, and you need to verify that the cast will not fail before you try it.

exam Watch

It's important to remember that when you override *`equals()`* **you MUST take an argument of type** *`Object`*, **but that when you override** *`compareTo()`* **you should take an argument of the type you're sorting.**

Putting it all together, our DVDInfo class should now look like this:

```
class DVDInfo implements Comparable<DVDInfo> {
  String title;
  String genre;
  String leadActor;
  DVDInfo(String t, String g, String a) {
    title = t;  genre = g;  leadActor = a;
  }
  public String toString() {
    return title + " " + genre + " " + leadActor + "\n";
  }
  public int compareTo(DVDInfo d) {
    return title.compareTo(d.getTitle());
  }
  public String getTitle() {
    return title;
  }
  // other getters and setters
}
```

Now, when we invoke `Collections.sort(dvdlist);` we get

```
[2001 sci-fi ??
, Caddy Shack comedy Murray, Bill
, Donnie Darko sci-fi Gyllenhall, Jake
, Lost in Translation comedy Murray, Bill
, Patriot Games action Ford, Harrison
, Raiders of the Lost Ark action Ford, Harrison
, Star Wars sci-fi Ford, Harrison
]
```

Hooray! Our ArrayList has been sorted by title. Of course, if we want our home automation system to really rock, we'll probably want to sort DVD collections in lots of different ways. Since we sorted our ArrayList by implementing the `compareTo()` method, we seem to be stuck. We can only implement `compareTo()` once in a class, so how do we go about sorting our classes in an order different than what we specify in our `compareTo()` method? Good question. As luck would have it, the answer is coming up next.

Sorting with Comparator

While you were looking up the `Collections.sort()` method you might have noticed that there is an overloaded version of `sort()` that takes a List, AND something called a *Comparator*. The Comparator interface gives you the capability to sort a given collection any number of different ways. The other handy thing about the Comparator interface is that you can use it to sort instances of any class—even classes you can't modify—unlike the Comparable interface, which forces you to change the class whose instances you want to sort. The Comparator interface is also very easy to implement, having only one method, `compare()`. Here's a small class that can be used to sort a List of DVDInfo instances, by `genre`.

```
import java.util.*;
class GenreSort implements Comparator<DVDInfo> {
  public int compare(DVDInfo one, DVDInfo two) {
    return one.getGenre().compareTo(two.getGenre());
  }
}
```

The `Comparator.compare()` method returns an `int` whose meaning is the same as the `Comparable.compareTo()` method's return value. In this case we're taking advantage of that by asking `compareTo()` to do the actual comparison work for us. Here's a test program that lets us test both our Comparable code and our new Comparator code:

```
import java.util.*;
import java.io.*;          // populateList() needs this
public class TestDVD {
  ArrayList<DVDInfo> dvdlist = new ArrayList<DVDInfo>();
  public static void main(String[] args) {
    new TestDVD().go();
  }
  public void go() {
    populateList();
    System.out.println(dvdlist);     // output as read from file
    Collections.sort(dvdlist);
    System.out.println(dvdlist);     // output sorted by title

    GenreSort gs = new GenreSort();
    Collections.sort(dvdlist, gs);
    System.out.println(dvdlist);     // output sorted by genre
  }
```

```
  public void populateList() {
     // read the file, create DVDInfo instances, and
     // populate the ArrayList dvdlist with these instances
  }
}
```

You've already seen the first two output lists, here's the third:

```
[Patriot Games action Ford, Harrison
, Raiders of the Lost Ark action Ford, Harrison
, Caddy Shack comedy Murray, Bill
, Lost in Translation comedy Murray, Bill
, 2001 sci-fi ??
, Donnie Darko sci-fi Gyllenhall, Jake
, Star Wars sci-fi Ford, Harrison
]
```

Because the Comparable and Comparator interfaces are so similar, expect the exam to try to confuse you. For instance you might be asked to implement the `compareTo()` method in the Comparator interface. Study Table 7-3 to burn in the differences between these two interfaces.

TABLE 7-3 Comparing Comparable to Comparator

java.lang.Comparable	java.util.Comparator
`int objOne.compareTo(objTwo)`	`int compare(objOne, objTwo)`
Returns negative if `objOne < objTwo` zero if `objOne == objTwo` positive if `objOne > objTwo`	Same as Comparable
You must modify the class whose instances you want to sort.	You build a class separate from the class whose instances you want to sort.
Only **one** sort sequence can be created	**Many** sort sequences can be created
Implemented frequently in the API by: String, Wrapper classes, Date, Calendar...	Meant to be implemented to sort instances of third-party classes.

Sorting with the Arrays Class

We've been using the java.util.Collections class to sort collections; now let's look at using the java.util.Arrays class to sort arrays. The good news is that sorting arrays of objects is just like sorting collections of objects. The `Arrays.sort()` method is overridden in the same way the `Collections.sort()` method is.

- `Arrays.sort(arrayToSort)`
- `Arrays.sort(arrayToSort, Comparator)`

In addition, the `Arrays.sort()` method is overloaded about a million times to provide a couple of sort methods for every type of primitive. The `Arrays.sort()` methods that sort primitives always sort based on natural order. Don't be fooled by an exam question that tries to sort a primitive array using a Comparator.

Finally, remember that the `sort()` methods for both the Collections class and the Arrays class are `static` methods, and that they alter the objects they are sorting, instead of returning a different sorted object.

exam Watch

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

Searching Arrays and Collections

The Collections class and the Arrays class both provide methods that allow you to search for a specific element. When searching through collections or arrays, the following rules apply:

- Searches are performed using the `binarySearch()` method.
- Successful searches return the `int` index of the element being searched.
- Unsuccessful searches return an `int` index that represents the *insertion point*. The insertion point is the place in the collection/array where the element would be inserted to keep the collection/array properly sorted. Because posi-

tive return values and `0` indicate successful searches, the `binarySearch()` method uses negative numbers to indicate insertion points. Since `0` is a valid result for a successful search, the first available insertion point is `-1`. Therefore, the actual insertion point is represented as (-(insertion point) `-1`). For instance, if the insertion point of a search is at element `2`, the actual insertion point returned will be `-3`.

- The collection/array being searched must be sorted before you can search it.
- If you attempt to search an array or collection that has not already been sorted, the results of the search will not be predictable.
- If the collection/array you want to search was sorted in natural order, it *must* be searched in natural order. (This is accomplished by NOT sending a Comparator as an argument to the `binarySearch()` method.)
- If the collection/array you want to search was sorted using a Comparator, it *must* be searched using the same Comparator, which is passed as the second argument to the `binarySearch()` method. Remember that Comparators cannot be used when searching arrays of primitives.

Let's take a look at a code sample that exercises the `binarySearch()` method:

```
import java.util.*;
class SearchObjArray {
  public static void main(String [] args) {
    String [] sa = {"one", "two", "three", "four"};

    Arrays.sort(sa);                                          // #1
    for(String s : sa)
      System.out.print(s + " ");
    System.out.println("\none = "
                       + Arrays.binarySearch(sa,"one"));  // #2

    System.out.println("now reverse sort");
    ReSortComparator rs = new ReSortComparator();          // #3
    Arrays.sort(sa,rs);
    for(String s : sa)
      System.out.print(s + " ");
    System.out.println("\none = "
                       + Arrays.binarySearch(sa,"one"));  // #4
    System.out.println("one = "
                    + Arrays.binarySearch(sa,"one",rs));  // #5
  }
```

```
  static class ReSortComparator
                  implements Comparator<String> {          // #6
    public int compare(String a, String b) {
      return b.compareTo(a);                               // #7
    }
  }
}
```

which produces something like this:

```
four one three two
one = 1
now reverse sort
two three one four
one = -1
one = 2
```

Here's what happened:

Line 1 Sort the `sa` array, alphabetically (the natural order).
Line 2 Search for the location of element "one", which is 1.
Line 3 Make a Comparator instance. On the next line we re-sort the array using the Comparator.
Line 4 Attempt to search the array. We didn't pass the `binarySearch()` method the Comparator we used to sort the array, so we got an incorrect (undefined) answer.
Line 5 Search again, passing the Comparator to `binarySearch()`. This time we get the correct answer, 2
Line 6 We define the Comparator; it's okay for this to be an inner class.
Line 7 By switching the use of the arguments in the invocation of `compareTo()`, we get an inverted sort.

exam Watch

When solving searching and sorting questions, two big gotchas are:

1. ***Searching an array or collection that hasn't been sorted.***
2. ***Using a Comparator in either the sort or the search, but not both.***

Converting Arrays to Lists to Arrays

There are a couple of methods that allow you to convert arrays to Lists, and Lists to arrays. The List and Set classes have `toArray()` methods, and the Arrays class has a method called `asList()`.

The `Arrays.asList()` method copies an array into a List. The API says, "Returns a fixed-size list backed by the specified array. (Changes to the returned list 'write through' to the array.)" When you use the `asList()` method, the array and the List become joined at the hip. When you update one of them, the other gets updated automatically. Let's take a look:

```
String[] sa = {"one", "two", "three", "four"};
List sList = Arrays.asList(sa);                 // make a List
System.out.println("size  " + sList.size());
System.out.println("idx2  " + sList.get(2));

sList.set(3,"six");                             // change List
sa[1] = "five";                                 // change array
for(String s : sa)
  System.out.print(s + " ");
System.out.println("\nsl[1] " + sList.get(1));
```

This produces

```
size 4
idx2 three
one five three six
sl[1] five
```

Notice that when we print the final state of the array and the List, they have both been updated with each other's changes. Wouldn't something like this behavior make a great exam question?

Now let's take a look at the `toArray()` method. There's nothing too fancy going on with the `toArray()` method; it comes in two flavors: one that returns a new Object array, and one that uses the array you send it as the destination array:

```
List<Integer> iL = new ArrayList<Integer>();
for(int x=0; x<3; x++)
  iL.add(x);
Object[] oa = iL.toArray();          // create an Object array
Integer[] ia2 = new Integer[3];
ia2 = iL.toArray(ia2);               // create an Integer array
```

Using Lists

Remember that Lists are usually used to keep things in some kind of order. You can use a LinkedList to create a first-in, first-out queue. You can use an ArrayList to keep track of what locations were visited, and in what order. Notice that in both of these examples it's perfectly reasonable to assume that duplicates might occur. In addition, Lists allow you to manually override the ordering of elements by adding or removing elements via the element's index. Before Java 5, and the enhanced `for` loop, the most common way to examine a List "element by element" was by the use of an Iterator. You'll still find Iterators in use in the Java code you encounter, and you might just find an Iterator or two on the exam. An Iterator is an object that's associated with a specific collection. It let's you loop through the collection step by step. The two Iterator methods you need to understand for the exam are

- **`boolean hasNext()`** Returns `true` if there is at least one more element in the collection being traversed. Invoking `hasNext()` does NOT move you to the next element of the collection.
- **`object next()`** This method returns the next object in the collection, AND moves you forward to the element after the element just returned.

Let's look at a little code that uses a List and an Iterator:

```
import java.util.*;
class Dog {
  public String name;
  Dog(String n) { name = n; }
}
class ItTest {
  public static void main(String[] args) {
    List<Dog> d = new ArrayList<Dog>();
    Dog dog = new Dog("aiko");
    d.add(dog);
    d.add(new Dog("clover"));
    d.add(new Dog("magnolia"));
    Iterator<Dog> i3 = d.iterator();  // make an iterator
    while (i3.hasNext()) {
      Dog d2 = i3.next();             // cast not required
      System.out.println(d2.name);
    }
    System.out.println("size " + d.size());
    System.out.println("get1 " + d.get(1).name);
    System.out.println("aiko " + d.indexOf(dog));
```

```
    d.remove(2);
    Object[] oa = d.toArray();
    for(Object o : oa) {
      Dog d2 = (Dog)o;
      System.out.println("oa " + d2.name);
    }
  }
}
```

This produces

```
aiko
clover
magnolia
size 3
get1 clover
aiko 0
oa aiko
oa clover
```

First off, we used generics syntax to create the Iterator (an Iterator of type Dog). Because of this, when we used the `next()` method, we didn't have to cast the Object returned by `next()` to a Dog. We could have declared the Iterator like this:

```
Iterator i3 = d.iterator();  // make an iterator
```

But then we would have had to cast the returned value:

```
Dog d2 = (Dog)i3.next();
```

The rest of the code demonstrates using the `size()`, `get()`, `indexOf()`, and toArray() methods. There shouldn't be any surprises with these methods. In a few pages Table 7-5 will list all of the List, Set, and Map methods you should be familiar with for the exam. As a last warning, remember that List is an interface!

Using Sets

Remember that Sets are used when you don't want any duplicates in your collection. If you attempt to add an element to a set that already exists in the set, the duplicate element will not be added, and the `add()` method will return `false`. Remember, HashSets tend to be very fast because, as we discussed earlier, they use hashcodes.

You can also create a TreeSet, which is a Set whose elements are sorted. You must use caution when using a TreeSet (we're about to explain why):

```
import java.util.*;
class SetTest {
  public static void main(String[] args) {
    boolean[] ba = new boolean[5];
    // insert code here

    ba[0] = s.add("a");
    ba[1] = s.add(new Integer(42));
    ba[2] = s.add("b");
    ba[3] = s.add("a");
    ba[4] = s.add(new Object());
    for(int x=0; x<ba.length; x++)
      System.out.print(ba[x] + " ");
    System.out.println("\n");
    for(Object o : s)
      System.out.print(o + " ");
  }
}
```

If you insert the following line of code you'll get output something like this:

```
Set s = new HashSet();       // insert this code

true true true false true
a java.lang.Object@e09713 42 b
```

It's important to know that the order of objects printed in the second `for` loop is not predictable: HashSets and LinkedHashSets do not guarantee any ordering. Also, notice that the fourth invocation of `add()` failed, because it attempted to insert a duplicate entry (a String with the value `a`) into the Set.

If you insert this line of code you'll get something like this:

```
Set s = new TreeSet();        // insert this code

Exception in thread "main" java.lang.ClassCastException: java.
lang.String
        at java.lang.Integer.compareTo(Integer.java:35)
        at java.util.TreeMap.compare(TreeMap.java:1093)
```

```
        at java.util.TreeMap.put(TreeMap.java:465)
        at java.util.TreeSet.add(TreeSet.java:210)
```

The issue is that whenever you want a collection to be sorted, its elements must be mutually comparable. Remember that unless otherwise specified, objects of different types are not mutually comparable.

Using Maps

Remember that when you use a class that implements Map, any classes that you use as a part of the keys for that map must override the `hashCode()` and `equals()` methods. (Well, you only have to override them if you're interested in retrieving stuff from your Map. Seriously, it's legal to use a class that doesn't override `equals()` and `hashCode()` as a key in a Map; your code will compile and run, you just won't find your stuff.) Here's some code demonstrating the use of a HashMap:

```
import java.util.*;
class Dog {
  public Dog(String n) { name = n; }
  public String name;
  public boolean equals(Object o) {
    if((o instanceof Dog) &&
       (((Dog)o).name == name)) {
      return true;
    } else {
      return false;
    }
  }
  public int hashCode() {return name.length(); }
}
class Cat { }

enum Pets {DOG, CAT, HORSE }

class MapTest {
  public static void main(String[] args) {
    Map<Object, Object> m = new HashMap<Object, Object>();

    m.put("k1", new Dog("aiko"));   // add some key/value pairs
    m.put("k2", Pets.DOG);
    m.put(Pets.CAT, "CAT key");
    Dog d1 = new Dog("clover");     // let's keep this reference
```

```
    m.put(d1, "Dog key");
    m.put(new Cat(), "Cat key");

    System.out.println(m.get("k1"));                    // #1
    String k2 = "k2";
    System.out.println(m.get(k2));                      // #2
    Pets p = Pets.CAT;
    System.out.println(m.get(p));                       // #3
    System.out.println(m.get(d1));                      // #4
    System.out.println(m.get(new Cat()));               // #5
    System.out.println(m.size());                       // #6
  }
}
```

which produces something like this:

```
Dog@1c
DOG
CAT key
Dog key
null
5
```

Let's review the output. The first value retrieved is a Dog object (your value will vary). The second value retrieved is an enum value (DOG). The third value retrieved is a String; note that the key was an enum value. Pop quiz: What's the implication of the fact that we were able to successfully use an enum as a key?

The implication of this is that enums override `equals()` and `hashCode()`. And, if you look at the java.lang.Enum class in the API, you will see that, in fact, these methods have been overridden.

The fourth output is a String. The important point about this output is that the key used to retrieve the String was made of a Dog object. The fifth output is `null`. The important point here is that the `get()` method failed to find the Cat object that was inserted earlier. (The last line of output confirms that indeed, 5 key/value pairs exist in the Map.) Why didn't we find the `Cat key` String? Why did it work to use an instance of Dog as a key, when using an instance of Cat as a key failed?

It's easy to see that Dog overrode `equals()` and `hashCode()` while Cat didn't.

Let's take a quick look at hashcodes. We used an incredibly simplistic hashcode formula in the Dog class—the hashcode of a Dog object is the length of the instance's `name`. So in this example the hashcode = `4`. Let's compare the following two `hashCode()` methods:

```
public int hashCode() {return name.length(); }   // #1
public int hashCode() {return 4; }               // #2
```

Time for another pop quiz: Are the preceding two hashcodes legal? Will they successfully retrieve objects from a Map? Which will be faster?

The answer to the first two questions is Yes and Yes. Neither of these hashcodes will be very efficient (in fact they would both be incredibly inefficient), but they are both legal, and they will both work. The answer to the last question is that the first hashcode will be a little bit faster than the second hashcode. In general, the more *unique* hashcodes a formula creates, the faster the retrieval will be. The first hashcode formula will generate a different code for each `name` length (for instance the name `Robert` will generate one hashcode and the name `Benchley` will generate a different hashcode). The second hashcode formula will always produce the same result, `4`, so it will be slower than the first.

Our last Map topic is what happens when an object used as a key has its values changed? If we add two lines of code to the end of the earlier `MapTest.main()`,

```
d1.name = "magnolia"
System.out.println(m.get(d1));
```

we get something like this:

```
Dog@4
DOG
CAT key
Dog key
null
5
null
```

The Dog that was previously found now cannot be found. Because the `Dog.name` variable is used to create the hashcode, changing the `name` changed the value of the hashcode. As a final quiz for hashcodes, determine the output for the following lines of code if they're added to the end of `MapTest.main()`:

```
d1.name = "magnolia";
System.out.println(m.get(d1));                  // #1
d1.name = "clover";
System.out.println(m.get(new Dog("clover")));   // #2
d1.name = "arthur";
System.out.println(m.get(new Dog("clover")));   // #3
```

Remember that the hashcode is equal to the length of the `name` variable. When you study a problem like this, it can be useful to think of the two stages of retrieval:

1. Use the `hashcode()` method to find the correct bucket
2. Use the `equals()` method to find the object in the bucket

In the first call to `get()`, the hashcode is 8 (`magnolia`) and it should be 6 (`clover`), so the retrieval fails at step 1 and we get `null`. In the second call to `get()`, the hashcodes are both 6, so step 1 succeeds. Once in the correct bucket (the "length of name = 6" bucket), the `equals()` method is invoked, and since Dog's `equals()` method compares names, `equals()` succeeds, and the output is `Dog key`. In the third invocation of `get()`, the hashcode test succeeds, but the `equals()` test fails because `arthur` is NOT equal to `clover`.

In a few pages Table 7-5 will summarize the Map methods you should be familiar with for the exam.

Using the PriorityQueue Class

The last collection class you'll need to understand for the exam is the PriorityQueue. Unlike basic queue structures that are first-in, first-out by default, a PriorityQueue orders its elements using a user-defined priority. The priority can be as simple as natural ordering (in which, for instance, an entry of `1` would be a higher priority than an entry of `2`). In addition, a PriorityQueue can be ordered using a Comparator, which lets you define any ordering you want. Queues have a few methods not found in other collection interfaces: `peek()`, `poll()`, and `offer()`.

```
import java.util.*;
class PQ {
  static class PQsort
          implements Comparator<Integer> {  // inverse sort
    public int compare(Integer one, Integer two) {
      return two - one;                     // unboxing
    }
  }
  public static void main(String[] args) {
    int[] ia = {1,5,3,7,6,9,8 };            // unordered data
    PriorityQueue<Integer> pq1 =
      new PriorityQueue<Integer>();         // use natural order

    for(int x : ia)                         // load queue
      pq1.offer(x);
    for(int x : ia)                         // review queue
```

```
      System.out.print(pq1.poll() + " ");
    System.out.println("");

    PQsort pqs = new PQsort();              // get a Comparator
    PriorityQueue<Integer> pq2 =
      new PriorityQueue<Integer>(10,pqs);   // use Comparator

    for(int x : ia)                         // load queue
      pq2.offer(x);
    System.out.println("size " + pq2.size());
    System.out.println("peek " + pq2.peek());
    System.out.println("size " + pq2.size());
    System.out.println("poll " + pq2.poll());
    System.out.println("size " + pq2.size());
    for(int x : ia)                         // review queue
      System.out.print(pq2.poll() + " ");
  }
}
```

This code produces something like this:

```
1 3 5 6 7 8 9

size 7
peek 9
size 7
poll 9
size 6
8 7 6 5 3 1 null
```

Let's look at this in detail. The first `for` loop iterates through the `ia` array, and uses the `offer()` method to add elements to the PriorityQueue named `pq1`. The second `for` loop iterates through `pq1` using the `poll()` method, which returns the highest priority entry in `pq1` AND removes the entry from the queue. Notice that the elements are returned in priority order (in this case, natural order). Next, we create a Comparator—in this case, a Comparator that orders elements in the opposite of natural order. We use this Comparator to build a second PriorityQueue, `pq2`, and we load it with the same array we used earlier. Finally, we check the size of `pq2` before and after calls to `peek()` and `poll()`. This confirms that `peek()` returns the highest priority element in the queue without removing it, and `poll()` returns the highest priority element, AND removes it from the queue. Finally, we review the remaining elements in the queue.

Method Overview for Arrays and Collections

For these two classes, we've already covered the trickier methods you might encounter on the exam. Table 7-4 lists a summary of the methods you should be aware of. (Note: The `T[]` syntax will be explained later in this chapter; for now, think of it as meaning "any array that's NOT an array of primitives.")

TABLE 7-4 Key Methods in Arrays and Collections

Key Methods in java.util.Arrays	Descriptions
`static List asList(T[])`	Convert an array to a List, (and bind them).
`static int binarySearch(Object[], key)` `static int binarySearch(primitive[], key)`	Search a sorted array for a given value, return an index or insertion point.
`static int binarySearch(T[], key, Comparator)`	Search a Comparator-sorted array for a value.
`static boolean equals(Object[], Object[])` `static boolean equals(primitive[], primitive[])`	Compare two arrays to determine if their contents are equal.
`public static void sort(Object[ ] )` `public static void sort(primitive[ ] )`	Sort the elements of an array by natural order.
`public static void sort(T[], Comparator)`	Sort the elements of an array using a Comparator.
`public static String toString(Object[])` `public static String toString(primitive[])`	Create a String containing the contents of an array.
Key Methods in java.util.Collections	**Descriptions**
`static int binarySearch(List, key)` `static int binarySearch(List, key, Comparator)`	Search a "sorted" List for a given value, return an index or insertion point.
`static void reverse(List)`	Reverse the order of elements in a List.
`static Comparator reverseOrder()` `static Comparator reverseOrder(Comparator)`	Return a Comparator that sorts the reverse of the collection's current sort sequence.
`static void sort(List)` `static void sort(List, Comparator)`	Sort a List either by natural order or by a Comparator.

Method Overview for List, Set, Map, and Queue

For these four interfaces, we've already covered the trickier methods you might encounter on the exam. Table 7-5 lists a summary of the List, Set, and Map methods you should be aware of.

TABLE 7-5 Key Methods in List, Set, and Map

Key Interface Methods	List	Set	Map	Descriptions
`boolean add(element)` `boolean add(index, element)`	X X	X		Add an element. For Lists, optionally add the element at an index point.

`boolean contains(object)`
`boolean containsKey(object key)`
`boolean containsValue(object value)`
	X	X	
			X
			X
				Search a collection for an object (or, optionally for Maps a key), return the result as a `boolean`.

`object get(index)` `object get(key)`	X		 X	Get an object from a collection, via an index or a key.
`int indexOf(object)`	X			Get the location of an object in a List.
`Iterator iterator()`	X	X		Get an Iterator for a List or a Set.
`Set keySet()`			X	Return a Set containing a Map's keys.
`put(key, value)`			X	Add a key/value pair to a Map.

`remove(index)`
`remove(object)`
`remove(key)`
	X
	X
		
		X
			
			X
				Remove an element via an index, or via the element's value, or via a key.

`int size()`	X	X	X	Return the number of elements in a collection.
`Object[] toArray()` `T[] toArray(T[])`	X	X		Return an array containing the elements of the collection.

For the exam, the PriorityQueue methods that are important to understand are `offer()` (which is similar to `add()`), `peek()` (which retrieves the element at the head of the queue, but doesn't delete it), and `poll()` (which retrieves the head element and removes it from the queue).

exam Watch

It's important to know some of the details of natural ordering. The following code will help you understand the relative positions of uppercase characters, lowercase characters, and spaces in a natural ordering:

```
String[] sa = {">ff<", "> f<", ">f <", ">FF<" }; // ordered?
PriorityQueue<String> pq3 = new PriorityQueue<String>();
for(String s : sa)
  pq3.offer(s);
for(String s : sa)
  System.out.print(pq3.poll() + " ");
```

This produces:

```
> f< >FF< >f < >ff<
```

If you remember that spaces sort before characters and that uppercase letters sort before lowercase characters, you should be good to go for the exam.

CERTIFICATION OBJECTIVE

Generic Types (Objectives 6.3 and 6.4)

6.3 Write code that uses the generic versions of the Collections API, in particular the Set, List, and Map interfaces and implementation classes. Recognize the limitations of the non-generic Collections API and how to refactor code to use the generic versions.

6.4 Develop code that makes proper use of type parameters in class/interface declarations, instance variables, method arguments, and return types; and write generic methods or methods that make use of wildcard types and understand the similarities and differences between these two approaches.

Arrays in Java have always been type safe—an array declared as type String (`String []`) can't accept Integers (or `int`s), Dogs, or anything other than Strings. But remember that before Java 5 there was no syntax for declaring a type safe collection. To make an ArrayList of Strings, you said,

```
ArrayList myList = new ArrayList();
```

or, the polymorphic equivalent:

```
List myList = new ArrayList();
```

There was no syntax that let you specify that `myList` will take Strings and only Strings. And with no way to specify a type for the ArrayList, the compiler couldn't enforce that you put only things of the specified type into the list. As of Java 5, we can use generics, and while they aren't only for making type safe collections, that's just about all most developers use generics for. So, while generics aren't just for collections, think of collections as the overwhelming reason and motivation for adding generics to the language.

And it was not an easy decision, nor has it been an entirely welcome addition. Because along with all the nice happy type safety, generics come with a lot of baggage—most of which you'll never see or care about, but there are some gotchas that come up surprisingly quickly. We'll cover the ones most likely to show up in your own code, and those are also the issues that you'll need to know for the exam.

The biggest challenge for Sun in adding generics to the language (and the main reason it took them so long) was how to deal with legacy code built without generics. Sun's Java engineers obviously didn't want to break everyone's existing Java code, so they had to find a way for Java classes with both type safe (generic) and non-type safe (non-generic/pre-Java 5) collections to still work together. Their solution isn't the friendliest, but it does let you use older non-generic code, as well as use generic code that plays with non-generic code. But notice we said "plays," and not "plays WELL."

While you can integrate Java 5 generic code with legacy non-generic code, the consequences can be disastrous, and unfortunately, most of the disasters happen at runtime, not compile time. Fortunately, though, most compilers will generate warnings to tell you when you're using unsafe (meaning non-generic) collections.

The Java 5 exam covers both pre-Java 5 (non-generic) and Java 5 style collections, and you'll see questions that expect you to understand the tricky

problems that can come from mixing non-generic and generic code together. And like some of the other topics in this book, you could fill an entire book if you really wanted to cover every detail about generics, but the exam (and this book) covers more than most developers will ever need to use.

The Legacy Way to Do Collections

Here's a review of a pre-Java 5 ArrayList intended to hold Strings. (We say "intended" because that's about all you had—good intentions—to make sure that the ArrayList would hold only Strings).

```
List myList = new ArrayList();  // can't declare a type

myList.add("Fred");             // OK, it will hold Strings

myList.add(new Dog());          // and it will hold Dogs too

myList.add(new Integer(42));    // and Integers...
```

A non-generic collection can hold any kind of object! A non-generic collection is quite happy to hold anything that is NOT a primitive.

This meant it was entirely up to the programmer to be…careful. Having no way to guarantee collection type wasn't very programmer-friendly for such a strongly typed language. We're so used to the compiler stopping us from, say, assigning an `int` to a `boolean` reference or a String to a Dog reference, but with collections, it was, "Come on in! The door is always open! All objects are welcome here any time!"

And since a collection could hold anything, the methods that get objects out of the collection could have only one kind of return type—java.lang.Object. That meant that getting a String back out of our only-Strings-intended list required a cast:

```
String s = (String) myList.get(0);
```

And since you couldn't guarantee that what was coming out really was a String (since you were allowed to put anything in the list), the cast could fail at runtime.

So, generics takes care of both ends (the putting in and getting out) by enforcing the type of your collections. Let's update the String list:

```
List<String> myList = new ArrayList<String>();
myList.add("Fred");        // OK, it will hold Strings
myList.add(new Dog());     // compiler error!!
```

Perfect. That's exactly what we want. By using generics syntax—which means putting the type in angle brackets <String>, we're telling the compiler that this collection can hold only String objects. The type in angle brackets is referred to as either the "parameterized type," "type parameter," or of course just old-fashioned "type." In this chapter, we'll refer to it both new ways.

So, now that what you put IN is guaranteed, you can also guarantee what comes OUT, and that means you can get rid of the cast when you get something from the collection. Instead of

```
String s = (String)myList.get(0);  // pre-generics, when a
                                   // String wasn't guaranteed
```

we can now just say

```
String s = myList.get(0);
```

The compiler already knows that `myList` contains only things that can be assigned to a String reference, so now there's no need for a cast. So far, it seems pretty simple. And with the new `for` loop, you can of course iterate over the guaranteed-to-be-String list:

```
for (String s : myList) {
   int x = s.length();
   // no need for a cast before calling a String method! The
   // compiler already knew "s" was a String coming from MyList
}
```

And of course you can declare a type parameter for a method argument, which then makes the argument a type safe reference:

```
void takeListOfStrings(List<String> strings) {
    strings.add("foo");  // no problem adding a String
}
```

The method above would NOT compile if we changed it to

```
void takeListOfStrings(List<String> strings) {
   strings.add(new Integer(42)); // NO!! strings is type safe
}
```

Return types can obviously be declared type safe as well:

```
public Set<Dog> getDogList() {
    Set<Dog> dogs = new HashSet<Dog>();
    // more code to insert dogs
    return dogs;
}
```

The compiler will stop you from returning anything not compatible with a Set<Dog> (although what is and is not compatible is going to get very interesting in a minute). And since the compiler guarantees that only a type safe Dog Set is returned, those calling the method won't need a cast to take Dogs from the Set:

```
Dog d = getDogList().get(0);  // we KNOW a Dog is coming out
```

With pre-Java 5, non-generic code, the `getDogList()` method would be

```
public Set getDogList() {
    Set dogs = new HashSet();
     // code to add only Dogs... fingers crossed...
    return dogs; // a Set of ANYTHING will work here
}
```

and the caller would need a cast:

```
Dog d = (Dog) getDogList().get(0);
```

(The cast in this example applies to what comes from the Set's `get()` method; we aren't casting what is returned from the `getDogList()` method, which is a Set.)

But what about the benefit of a completely heterogeneous collection? In other words, what if you liked the fact that before generics you could make an ArrayList that could hold any kind of object?

```
List myList = new ArrayList();  // old-style, non-generic
```

is almost identical to

```
List<Object> myList = new
                 ArrayList<Object>(); // holds ANY object type
```

Declaring a List with a type parameter of <Object> makes a collection that works in almost the same way as the original pre-Java 5, non-generic collection—you can put ANY Object type into the collection. You'll see a little later that non-generic collections and collections of type <Object> aren't entirely the same, but most of the time the differences do not matter.

Oh, if only this were the end of the story...but there are still a few tricky issues with methods arguments, polymorphism, and integrating generic and non-generic code, so we're just getting warmed up here.

Generics and Legacy Code

The easiest generics thing you'll need to know for the exam is how to update non-generic code to make it generic. You just add a type in angle brackets (<>) immediately following the collection type in BOTH the variable declaration and the constructor call, including any place you declare a variable (so that means arguments and return types too). A pre-Java 5 List meant to hold only Integers:

```
List myList = new ArrayList();
```

becomes

```
List<Integer> myList = new ArrayList<Integer>();
```

and a list meant to hold only Strings goes from

```
public List changeStrings(ArrayList s) { }
```

to this:

```
public List<String> changeStrings(ArrayList<String> s) { }
```

Easy. And if there's code that used the earlier non-generic version and performed a cast to get things out, that won't break anyone's code:

```
Integer i = (Integer) list.get(0); // cast no longer needed,
                                   // but it won't hurt
```

Mixing Generic and Non-generic Collections

Now here's where it starts to get interesting…imagine we have an ArrayList, of type Integer, and we're passing it into a method from a class whose source code we don't have access to. Will this work?

```
// a Java 5 class using a generic collection
import java.util.*;
public class TestLegacy {
    public static void main(String[] args) {
        List<Integer> myList = new ArrayList<Integer>();
                                    // type safe collection
        myList.add(4);
        myList.add(6);
        Adder adder = new Adder();
        int total = adder.addAll(myList);
                            // pass it to an untyped argument
        System.out.println(total);
      }
}
```

The older, non-generics class we want to use:

```
import java.util.*;
class Adder {
   int addAll(List list) {
      // method with a non-generic List argument,
      // but assumes (with no guarantee) that it will be Integers
      Iterator it = list.iterator();
      int total = 0;
      while (it.hasNext()) {
        int i = ((Integer)it.next()).intValue();
        total += i;
      }
    return total;
   }
}
```

Yes, this works just fine. You can mix correct generic code with older non-generic code, and everyone is happy.

In the previous example, the `addAll()` legacy method assumed (trusted? hoped?) that the list passed in was indeed restricted to Integers, even though when the code was written, there was no guarantee. It was up to the programmers to be careful.

Since the `addAll()` method wasn't doing anything except getting the Integer (using a cast) from the list and accessing its value, there were no problems. In that example, there was no risk to the caller's code, but the legacy method might have blown up if the list passed in contained anything but Integers (which would cause a ClassCastException).

But now imagine that you call a legacy method that doesn't just *read* a value but *adds* something to the ArrayList? Will this work?

```
import java.util.*;
public class TestBadLegacy {
    public static void main(String[] args) {
        List<Integer> myList = new ArrayList<Integer>();
        myList.add(4);
        myList.add(6);
        Inserter in  = new Inserter();
        in.insert(myList);  // pass List<Integer> to legacy code
      }
}
class Inserter {
     // method with a non-generic List argument
     void insert(List list) {
       list.add(new Integer(42)); // adds to the incoming list
    }
}
```

Sure, this code works. It compiles, and it runs. The `insert()` method puts an Integer into the list that was originally typed as <Integer>, so no problem.

But…what if we modify the `insert()` method like this:

```
void insert(List list) {
   list.add(new String("42"));  // put a String in the list
                                // passed in
}
```

Will that work? Yes, sadly, it does! It both compiles and runs. No runtime exception. Yet, someone just stuffed a String into a *supposedly* type safe ArrayList of type <Integer>. How can that be?

Remember, the older legacy code was allowed to put anything at all (except primitives) into a collection. And in order to support legacy code, Java 5 allows your newer type safe code to make use of older code (the last thing Sun wanted to do was ask several million Java developers to modify all their existing code).

So, the Java 5 compiler is *forced* into letting you compile your new type safe code even though your code invokes a method of an older class that takes a non-type safe argument and does who knows what with it.

However, just because the Java 5 compiler allows this code to compile doesn't mean it has to be HAPPY about it. In fact the compiler will warn you that you're taking a big, big risk sending your nice protected ArrayList<Integer> into a dangerous method that can have its way with your list and put in Floats, Strings, or even Dogs.

When you called the `addAll()` method in the earlier example, it didn't insert anything to the list (it simply added up the values within the collection), so there was no risk to the caller that his list would be modified in some horrible way. It compiled and ran just fine. But in the second version, with the legacy `insert()` method that adds a String, the compiler generated a warning:

```
javac TestBadLegacy.java
Note: TestBadLegacy.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Remember that *compiler warnings are NOT considered a compiler failure*. The compiler generated a perfectly valid class file from the compilation, but it was kind enough to tell you by saying, in so many words, "I seriously hope you know what you are doing because this old code has NO respect (or even knowledge) of your <Integer> typing, and can do whatever the heck it wants to your precious ArrayList<Integer>."

exam Watch

Be sure you know the difference between "compilation fails" and "compiles without error" and "compiles without warnings" and "compiles with warnings." In most questions on the exam, you care only about compiles vs. compilation fails—compiler warnings don't matter for most of the exam. But when you are using generics, and mixing both typed and untyped code, warnings matter.

Back to our example with the legacy code that does an insert, keep in mind that for BOTH versions of the `insert()` method (one that adds an Integer and one that adds a String) the compiler issues warnings. The compiler does NOT know whether

the `insert()` method is adding the right thing (Integer) or wrong thing (String). The reason the compiler produces a warning is because the method is ADDING something to the collection! In other words, the compiler knows there's a chance the method might add the wrong thing to a collection the caller thinks is type safe.

exam Watch

For the purposes of the exam, unless the question includes an answer that mentions warnings, then even if you know compilation will produce warnings, that is still a successful compile! Compiling with warnings is NEVER considered a compilation failure.

One more time—if you see code that you know will compile with warnings, you must NOT choose "Compilation fails." as an answer. The bottom line is this: code that compiles with warnings is still a successful compile. If the exam question wants to test your knowledge of whether code will produce a warning (or what you can do to the code to ELIMINATE warnings), the question (or answer) will explicitly include the word "warnings."

So far, we've looked at how the compiler will generate warnings if it sees that there's a chance your type safe collection could be harmed by older, non-type-safe code. But one of the questions developers often ask is, "Okay, sure, it compiles, but why does it RUN? Why does the code that inserts the wrong thing into my list work at runtime?" In other words, why does the JVM let old code stuff a String into your ArrayList<Integer>, without any problems at all? No exceptions, nothing. Just a quiet, behind-the-scenes, total violation of your type safety that you might not discover until the worst possible moment.

There's one Big Truth you need to know to understand why it runs without problems—the JVM has no idea that your ArrayList was supposed to hold only Integers. The typing information does not exist at runtime! All your generic code is strictly for the compiler. Through a process called "type erasure," the compiler does all of its verifications on your generic code and then strips the type information out of the class bytecode. At runtime, ALL collection code—both legacy and new Java 5 code you write using generics—looks exactly like the pre-generic version of collections. None of your typing information exists at runtime. In other words, even though you WROTE

```
List<Integer> myList = new ArrayList<Integer>();
```

By the time the compiler is done with it, the JVM sees what it always saw before Java 5 and generics:

```
List myList = new ArrayList();
```

The compiler even inserts the casts for you—the casts you had to do to get things out of a pre-Java 5 collection.

Think of generics as strictly a compile-time protection. The compiler uses generic type information (the <type> in the angle brackets) to make sure that your code doesn't put the wrong things into a collection, and that you do not assign what you get from a collection to the wrong reference type. But NONE of this protection exists at runtime.

This is a little different from arrays, which give you BOTH compile-time protection and runtime protection. Why did they do generics this way? Why is there no type information at runtime? To support legacy code. At runtime, collections are collections just like the old days. What you gain from using generics is compile-time protection that guarantees that you won't put the wrong thing into a typed collection, and it also eliminates the need for a cast when you get something out, since the compiler already knows that only an Integer is coming out of an Integer list.

The fact is, you don't NEED runtime protection…until you start mixing up generic and non-generic code, as we did in the previous example. Then you can have disasters at runtime. The only advice we have is to pay very close attention to those compiler warnings:

```
javac TestBadLegacy.java
Note: TestBadLegacy.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

This compiler warning isn't very descriptive, but the second note suggests that you recompile with `-Xlint:unchecked`. If you do, you'll get something like this:

```
javac -Xlint:unchecked TestBadLegacy.java
TestBadLegacy.java:17: warning: [unchecked] unchecked call to
add(E) as a member of the raw type java.util.List
       list.add(new String("42"));
                ^
1 warning
```

When you compile with the `-Xlint:unchecked` flag, the compiler shows you exactly which method(s) might be doing something dangerous. In this example,

since the list argument was not declared with a type, the compiler treats it as legacy code and assumes no risk for what the method puts into the "raw" list.

On the exam, you must be able to recognize when you are compiling code that will produce warnings but still compile. And any code that compiles (even with warnings) will run! No type violations will be caught at runtime by the JVM, *until* those type violations mess with your code in some other way. In other words, the act of adding a String to an <Integer> list won't fail at runtime *until* you try to treat that String-you-think-is-an-Integer as an Integer.

For example, imagine you want your code to pull something out of your *supposedly* type safe ArrayList<Integer> that older code put a String into. It compiles (with warnings). It runs...or at least the code that actually adds the String to the list runs. But when you take the String-that-wasn't-supposed-to-be-there out of the list, and try to assign it to an Integer reference or invoke an Integer method, you're dead.

Keep in mind, then, that the problem of putting the wrong thing into a typed (generic) collection does not show up at the time you actually do the `add()` to the collection. It only shows up later, when you try to use something in the list and it doesn't match what you were expecting. In the old (pre-Java 5) days, you always assumed that you might get the wrong thing out of a collection (since they were all non-type safe), so you took appropriate defensive steps in your code. The problem with mixing generic with non-generic code is that you won't be expecting those problems if you have been lulled into a false sense of security by having written type safe code. Just remember that the moment you turn that type safe collection over to older, non-type safe code, your protection vanishes.

Again, pay very close attention to compiler warnings, and be prepared to see issues like this come up on the exam.

exam Watch

When using legacy (non-type safe) collections—watch out for unboxing problems! If you declare a non-generic collection, the `get()` ***method ALWAYS returns a reference of type java.lang.Object. Remember that unboxing can't convert a plain old Object to a primitive, even if that Object reference points to an Integer (or some other primitive) on the heap. Unboxing converts only from a wrapper class reference (like an Integer or a Long) to a primitive.***

exam Watch

Unboxing gotcha, continued:

```
List test = new ArrayList();
test.add(43);
int x = (Integer)test.get(0);    // you must cast !!

List<Integer> test2 = new ArrayList<Integer>();
test2.add(343);
int x2 = test2.get(0);           // cast not necessary
```

Watch out for missing casts associated with pre-Java 5, non-generic collections.

Polymorphism and Generics

Generic collections give you the same benefits of type safety that you've always had with arrays, but there are some crucial differences that can bite you if you aren't prepared. Most of these have to do with polymorphism.

You've already seen that polymorphism applies to the "base" type of the collection:

```
List<Integer> myList = new ArrayList<Integer>();
```

In other words, we were able to assign an ArrayList to a List reference, because List is a supertype of ArrayList. Nothing special there—this polymorphic assignment works the way it always works in Java, regardless of the generic typing.

But what about this?

```
class Parent { }
class Child extends Parent { }
List<Parent> myList = new ArrayList<Child>();
```

Think about it for a minute.

Keep thinking...

No, it doesn't work. There's a very simple rule here—the type of the variable declaration must match the type you pass to the actual object type. If you declare List<Foo> foo then whatever you assign to the foo reference MUST be of the generic type <Foo>. Not a subtype of <Foo>. Not a supertype of <Foo>. Just <Foo>.

These are wrong:

```
List<Object> myList = new ArrayList<JButton>();  // NO!
List<Number> numbers = new ArrayList<Integer>(); // NO!
// remember that Integer is a subtype of Number
```

But these are fine:

```
List<JButton> myList = new ArrayList<JButton>(); // yes
List<Object> myList = new ArrayList<Object>();   // yes
List<Integer> myList = new ArrayList<Integer>(); // yes
```

So far so good. Just keep the generic type of the reference and the generic type of the object to which it refers identical. In other words, polymorphism applies here to only the "base" type. And by "base," we mean the type of the collection class itself—the class that can be customized with a type. In this code,

```
List<JButton> myList = new ArrayList<JButton>();
```

List and ArrayList are the *base* type and JButton is the *generic* type. So an ArrayList can be assigned to a List, but a collection of <JButton> cannot be assigned to a reference of <Object>, even though JButton is a subtype of Object.

The part that feels wrong for most developers is that this is NOT how it works with arrays, where you *are* allowed to do this,

```
import java.util.*;
class Parent { }
class Child extends Parent { }
public class TestPoly {
    public static void main(String[] args) {
        Parent[] myArray = new Child[3];  // yes
    }
}
```

which means you're also allowed to do this

```
Object[] myArray = new JButton[3];  // yes
```

but not this:

```
List<Object> list = new ArrayList<JButton>();  // NO!
```

Why are the rules for typing of arrays different from the rules for generic typing? We'll get to that in a minute. For now, just burn it into your brain that polymorphism does not work the same way for generics as it does with arrays.

Generic Methods

If you weren't already familiar with generics, you might be feeling very uncomfortable with the implications of the previous no-polymorphic-assignment-for-generic-types thing. And why shouldn't you be uncomfortable? One of the biggest benefits of polymorphism is that you can declare, say, a method argument of a particular type and at runtime be able to have that argument refer to any subtype—including those you'd never known about at the time you wrote the method with the supertype argument.

For example, imagine a classic (simplified) polymorphism example of a veterinarian (AnimalDoctor) class with a method `checkup()`. And right now, you have three Animal subtypes—Dog, Cat, and Bird—each implementing the `abstract checkup()` method from Animal:

```
abstract class Animal {
    public abstract void checkup();
}
class Dog extends Animal {
    public void checkup() {   // implement Dog-specific code
      System.out.println("Dog checkup");
    }
}
class Cat extends Animal {
    public void checkup() {   // implement Cat-specific code
      System.out.println("Cat checkup");
    }
}
class Bird extends Animal {
    public void checkup() {   // implement Bird-specific code
      System.out.println("Bird checkup");
}   }
```

Forgetting collections/arrays for a moment, just imagine what the AnimalDoctor class needs to look like in order to have code that takes any kind of Animal and invokes the Animal `checkup()` method. Trying to overload the AnimalDoctor class with checkup() methods for every possible kind of animal is ridiculous, and obviously not extensible. You'd have to change the AnimalDoctor class every time someone added a new subtype of Animal.

So in the AnimalDoctor class, you'd probably have a polymorphic method:

```
public void checkAnimal(Animal a) {
     a.checkup(); // does not matter which animal subtype each
                  // Animal's overridden checkup() method runs
}
```

And of course we do want the AnimalDoctor to also have code that can take arrays of Dogs, Cats, or Birds, for when the vet comes to the dog, cat, or bird kennel. Again, we don't want overloaded methods with arrays for each potential Animal subtype, so we use polymorphism in the AnimalDoctor class:

```
public void checkAnimals(Animal[] animals) {
    for(Animal a : animals) {
      a.checkup();
    }
  }
```

Here is the entire example, complete with a test of the array polymorphism that takes any type of animal array (`Dog[]`, `Cat[]`, `Bird[]`).

```
import java.util.*;
abstract class Animal {
     public abstract void checkup();
}
class Dog extends Animal {
    public void checkup() {   // implement Dog-specific code
      System.out.println("Dog checkup");
    }
}
class Cat extends Animal {
   public void checkup() {    // implement Cat-specific code
      System.out.println("Cat checkup");
   }
}
```

```
class Bird extends Animal {
   public void checkup() {     // implement Bird-specific code
      System.out.println("Bird checkup");
   }
}
public class AnimalDoctor  {
    // method takes an array of any animal subtype
    public void checkAnimals(Animal[] animals) {
      for(Animal a : animals) {
        a.checkup();
      }
    }
    public static void main(String[] args) {
        // test it
       Dog[] dogs = {new Dog(), new Dog()};
       Cat[] cats = {new Cat(), new Cat(), new Cat()};
       Bird[] birds = {new Bird()};

       AnimalDoctor doc = new AnimalDoctor();
       doc.checkAnimals(dogs);  // pass the Dog[]
       doc.checkAnimals(cats);  // pass the Cat[]
       doc.checkAnimals(birds); // pass the Bird[]
   }
}
```

This works fine, of course (we know, we know, this is old news). But here's why we brought this up as refresher—this approach does NOT work the same way with type safe collections!

In other words, a method that takes, say, an ArrayList<Animal> will NOT be able to accept a collection of any Animal subtype! That means ArrayList<Dog> cannot be passed into a method with an argument of ArrayList<Animal>, even though we already know that this works just fine with plain old arrays.

Obviously this difference between arrays and ArrayList is consistent with the polymorphism assignment rules we already looked at—the fact that you cannot assign an object of type ArrayList<JButton> to a List<Object>. But this is where you really start to feel the pain of the distinction between typed arrays and typed collections.

We know it won't work correctly, but let's try changing the AnimalDoctor code to use generics instead of arrays:

```
public class AnimalDoctorGeneric  {
```

```
    // change the argument from Animal[] to ArrayList<Animal>
    public void checkAnimals(ArrayList<Animal> animals) {
      for(Animal a : animals) {
        a.checkup();
      }
    }
    public static void main(String[] args) {
       // make ArrayLists instead of arrays for Dog, Cat, Bird
       List<Dog> dogs = new ArrayList<Dog>();
       dogs.add(new Dog());
       dogs.add(new Dog());
       List<Cat> cats = new ArrayList<Cat>();
       cats.add(new Cat());
       cats.add(new Cat());
       List<Bird> birds = new ArrayList<Bird>();
       birds.add(new Bird());
       // this code is the same as the Array version
       AnimalDoctorGeneric doc = new AnimalDoctorGeneric();
       // this worked when we used arrays instead of ArrayLists
       doc.checkAnimals(dogs);  // send a List<Dog>
       doc.checkAnimals(cats);  // send a List<Cat>
       doc.checkAnimals(birds); // send a List<Bird>
    }
}
```

So what does happen?

```
javac AnimalDoctorGeneric.java
AnimalDoctorGeneric.java:51: checkAnimals(java.util.
ArrayList<Animal>) in AnimalDoctorGeneric cannot be applied to
(java.util.List<Dog>)
       doc.checkAnimals(dogs);
          ^
AnimalDoctorGeneric.java:52: checkAnimals(java.util.
ArrayList<Animal>) in AnimalDoctorGeneric cannot be applied to
(java.util.List<Cat>)
       doc.checkAnimals(cats);
          ^
AnimalDoctorGeneric.java:53: checkAnimals(java.util.
ArrayList<Animal>) in AnimalDoctorGeneric cannot be applied to
(java.util.List<Bird>)
       doc.checkAnimals(birds);
          ^
3 errors
```

The compiler stops us with errors, not warnings. You simply CANNOT assign the individual ArrayLists of Animal subtypes (<Dog>, <Cat>, or <Bird>) to an ArrayList of the supertype <Animal>, which is the declared type of the argument.

This is one of the biggest gotchas for Java programmers who are so familiar with using polymorphism with arrays, where the same scenario (`Animal[]` can refer to `Dog[]`, `Cat[]`, or `Bird[]`) works as you would expect. So we have two real issues:

1. Why doesn't this work?
2. How do you get around it?

You'd hate us and all of the Sun engineers if we told you that there wasn't a way around it—that you had to accept it and write horribly inflexible code that tried to anticipate and code overloaded methods for each specific <type>. Fortunately, there is a way around it.

But first, why can't you do it if it works for arrays? Why can't you pass an ArrayList<Dog> into a method with an argument of ArrayList<Animal>?

We'll get there, but first let's step way back for a minute and consider this perfectly legal scenario:

```
Animal[] animals = new Animal[3];
animals[0] = new Cat();
animals[1] = new Dog();
```

Part of the benefit of declaring an array using a more abstract supertype is that the array itself can hold objects of multiple subtypes of the supertype, and then you can manipulate the array assuming everything in it can respond to the Animal interface (in other words, everything in the array can respond to method calls defined in the Animal class). So here, we're using polymorphism not for the object that the array reference points to, but rather what the array can actually HOLD—in this case, any subtype of Animal. You can do the same thing with generics:

```
List<Animal> animals = new ArrayList<Animal>();
animals.add(new Cat());  // OK
animals.add(new Dog());  // OK
```

So this part works with both arrays and generic collections—we can add an instance of a subtype into an array or collection declared with a supertype. You can add Dogs and Cats to an Animal array (`Animal[]`) or an Animal collection (ArrayList<Animal>).

And with arrays, this applies to what happens within a method:

```
public void addAnimal(Animal[] animals) {
      animals[0] = new Dog();  // no problem, any Animal works
                               // in Animal[]
}
```

So if this is true, and if you can put Dogs into an ArrayList<Animal>, then why can't you use that same kind of method scenario? Why can't you do this?

```
public void addAnimal(ArrayList<Animal> animals) {
     animals.add(new Dog()); // sometimes allowed...
}
```

Actually, you CAN do this under certain conditions. The code above WILL compile just fine IF what you pass into the method is also an ArrayList<Animal>. This is the part where it differs from arrays, because in the array version, you COULD pass a `Dog[]` into the method that takes an `Animal[]`.

The ONLY thing you can pass to a method argument of ArrayList<Animal> is an ArrayList<Animal>! (Assuming you aren't trying to pass a subtype of ArrayList, since remember—the "base" type can be polymorphic.)

The question is still out there—why is this bad? And why is it bad for ArrayList but not arrays? Why can't you pass an ArrayList<Dog> to an argument of ArrayList<Animal>? Actually, the problem IS just as dangerous whether you're using arrays or a generic collection. It's just that the compiler and JVM behave differently for arrays vs. generic collections.

The reason it is dangerous to pass a collection (array or ArrayList) of a subtype into a method that takes a collection of a supertype, is because you might add something. And that means you might add the WRONG thing! This is probably really obvious, but just in case (and to reinforce), let's walk through some scenarios. The first one is simple:

```
public void foo() {
  Dog[] dogs = {new Dog(), new Dog()};
  addAnimal(dogs);  // no problem, send the Dog[] to the method
}
public void addAnimal(Animal[] animals) {
  animals[0] = new Dog();  // ok, any Animal subtype works
}
```

This is no problem. We passed a `Dog[]` into the method, and added a Dog to the array (which was allowed since the method parameter was type `Animal[]`, which can hold any Animal subtype). But what if we changed the calling code to

```
public void foo() {
  Cat[] cats = {new Cat(), new Cat()};
  addAnimal(cats);  // no problem, send the Cat[] to the method
}
```

and the original method stays the same:

```
public void addAnimal(Animal[] animals) {
   animals[0] = new Dog();  // Eeek! We just put a Dog
                            // in a Cat array!
}
```

The compiler thinks it is perfectly fine to add a Dog to an `Animal[]` array, since a Dog can be assigned to an Animal reference. The problem is, if you passed in an array of an Animal subtype (Cat, Dog, or Bird), the compiler does not know. The compiler does not realize that out on the heap somewhere is an array of type `Cat[]`, not `Animal[]`, and you're about to try to add a Dog to it. To the compiler, you have passed in an array of type Animal, so it has no way to recognize the problem.

THIS is the scenario we're trying to prevent, regardless of whether it's an array or an ArrayList. The difference is, the compiler lets you get away with it for arrays, but not for generic collections.

The reason the compiler won't let you pass an ArrayList<Dog> into a method that takes an ArrayList<Animal>, is because within the method, that parameter is of type ArrayList<Animal>, and that means you could put *any* kind of Animal into it. There would be no way for the compiler to stop you from putting a Dog into a List that was originally declared as <Cat>, but is now referenced from the <Animal> parameter.

We still have two questions…how do you get around it and why the heck does the compiler allow you to take that risk for arrays but not for ArrayList (or any other generic collection)?

The reason you can get away with compiling this for arrays is because there is a runtime exception (ArrayStoreException) that will prevent you from putting the wrong type of object into an array. If you send a `Dog` array into the method that takes an `Animal` array, and you add only Dogs (including Dog subtypes, of course) into the array now referenced by `Animal`, no problem. But if you DO try to add a Cat to the object that is actually a `Dog` array, you'll get the exception.

But there IS no equivalent exception for generics, because of type erasure! In other words, at runtime the JVM KNOWS the type of arrays, but does NOT know the type of a collection. All the generic type information is removed during compilation, so by the time it gets to the JVM, there is simply no way to recognize the disaster of putting a Cat into an ArrayList<Dog> and vice versa (and it becomes exactly like the problems you have when you use legacy, non-type safe code).

So this actually IS legal code:

```
public void addAnimal(List<Animal> animals) {
    animals.add(new Dog());  // this is always legal,
                             // since Dog can
                             // be assigned to an Animal
                             // reference
 }
  public static void main(String[] args) {
     List<Animal> animals = new ArrayList<Animal>();
     animals.add(new Dog());
     animals.add(new Dog());
     AnimalDoctorGeneric doc = new AnimalDoctorGeneric();
     doc.addAnimal(animals); // OK, since animals matches
                             // the method arg
  }
```

As long as the only thing you pass to the `addAnimals(List<Animal>)` is an ArrayList<Animal>, the compiler is pleased—knowing that any Animal subtype you add will be valid (you can always add a Dog to an Animal collection, yada, yada, yada). But if you try to invoke `addAnimal()` with an argument of any OTHER ArrayList type, the compiler will stop you, since at runtime the JVM would have no way to stop you from adding a Dog to what was created as a Cat collection.

For example, this code that changes the generic type to <Dog>, but without changing the `addAnimal()` method, will NOT compile:

```
public void addAnimal(List<Animal> animals) {
  animals.add(new Dog());  // still OK as always
}
public static void main(String[] args) {
  List<Dog> animals = new ArrayList<Dog>();
  animals.add(new Dog());
  animals.add(new Dog());
  AnimalDoctorGeneric doc = new AnimalDoctorGeneric();
  doc.addAnimal(animals); // THIS is where it breaks!
}
```

The compiler says something like:

```
javac AnimalDoctorGeneric.java
AnimalDoctorGeneric.java:49: addAnimal(java.util.List<Animal>)
in AnimalDoctorGeneric cannot be applied to (java.util.
List<Dog>)
       doc.addAnimal(animals);
           ^
1 error
```

Notice that this message is virtually the same one you'd get trying to invoke any method with the wrong argument. It's saying that you simply cannot invoke `addAnimal(List<Animal>)` using something whose reference was declared as List<Dog>. (It's the reference type, not the actual object type that matters—but remember—the generic type of an object is ALWAYS the same as the generic type declared on the reference. List<Dog> can refer ONLY to collections that are subtypes of List, but which were instantiated as generic type <Dog>.)

Once again, remember that once inside the `addAnimals()` method, all that matters is the type of the parameter—in this case, List<Animal>. (We changed it from ArrayList to List to keep our "base" type polymorphism cleaner.)

Back to the key question—how do we get around this? If the problem is related only to the danger of adding the wrong thing to the collection, what about the `checkup()` method that used the collection passed in as read-only? In other words, what about methods that invoke Animal methods on each thing in the collection, which will work regardless of which kind of ArrayList subtype is passed in?

And that's a clue! It's the `add()` method that is the problem, so what we need is a way to tell the compiler, "Hey, I'm using the collection passed in just to invoke methods on the elements—and I promise not to ADD anything into the collection." And there IS a mechanism to tell the compiler that you can take any generic subtype of the declared argument type because you won't be putting anything in the collection. And that mechanism is the wildcard `<?>`.

The method signature would change from

```
public void addAnimal(List<Animal> animals)
```

to

```
public void addAnimal(List<? extends Animal> animals)
```

By saying `<? extends Animal>`, we're saying, "I can be assigned a collection that is a subtype of List and typed for <Animal> or anything that *extends* Animal. And oh yes, I SWEAR that I will not ADD anything into the collection." (There's a little more to the story, but we'll get there.)

So of course the `addAnimal()` method above won't actually compile even with the wildcard notation, because that method DOES add something.

```
public void addAnimal(List<? extends Animal> animals) {
      animals.add(new Dog());  // NO! Can't add if we
                               // use <? extends Animal>
}
```

You'll get a very strange error that might look something like this

```
javac AnimalDoctorGeneric.java
AnimalDoctorGeneric.java:38: cannot find symbol
symbol  : method add(Dog)
location: interface java.util.List<capture of ? extends Animal>
      animals.add(new Dog());
              ^
1 error
```

which basically says, "you can't add a Dog here." If we change the method so that it doesn't add anything, it works.

But wait—there's more. (And by the way, everything we've covered in this generics section is likely to be tested for on the exam, with the exception of "type erasure," for which you aren't required to know any details.)

First, the `<? extends Animal>` means that you can take any subtype of Animal; however, that subtype can be EITHER a subclass of a class (abstract or concrete) OR a type that implements the interface after the word `extends`. In other words, the keyword `extends` in the context of a wildcard represents BOTH subclasses and interface implementations. There is no `<? implements Serializable>` syntax. If you want to declare a method that takes anything that is of a type that implements Serializable, you'd still use `extends` like this:

```
void foo(List<? extends Serializable> list) // odd, but correct
                                            // to use "extends"
```

This looks strange since you would never say this in a class declaration because Serializable is an interface, not a class. But that's the syntax, so burn it in!

One more time—there is only ONE wildcard keyword that represents *both* interface implementations and subclasses. And that keyword is `extends`. But when you see it, think "Is-a", as in something that passes the `instanceof` test.

However, there is another scenario where you can use a wildcard AND still add to the collection, but in a safe way—the keyword `super`.

Imagine, for example, that you declared the method this way:

```
public void addAnimal(List<? super Dog> animals) {
  animals.add(new Dog());  // adding is sometimes OK with super
}
public static void main(String[] args) {
  List<Animal> animals = new ArrayList<Animal>();
  animals.add(new Dog());
  animals.add(new Dog());
  AnimalDoctorGeneric doc = new AnimalDoctorGeneric();
  doc.addAnimal(animals); // passing an Animal List
}
```

Now what you've said in this line

```
public void addAnimal(List<? super Dog> animals)
```

is essentially, "Hey compiler, please accept any List with a generic type that is of type Dog, or a supertype of Dog. Nothing lower in the inheritance tree can come in, but anything higher than Dog is OK."

You probably already recognize why this works. If you pass in a list of type Animal, then it's perfectly fine to add a Dog to it. If you pass in a list of type Dog, it's perfectly fine to add a Dog to it. And if you pass in a list of type Object, it's STILL fine to add a Dog to it. When you use the `<? super ...>` syntax, you are telling the compiler that you can accept the type on the right-hand side of `super` or any of its supertypes, since—and this is the key part that makes it work—a collection declared as any supertype of Dog will be able to accept a Dog as an element. List<Object> can take a Dog. List<Animal> can take a Dog. And List<Dog> can take a Dog. So passing any of those in will work. So the `super` keyword in wildcard notation lets you have a restricted, but still possible way to add to a collection.

So, the wildcard gives you polymorphic assignments, but with certain restrictions that you don't have for arrays. Quick question: are these two identical?

```
public void foo(List<?> list) { }
public void foo(List<Object> list) { }
```

If there IS a difference (and we're not yet saying there is), what is it?

There IS a huge difference. `List<?>`, which is the wildcard `<?>` without the keywords `extends` or `super`, simply means "any type." So that means any type of List can be assigned to the argument. That could be a List of <Dog>, <Integer>, <JButton>, <Socket>, whatever. And using the wildcard alone, without the keyword `super` (followed by a type), means that you cannot ADD anything to the list referred to as `List<?>`.

`List<Object>` is completely different from `List<?>`. `List<Object>` means that the method can take ONLY a `List<Object>`. Not a `List<Dog>`, or a `List<Cat>`. It does, however, mean that you can add to the list, since the compiler has already made certain that you're passing only a valid `List<Object>` into the method.

Based on the previous explanations, figure out if the following will work:

```
import java.util.*;
public class TestWildcards {
    public static void main(String[] args) {
        List<Integer> myList = new ArrayList<Integer>();
        Bar bar = new Bar();
        bar.doInsert(myList);
      }
}
class Bar {
   void doInsert(List<?> list) {
     list.add(new Dog());
   }
}
```

If not, where is the problem?

The problem is in the `list.add()` method within `doInsert()`. The `<?>` wildcard allows a list of ANY type to be passed to the method, but the `add()` method is not valid, for the reasons we explored earlier (that you could put the wrong kind of thing into the collection). So this time, the TestWildcards class is fine, but the Bar class won't compile because it does an `add()` in a method that uses a wildcard (without `super`). What if we change the `doInsert()` method to this:

```
public class TestWildcards {
    public static void main(String[] args) {
        List<Integer> myList = new ArrayList<Integer>();
        Bar bar = new Bar();
        bar.doInsert(myList);
      }
}
```

```
class Bar {
   void doInsert(List<Object> list) {
     list.add(new Dog());
   }
}
```

Now will it work? If not, why not?

This time, class Bar, with the `doInsert()` method, compiles just fine. The problem is that the TestWildcards code is trying to pass a List<Integer> into a method that can take ONLY a List<Object>. And *nothing* else can be substituted for <Object>.

By the way, `List<? extends Object>` and `List<?>` are absolutely identical! They both say, "I can refer to any type of object." But as you can see, neither of them are the same as List<Object>. One way to remember this is that if you see the wildcard notation (a question mark `?`), this means "many possibilities". If you do NOT see the question mark, then it means the <type> in the brackets, and absolutely NOTHING ELSE. List<Dog> means List<Dog> and not List<Beagle>, List<Poodle>, or any other subtype of Dog. But List<? extends Dog> could mean List<Beagle>, List<Poodle>, and so on. Of course List<?> could be... anything at all.

Keep in mind that the wildcards can be used only for reference declarations (including arguments, variables, return types, and so on). They can't be used as the type parameter when you create a new typed collection. Think about that—while a reference can be abstract and polymorphic, the actual object created must be of a specific type. You have to lock down the type when you make the object using `new`.

As a little review before we move on with generics, look at the following statements and figure out which will compile:

```
1) List<?> list = new ArrayList<Dog>();
2) List<? extends Animal> aList = new ArrayList<Dog>();
3) List<?> foo = new ArrayList<? extends Animal>();
4) List<? extends Dog> cList = new ArrayList<Integer>();
5) List<? super Dog> bList = new ArrayList<Animal>();
6) List<? super Animal> dList = new ArrayList<Dog>();
```

The correct answers (the statements that compile) are 1, 2, and 5.

The three that won't compile are

- Statement: `List<?> foo = new ArrayList<? extends Animal>();`
  Problem: you cannot use wildcard notation in the object creation. So the `new ArrayList<? extends Animal>()` will not compile.

- Statement: `List<? extends Dog> cList = new ArrayList<Integer>();`

  Problem: You cannot assign an Integer list to a reference that takes only a Dog (including any subtypes of Dog, of course).
- Statement: `List<? super Animal> dList = new ArrayList<Dog>();`

  Problem: You cannot assign a Dog to <? super Animal>. The Dog is too "low" in the class hierarchy. Only <Animal> or <Object> would have been legal.

Generic Declarations

Until now, we've talked about how to create type safe collections, and how to declare reference variables including arguments and return types using generic syntax. But here are a few questions: How do we even know that we're allowed/supposed to specify a type for these collection classes? And does generic typing work with any other classes in the API? And finally, can we declare our own classes as generic types? In other words, can we make a class that requires that someone pass a type in when they declare it and instantiate it?

First, the one you obviously know the answer to—the API tells you when a parameterized type is expected. For example, this is the API declaration for the java.util.List interface:

```
public interface List<E>
```

The <E> is a placeholder for the type you pass in. The List interface is behaving as a generic "template" (sort of like C++ templates), and when you write your code, you change it from a generic List to a List<Dog> or List<Integer>, and so on.

The E, by the way, is only a convention. Any valid Java identifier would work here, but E stands for "Element," and it's used when the template is a collection. The other main convention is T (stands for "type"), used for, well, things that are NOT collections.

Now that you've seen the interface declaration for List, what do you think the `add()` method looks like?

```
boolean add(E o)
```

In other words, whatever E is when you declare the List, *that's what you can add to it*. So imagine this code:

```
List<Animal> list = new ArrayList<Animal>();
```

The E in the List API suddenly has its waveform collapsed, and goes from the abstract <your type goes here>, to a List of Animals. And if it's a List of Animals, then the `add()` method of List must obviously behave like this:

```
boolean add(Animal a)
```

When you look at an API for a generics class or interface, pick a type parameter (Dog, JButton, even Object) and do a mental find and replace on each instance of E (or whatever identifier is used as the placeholder for the type parameter).

Making Your Own Generic Class

Let's try making our own generic class, to get a feel for how it works, and then we'll look at a few remaining generics syntax details. Imagine someone created a class Rental, that manages a pool of rentable items.

```
public class Rental {
   private List rentalPool;
   private int maxNum;
   public Rental(int maxNum, List rentalPool) {
      this.maxNum = maxNum;
      this.rentalPool = rentalPool;
   }
   public Object getRental() {
      // blocks until there's something available
      return rentalPool.get(0);
   }
   public void returnRental(Object o) {
      rentalPool.add(o);
   }
}
```

Now imagine you wanted to make a subclass of Rental that was just for renting cars. You might start with something like this:

```
import java.util.*;
public class CarRental extends Rental {
   public CarRental(int maxNum, List<Car> rentalPool) {
      super(maxNum, rentalPool);
   }
   public Car getRental() {
      return (Car) super.getRental();
   }
```

```
   public void returnRental(Car c) {
      super.returnRental(c);
   }
   public void returnRental(Object o) {
     if (o instanceof Car) {
       super.returnRental(o);
     } else {
        System.out.println("Cannot add a non-Car");
        // probably throw an exception
}  } }
```

But then the more you look at it, the more you realize:

1. You are doing your own type checking in the `returnRental()` method. You can't change the argument type of `returnRental()` to take a Car, since it's an override (not an overload) of the method from class Rental. (Overloading would take away your polymorphic flexibility with Rental).
2. You really don't want to make separate subclasses for every possible kind of rentable thing (cars, computers, bowling shoes, children, and so on).

But given your natural brilliance (heightened by this contrived scenario), you quickly realize that you can make the Rental class a generic type—a template for any kind of Rentable thing—and you're good to go.

(We did say contrived…since in reality, you might very well want to have different behaviors for different kinds of rentable things, but even that could be solved cleanly through some kind of behavior composition as opposed to inheritance (using the Strategy design pattern, for example). And no, design patterns aren't on the exam, but we still think you should read our design patterns book. Think of the kittens.) So here's your new and improved generic Rental class:

```
import java.util.*;
public class RentalGeneric<T> {  // "T" is for the type
                                 // parameter
   private List<T> rentalPool;   // Use the class type for the
                                 // List type
   private int maxNum;
   public RentalGeneric(
     int maxNum, List<T> rentalPool) { // constructor takes a
                                       // List of the class type
     this.maxNum = maxNum;
     this.rentalPool = rentalPool;
   }
```

```
    public T getRental() {                    // we rent out a T
      // blocks until there's something available
      return rentalPool.get(0);
    }
    public void returnRental(T returnedThing) { // and the renter
                                                // returns a T
      rentalPool.add(returnedThing);
    }
}
```

Let's put it to the test:

```
class TestRental {
  public static void main (String[] args) {
     //make some Cars for the pool
     Car c1 = new Car();
     Car c2 = new Car();
     List<Car> carList = new ArrayList<Car>();
     carList.add(c1);
     carList.add(c2);
     RentalGeneric<Car> carRental = new
                         RentalGeneric<Car>(2, carList);
     // now get a car out, and it won't need a cast
     Car carToRent = carRental.getRental();
     carRental.returnRental(carToRent);
     // can we stick something else in the original carList?
     carList.add(new Cat("Fluffy"));
  }
}
```

We get one error:

```
kathy% javac1.5 RentalGeneric.java
RentalGeneric.java:38: cannot find symbol
symbol  : method add(Cat)
location: interface java.util.List<Car>
     carList.add(new Cat("Fluffy"));
             ^
1 error
```

Now we have a Rental class that can be *typed* to whatever the programmer chooses, and the compiler will enforce it. In other words, it works just as the

Collections classes do. Let's look at more examples of generic syntax you might find in the API or source code. Here's another simple class that uses the parameterized type of the class in several ways:

```
public class TestGenerics<T> {  // as the class type
   T anInstance;                // as an instance variable type
   T [] anArrayOfTs;            // as an array type

   TestGenerics(T anInstance) {     // as an argument type
      this.anInstance = anInstance;
   }
   T getT() {                       // as a return type
      return anInstance;
   }
}
```

Obviously this is a ridiculous use of generics, and in fact you'll see generics only rarely outside of collections. But, you do need to understand the different kinds of generic syntax you might encounter, so we'll continue with these examples until we've covered them all.

You can use more than one parameterized type in a single class definition:

```
public class UseTwo<T, X> {
  T one;
  X two;
  UseTwo(T one, X two) {
    this.one = one;
    this.two = two;
  }
  T getT() { return one; }
  X getX() { return two; }

// test it by creating it with <String, Integer>

  public static void main (String[] args) {
    UseTwo<String, Integer> twos =
                    new UseTwo<String, Integer>("foo", 42);

    String theT = twos.getT(); // returns a String
    int theX = twos.getX();   // returns Integer, unboxes to int
  }
}
```

And you can use a form of wildcard notation in a class definition, to specify a range (called "bounds") for the type that can be used for the type parameter:

```
public class AnimalHolder<T extends Animal> { // use "T" instead
                                               // of "?"
 T animal;
 public static void main(String[] args) {
   AnimalHolder<Dog> dogHolder = new AnimalHolder<Dog>(); // OK
   AnimalHolder<Integer> x = new AnimalHolder<Integer>(); // NO!
 }
}
```

Creating Generic Methods

Until now, every example we've seen uses the class parameter type—the type declared with the class name. For example, in the UseTwo<T,X> declaration, we used the T and X placeholders throughout the code. But it's possible to define a parameterized type at a more granular level—a method.

Imagine you want to create a method that takes an instance of any type, instantiates an ArrayList of that type, and adds the instance to the ArrayList. The class itself doesn't need to be generic; basically we just want a utility method that we can pass a type to and that can use that type to construct a type safe collection. Using a generic method, we can declare the method without a specific type and then get the type information based on the type of the object passed to the method. For example:

```
import java.util.*;
public class CreateAnArrayList {
  public <T> void makeArrayList(T t) { // take an object of an
                                       // unknown type and use a
                                       // "T" to represent the type
    List<T> list = new ArrayList<T>(); // now we can create the
                                       // list using "T"
    list.add(t);
   }
}
```

In the preceding code, if you invoke the `makeArrayList()` method with a Dog instance, the method will behave as though it looked like this all along:

```
  public void makeArrayList(Dog t) {
     List<Dog> list = new ArrayList<Dog>();
     list.add(t);
  }
```

And of course if you invoke the method with an Integer, then the T is replaced by Integer (not in the bytecode, remember—we're describing how it appears to behave, not how it actually gets it done).

The strangest thing about generic methods is that you must declare the type variable BEFORE the return type of the method:

```
public <T> void makeArrayList(T t)
```

The <T> before `void` simply defines what T is before you use it as a type in the argument. You MUST declare the type like that unless the type is specified for the class. In CreateAnArrayList, the class is not generic, so there's no type parameter placeholder we can use.

You're also free to put boundaries on the type you declare, for example if you want to restrict the `makeArrayList()` method to only Number or its subtypes (Integer, Float, and so on) you would say

```
public <T extends Number> void makeArrayList(T t)
```

exam Watch

It's tempting to forget that the method argument is NOT where you declare the type parameter variable T. In order to use a type variable like T, you must have declared it either as the class parameter type or in the method, before the return type. The following might look right,

```
public void makeList(T t) { }
```

But the only way for this to be legal is if there is actually a class named T, in which case the argument is like any other type declaration for a variable. And what about constructor arguments? They, too, can be declared with a generic type, but then it looks even stranger since constructors have no return type at all:

```
public class Radio {
   public <T> Radio(T t) { }  // legal constructor
}
```

exam Watch

If you REALLY want to get ridiculous (or fired), you can declare a class with a name that is the same as the type parameter placeholder:

```
class X { public <X> X(X x) { } }
```

Yes, this works. The X that is the constructor name has no relationship to the <X> type declaration, which has no relationship to the constructor argument identifier, which is also, of course, X. The compiler is able to parse this and treat each of the different uses of X independently. So there is no naming conflict between class names, type parameter placeholders, and variable identifiers.

exam Watch

One of the most common mistakes programmers make when creating generic classes or methods is to use a <?> in the wildcard syntax rather than a type variable <T>, <E>, and so on. This code might look right, but isn't:

```
public class NumberHolder<? extends Number> { }
```

While the question mark works when declaring a reference for a variable, it does NOT work for generic class and method declarations. This code is not legal:

```
public class NumberHolder<?> { ? aNum; }    // NO!
```

But if you replace the <?> with a legal identifier, you're good:

```
public class NumberHolder<T> { T aNum; }    // Yes
```

98% of what you're likely to do with generics is simply declare and use type safe collections, including using (and passing) them as arguments. But now you know much more (but by no means everything) about the way generics works.

If this was clear and easy for you, that's excellent. If it was…painful…just know that adding generics to the Java language very nearly caused a revolt among some of the most experienced Java developers. Most of the outspoken critics are simply unhappy with the complexity, or aren't convinced that gaining type safe collections is worth the ten million little rules you have to learn now. It's true that with Java 5, learning Java just got harder. But trust us…we've never seen it take more than two days to "get" generics. That's 48 consecutive hours.

CERTIFICATION SUMMARY

We began with a quick review of the `toString()` method. The `toString()` method is automatically called when you ask `System.out.println()` to print an object—you override it to return a String of meaningful data about your objects.

Next we 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.

We covered the ins and outs of the Collections and Arrays classes: how to sort, and how to search. We learned about converting arrays to Lists and back again.

Finally we tackled generics. Generics let you enforce compile-time type-safety on collections or other classes. Generics help assure you that when you get an item from a collection it will be of the type you expect, with no casting required. You can mix legacy code with generics code, but this can cause exceptions. The rules for polymorphism change when you use generics, although by using wildcards you can still create polymorphic collections. Some generics declarations allow reading of a collection, but allow no updating of the collection.

All in all, one fascinating chapter.

TWO-MINUTE DRILL

Here are some of the key points from this chapter.

Overriding hashCode() and equals() (Objective 6.2)

- `equals()`, `hashCode()`, and `toString()` are `public`.
- Override `toString()` so that `System.out.println()` or other methods can see something useful, like 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 hashing keys.
- If you don't override `equals()`, different objects can't be considered equal.
- Strings and wrappers override `equals()` and make good hashing 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()` is true.
- If you override `equals()`, override `hashCode()`.
- HashMap, HashSet, Hashtable, LinkedHashMap, & LinkedHashSet use hashing.
- An appropriate `hashCode()` override sticks to the `hashCode()` contract.
- An efficient `hashCode()` override distributes keys evenly across its buckets.
- An overridden `equals()` must be at least as precise as its `hashCode()` mate.
- 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`, `x.hashCode() == y.hashCode()` is `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 (Objective 6.1)

- 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 interface from which Set and List extend
  - **Collections** A class that holds static collection utility methods
- Four basic flavors of collections include Lists, Sets, Maps, Queues:
  - **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 are not allowed.
  - **Queues of things to process** Ordered by FIFO or by priority.
- Four basic sub-flavors of collections Sorted, Unsorted, Ordered, Unordered.
  - **Ordered** Iterating through a collection in a specific, non-random order.
  - **Sorted** Iterating through a collection in a sorted order.
- Sorting can be alphabetic, numeric, or programmer-defined.

Key Attributes of Common Collection Classes (Objective 6.1)

- ArrayList: Fast iteration and fast random access.
- Vector: It's like a slower ArrayList, but it has synchronized methods.
- LinkedList: Good for adding elements to the ends, i.e., stacks and queues.
- HashSet: Fast access, assures no duplicates, provides no ordering.
- LinkedHashSet: No duplicates; iterates by insertion order.
- TreeSet: No duplicates; iterates in 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.
- TreeMap: A sorted map.
- PriorityQueue: A to-do list ordered by the elements' priority.

Using Collection Classes

- Collections hold only Objects, but primitives can be autoboxed.
- Iterate with the enhanced `for`, or with an Iterator via `hasNext()` & `next()`.
- `hasNext()` determines if more elements exist; the Iterator does NOT move.
- `next()` returns the next element AND moves the Iterator forward.
- To work correctly, a Map's keys must override `equals()` and `hashCode()`.
- Queues use `offer()` to add an element, `poll()` to remove the head of the queue, and `peek()` to look at the head of a queue.

Sorting and Searching Arrays and Lists

- Sorting can be in natural order, or via a Comparable or many Comparators.
- Implement Comparable using `compareTo()`; provides only one sort order.
- Create many Comparators to sort a class many ways; implement `compare()`.
- To be sorted and searched, a List's elements must be *comparable*.
- To be searched, an array or List must first be sorted.

Utility Classes: Collections and Arrays

- Both of these java.util classes provide
  - A `sort()` method. Sort using a Comparator or sort using natural order.
  - A `binarySearch()` method. Search a pre-sorted array or List.
- `Arrays.asList()` creates a List from an array and links them together.
- `Collections.reverse()` reverses the order of elements in a List.
- `Collections.reverseOrder()` returns a Comparator that sorts in reverse.
- Lists and Sets have a `toArray()` method to create arrays.

Generics

- Generics let you enforce compile-time type safety on Collections (or other classes and methods declared using generic type parameters).
- An ArrayList<Animal> can accept references of type Dog, Cat, or any other subtype of Animal (subclass, or if Animal is an interface, implementation).
- When using generic collections, a cast is not needed to get (declared type) elements out of the collection. With non-generic collections, a cast is required:

```
List<String> gList = new ArrayList<String>();
List list = new ArrayList();
// more code
String s = gList.get(0);           // no cast needed
String s = (String)list.get(0);    // cast required
```

- You can pass a generic collection into a method that takes a non-generic collection, but the results may be disastrous. The compiler can't stop the method from inserting the wrong type into the previously type safe collection.
- If the compiler can recognize that non-type-safe code is potentially endangering something you originally declared as type-safe, you will get a compiler warning. For instance, if you pass a List<String> into a method declared as

```
void foo(List aList) { aList.add(anInteger); }
```

  the compiler will issue a warning because the `add()` method is potentially an "unsafe operation."
- Remember that "compiles without error" is not the same as "compiles without warnings." On the exam, a compilation *warning* is not considered a compilation *error* or *failure*.
- Generic type information does not exist at runtime—it is for compile-time safety only. Mixing generics with legacy code can create compiled code that may throw an exception at runtime.
- Polymorphic assignments applies only to the base type, not the generic type parameter. You can say

```
List<Animal> aList = new ArrayList<Animal>();   // yes
```

  You can't say

```
List<Animal> aList = new ArrayList<Dog>();      // no
```

- The polymorphic assignment rule applies everywhere an assignment can be made. The following are NOT allowed:

```
void foo(List<Animal> aList) { }  // cannot take a List<Dog>
List<Animal> bar() { }            // cannot return a List<Dog>
```

- Wildcard syntax allows a generic method, accept subtypes (or supertypes) of the declared type of the method argument:

```
void addD(List<Dog> d) {}  // can take only <Dog>
void addD(List<? extends Dog>) {} // take a <Dog> or <Beagle>
```

- The wildcard keyword `extends` is used to mean either "extends" or "implements." So in `<? extends Dog>`, `Dog` can be a class or an interface.
- When using a wildcard, `List<? extends Dog>`, the collection can be accessed but not modified.
- When using a wildcard, `List<?>`, any generic type can be assigned to the reference, but for access only, no modifications.
- `List<Object>` refers only to a `List<Object>`, while `List<?>` or `List<? extends Object>` can hold any type of object, but for access only.
- Declaration conventions for generics use T for type and E for element:

```
public interface List<E>  // API declaration for List
boolean add(E o)          // List.add() declaration
```

- The generics type identifier can be used in class, method, and variable declarations:

```
class Foo<t> { }     // a class
T anInstance;        // an instance variable
Foo(T aRef) {}       // a constructor argument
void bar(T aRef) {}  // a method argument
T baz() {}           // a return type
```

  The compiler will substitute the actual type.

- You can use more than one parameterized type in a declaration:

```
public class UseTwo<T, X> { }
```

- You can declare a generic method using a type not defined in the class:

```
public <T> void makeList(T t) { }
```

  is NOT using T as the return type. This method has a `void` return type, but to use T within the method's argument you must declare the <T>, which happens before the return type.

SELF TEST

1. Given:

```
import java.util.*;
class Test {
  public static void main(String[] args) {
    // insert code here
    x.add("one");
    x.add("two");
    x.add("TWO");
    System.out.println(x.poll());
  }
}
```

Which, inserted at `// insert code here`, will compile? (Choose all that apply.)

A. `List<String> x = new LinkedList<String>();`

B. `TreeSet<String> x = new TreeSet<String>();`

C. `HashSet<String> x = new HashSet<String>();`

D. `Queue<String> x = new PriorityQueue<String>();`

E. `ArrayList<String> x = new ArrayList<String>();`

F. `LinkedList<String> x = new LinkedList<String>();`

2. Given:

```
public static void main(String[] args) {

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

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

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

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

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

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

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

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

G. None of the above.

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

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

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

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

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

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

4. Given:

```
import java.util.*;
class Flubber {
  public static void main(String[] args) {
    List<String> x = new ArrayList<String>();
    x.add(" x");  x.add("xx"); x.add("Xx");

    // insert code here
   for(String s: x)  System.out.println(s);
} }
```

And the output:

```
xx
Xx
 x
```

Which code, inserted at `// insert code here`, will produce the preceding output? (Choose all that apply.)

A. `Collections.sort(x);`

B.
```
Comparable c = Collections.reverse();
Collections.sort(x,c);
```

C.
```
Comparator c = Collections.reverse();
Collections.sort(x,c);
```

D.
```
Comparable c = Collections.reverseOrder();
Collections.sort(x,c);
```

E.
```
Comparator c = Collections.reverseOrder();
Collections.sort(x,c);
```

5. Given:

```
10.    public static void main(String[] args) {
11.        Queue<String> q = new LinkedList<String>();
12.        q.add("Veronica");
13.        q.add("Wallace");
14.        q.add("Duncan");
15.        showAll(q);
16.    }
17.
18.    public static void showAll(Queue q) {
19.        q.add(new Integer(42));
20.        while (!q.isEmpty())
21.            System.out.print(q.remove() + " ");
22.    }
```

What is the result?

A. `Veronica Wallace Duncan`

B. `Veronica Wallace Duncan 42`

C. `Duncan Wallace Veronica`

D. `42 Duncan Wallace Veronica`

E. Compilation fails.

F. An exception occurs at runtime.

6. Given:

```
public static void before() {
  Set set = new TreeSet();
  set.add("2");
  set.add(3);
  set.add("1");
  Iterator it = set.iterator();
    while (it.hasNext())
  System.out.print(it.next() + " ");
}
```

Which of the following statements are true?

A. The `before()` method will print `1 2`

B. The `before()` method will print `1 2 3`

C. The `before()` method will print three numbers, but the order cannot be determined.

D. The `before()` method will not compile.

E. The `before()` method will throw an exception at runtime.

7. Given:

```
import java.util.*;
class MapEQ {
  public static void main(String[] args) {
    Map<ToDos, String> m = new HashMap<ToDos, String>();
    ToDos t1 = new ToDos("Monday");
    ToDos t2 = new ToDos("Monday");
    ToDos t3 = new ToDos("Tuesday");
    m.put(t1, "doLaundry");
    m.put(t2, "payBills");
    m.put(t3, "cleanAttic");
    System.out.println(m.size());
  }
}
class ToDos{
  String day;
  ToDos(String d) { day = d; }
  public boolean equals(Object o) {
    return ((ToDos)o).day == this.day;
  }
  // public int hashCode() { return 9; }
}
```

Which is correct? (Choose all that apply.)

A. As the code stands it will not compile.

B. As the code stands the output will be `2`.

C. As the code stands the output will be `3`.

D. If the `hashCode()` method is uncommented the output will be `2`.

E. If the `hashCode()` method is uncommented the output will be `3`.

F. If the `hashCode()` method is uncommented the code will not compile.

8. Given:

```
12. public class AccountManager {
13.     private Map accountTotals = new HashMap();
14.     private int retirementFund;
15.
16.     public int getBalance(String accountName) {
17.         Integer total = (Integer) accountTotals.get(accountName);
18.         if (total == null)
19.             total = Integer.valueOf(0);
20.         return total.intValue();
21.     }
23.     public void setBalance(String accountName, int amount) {
24.         accountTotals.put(accountName, Integer.valueOf(amount));
25.     }
26. }
```

This class is to be updated to make use of appropriate generic types, with no changes in behavior (for better or worse). Which of these steps could be performed? (Choose three.)

A. Replace line 13 with

```
private Map<String, int> accountTotals = new HashMap<String, int>();
```

B. Replace line 13 with

```
private Map<String, Integer> accountTotals = new HashMap<String, Integer>();
```

C. Replace line 13 with

```
private Map<String<Integer>> accountTotals = new HashMap<String<Integer>>();
```

D. Replace lines 17–20 with

```
int total = accountTotals.get(accountName);
if (total == null)
    total = 0;
return total;
```

E. Replace lines 17–20 with
```
Integer total = accountTotals.get(accountName);
if (total == null)
    total = 0;
return total;
```

F. Replace lines 17–20 with
```
return accountTotals.get(accountName);
```

G. Replace line 24 with
```
accountTotals.put(accountName, amount);
```

H. Replace line 24 with
```
accountTotals.put(accountName, amount.intValue());
```

9. Given a properly prepared String array containing five elements, which range of results could a proper invocation of `Arrays.binarySearch()` produce?

A. 0 through 4

B. 0 through 5

C. -1 through 4

D. -1 through 5

E. -5 through 4

F. -5 through 5

G. -6 through 4

H. -6 through 5

10. Given:

```
interface Hungry<E> { void munch(E x); }
interface Carnivore<E extends Animal> extends Hungry<E> {}
interface Herbivore<E extends Plant> extends Hungry<E> {}
abstract class Plant {}
class Grass extends Plant {}
abstract class Animal {}
class Sheep extends Animal implements Herbivore<Sheep> {
  public void munch(Sheep x) {}
}
class Wolf extends Animal implements Carnivore<Sheep> {
  public void munch(Sheep x) {}
}
```

Which of the following changes (taken separately) would allow this code to compile? (Choose all that apply.)

A. Change the Carnivore interface to
```
interface Carnivore<E extends Plant> extends Hungry<E> {}
```

B. Change the Herbivore interface to
```
interface Herbivore<E extends Animal> extends Hungry<E> {}
```

C. Change the Sheep class to
```
class Sheep extends Animal implements Herbivore<Plant> {
    public void munch(Grass x) {}
}
```

D. Change the Sheep class to
```
class Sheep extends Plant implements Carnivore<Wolf> {
    public void munch(Wolf x) {}
}
```

E. Change the Wolf class to
```
class Wolf extends Animal implements Herbivore<Grass> {
    public void munch(Grass x) {}
}
```

F. No changes are necessary.

11. Which collection class(es) allows you to grow or shrink its size and provides indexed access to its elements, but whose methods are not synchronized? (Choose all that apply.)

A. `java.util.HashSet`

B. `java.util.LinkedHashSet`

C. `java.util.List`

D. `java.util.ArrayList`

E. `java.util.Vector`

F. `java.util.PriorityQueue`

12. Given:

```
import java.util.*;
public class Group extends HashSet<Person> {
    public static void main(String[] args) {
        Group g = new Group();
        g.add(new Person("Hans"));
        g.add(new Person("Lotte"));
```

```
        g.add(new Person("Jane"));
        g.add(new Person("Hans"));
        g.add(new Person("Jane"));
        System.out.println("Total: " + g.size());
    }
    public boolean add(Object o) {
        System.out.println("Adding: " + o);
        return super.add(o);
    }
}
class Person {
    private final String name;
    public Person(String name) { this.name = name; }
    public String toString() { return name; }
}
```

Which of the following occur at least once when the code is compiled and run? (Choose all that apply.)

A. `Adding Hans`

B. `Adding Lotte`

C. `Adding Jane`

D. `Total: 3`

E. `Total: 5`

F. The code does not compile.

G. An exception is thrown at runtime.

13. Given:

```
import java.util.*;
class AlgaeDiesel {
  public static void main(String[] args) {
    String[] sa = {"foo", "bar", "baz" };
    // insert method invocations here
  }
}
```

What `java.util.Arrays` and/or `java.util.Collections` methods could you use to convert `sa` to a List and then search the List to find the index of the element whose value is `"foo"`? (Choose from one to three methods.)

A. `sort()`

B. `asList()`

C. `toList()`

D. `search()`

E. `sortList()`

F. `contains()`

G. `binarySearch()`

14. Given that String implements java.lang.CharSequence, and:

```
import java.util.*;
public class LongWordFinder {
    public static void main(String[] args) {
        String[] array = { "123", "12345678", "1", "12", "1234567890"};
        List<String> list = Arrays.asList(array);
        Collection<String> resultList = getLongWords(list);
    }
    // INSERT DECLARATION HERE
    {
        Collection<E> longWords = new ArrayList<E>();
        for (E word : coll)
            if (word.length() > 6)  longWords.add(word);
        return longWords;
    }
}
```

Which declarations could be inserted at `// INSERT DECLARATION HERE` so that the program will compile and run? (Choose all that apply.)

A.
```
public static <E extends CharSequence> Collection<? extends CharSequence>
       getLongWords(Collection<E> coll)
```

B.
```
public static <E extends CharSequence> List<E>
        getLongWords(Collection<E> coll)
```

C.
```
public static Collection<E extends CharSequence>
        getLongWords(Collection<E> coll)
```

D.
```
public static List<CharSequence>
        getLongWords(Collection<CharSequence> coll)
```

E.
```
public static List<? extends CharSequence>
        getLongWords(Collection<? extends CharSequence> coll)
```

F.
```
static public <E extends CharSequence> Collection<E>
        getLongWords(Collection<E> coll)
```

G.
```
static public <E super CharSequence> Collection<E>
        getLongWords(Collection<E> coll)
```

15. Given:

```
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. Compilation fails.

B. `one two three four`

C. `four three two one`

D. `four one three two`

E. `one two three four one`

F. `one four three two one`

G. An exception is thrown at runtime.

H. The print order is not guaranteed.

16. Given a method declared as:

```
public static <E extends Number> List<? super E> process(List<E> nums)
```

A programmer wants to use this method like this:

```
// INSERT DECLARATIONS HERE

output = process(input);
```

Which pairs of declarations could be placed at `// INSERT DECLARATIONS HERE` to allow the code to compile? (Choose all that apply.)

A.
```
ArrayList<Integer> input = null;
ArrayList<Integer> output = null;
```

B.
```
ArrayList<Integer> input = null;
List<Integer> output = null;
```

C.
```
ArrayList<Integer> input = null;
List<Number> output = null;
```

D.
```
List<Number> input = null;
ArrayList<Integer> output = null;
```

E.
```
List<Number> input = null;
List<Number> output = null;
```

F.
```
List<Integer> input = null;
List<Integer> output = null;
```

G. None of the above.

SELF TEST ANSWERS

1. Given:

```
import java.util.*;
class Test {
  public static void main(String[] args) {
    // insert code here
    x.add("one");
    x.add("two");
    x.add("TWO");
    System.out.println(x.poll());
  }
}
```

Which, inserted at `// insert code here`, will compile? (Choose all that apply.)

A. `List<String> x = new LinkedList<String>();`

B. `TreeSet<String> x = new TreeSet<String>();`

C. `HashSet<String> x = new HashSet<String>();`

D. `Queue<String> x = new PriorityQueue<String>();`

E. `ArrayList<String> x = new ArrayList<String>();`

F. `LinkedList<String> x = new LinkedList<String>();`

Answer:

☑ **D** and **F** are correct. The `poll()` method is associated with Queues. The LinkedList class implements the Queue interface.

☒ **A** is incorrect because the List interface does not implement Queue, and the polymorphic instantiation would restrict `x` to invoking only those methods declared in the List interface. **B, C,** and **E** are incorrect, based on the above. (Objective 6.3)

2. Given:

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

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

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

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

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

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

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

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

G. None of the above.

Answer:

☑ **B** is correct.

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

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

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

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

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

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

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

Answer:

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

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

4. Given:

```
import java.util.*;
class Flubber {
  public static void main(String[] args) {
    List<String> x = new ArrayList<String>();
    x.add(" x");  x.add("xx"); x.add("Xx");

    // insert code here
   for(String s: x)  System.out.println(s);
  }
}
```

And the output:

```
xx
Xx
 x
```

Which code, inserted at `// insert code here`, will produce the preceding output? (Choose all that apply.)

A. `Collections.sort(x);`

B.
```
Comparable c = Collections.reverse();
    Collections.sort(x,c);
```

C.
```
Comparator c = Collections.reverse();
    Collections.sort(x,c);
```

D.
```
Comparable c = Collections.reverseOrder();
    Collections.sort(x,c);
```

E.
```
Comparator c = Collections.reverseOrder();
    Collections.sort(x,c);
```

Answer:

☑ **E** is correct. Natural ordering would produce output in reverse sequence to that listed. The `Collections.reverseOrder()` method takes a Comparator not a Comparable to re-sort a Collection.

☒ **A, B, C,** and **D** are incorrect based on the above. (Objective 6.5)

5. Given:

```
10.    public static void main(String[] args) {
11.        Queue<String> q = new LinkedList<String>();
12.        q.add("Veronica");
13.        q.add("Wallace");
14.        q.add("Duncan");
15.        showAll(q);
16.    }
```

```
17.
18.     public static void showAll(Queue q) {
19.         q.add(new Integer(42));
20.         while (!q.isEmpty())
21.             System.out.print(q.remove() + " ");
22.     }
```

What is the result?

A. `Veronica Wallace Duncan`
B. `Veronica Wallace Duncan 42`
C. `Duncan Wallace Veronica`
D. `42 Duncan Wallace Veronica`
E. Compilation fails.
F. An exception occurs at runtime.

Answer:

☑ **B** is correct. There is a compiler warning at line 19 because of an unchecked assignment, but other than that everything compiles and runs fine. Although `q` was originally declared as `Queue<String>`, in `showAll()` it's passed as an untyped Queue—nothing in the compiler or JVM prevents us from adding an Integer after that. The `add()` method puts things at the end of the queue, while `remove()` takes them from the beginning, so everything prints in the order they were put in.

☒ **A**, **C, D, E,** and **F** are incorrect based on the above. (Objective 6.3)

6. Given:

```
public static void before() {
  Set set = new TreeSet();
  set.add("2");
  set.add(3);
  set.add("1");
  Iterator it = set.iterator();
    while (it.hasNext())
  System.out.print(it.next() + " ");
}
```

Which of the following statements are true?

A. The `before()` method will print `1 2`
B. The `before()` method will print `1 2 3`
C. The `before()` method will print three numbers, but the order cannot be determined.
D. The `before()` method will not compile.
E. The `before()` method will throw an exception at runtime.

Answer:

☑ **E** is correct. You can't put both Strings and `int`s into the same TreeSet. Without generics, the compiler has no way of knowing what type is appropriate for this TreeSet, so it allows everything to compile. At runtime, the TreeSet will try to sort the elements as they're added, and when it tries to compare an Integer with a String it will throw a ClassCastException. Note that although the `before()` method does not use generics, it does use autoboxing. Watch out for code that uses some new features and some old features mixed together.

☒ **A, B, C,** and **D** are incorrect based on the above. (Objective 6.5)

7. Given:

```
import java.util.*;
class MapEQ {
  public static void main(String[] args) {
    Map<ToDos, String> m = new HashMap<ToDos, String>();
    ToDos t1 = new ToDos("Monday");
    ToDos t2 = new ToDos("Monday");
    ToDos t3 = new ToDos("Tuesday");
    m.put(t1, "doLaundry");
    m.put(t2, "payBills");
    m.put(t3, "cleanAttic");
    System.out.println(m.size());
} }
class ToDos{
  String day;
  ToDos(String d) { day = d; }
  public boolean equals(Object o) {
    return ((ToDos)o).day == this.day;
  }
  // public int hashCode() { return 9; }
}
```

Which is correct? (Choose all that apply.)

A. As the code stands it will not compile.

B. As the code stands the output will be `2`.

C. As the code stands the output will be `3`.

D. If the `hashCode()` method is uncommented the output will be `2`.

E. If the `hashCode()` method is uncommented the output will be `3`.

F. If the `hashCode()` method is uncommented the code will not compile.

Answer:

☑ **C** and **D** are correct. If `hashCode()` is not overridden then every entry will go into its own bucket, and the overridden `equals()` method will have no effect on determining equivalency. If `hashCode()` is overridden, then the overridden `equals()` method will view `t1` and `t2` as duplicates.

☒ **A, B, E,** and **F** are incorrect based on the above. (Objective 6.2)

8. Given:

```
12. public class AccountManager {
13.     private Map accountTotals = new HashMap();
14.     private int retirementFund;
15.
16.     public int getBalance(String accountName) {
17.         Integer total = (Integer) accountTotals.get(accountName);
18.         if (total == null)
19.             total = Integer.valueOf(0);
20.         return total.intValue();
21.     }
23.     public void setBalance(String accountName, int amount) {
24.         accountTotals.put(accountName, Integer.valueOf(amount));
25.  }  }
```

This class is to be updated to make use of appropriate generic types, with no changes in behavior (for better or worse). Which of these steps could be performed? (Choose three.)

A. Replace line 13 with
```
private Map<String, int> accountTotals = new HashMap<String, int>();
```

B. Replace line 13 with
```
private Map<String, Integer> accountTotals = new HashMap<String, Integer>();
```

C. Replace line 13 with
```
private Map<String<Integer>> accountTotals = new HashMap<String<Integer>>();
```

D. Replace lines 17–20 with
```
int total = accountTotals.get(accountName);
if (total == null)  total = 0;
return total;
```

E. Replace lines 17–20 with
```
Integer total = accountTotals.get(accountName);
if (total == null)  total = 0;
return total;
```

F. Replace lines 17–20 with
```
return accountTotals.get(accountName);
```

G. Replace line 24 with
```
accountTotals.put(accountName, amount);
```

H. Replace line 24 with

```
accountTotals.put(accountName, amount.intValue());
```

Answer:

☑ **B, E,** and **G** are correct.

☒ **A** is wrong because you can't use a primitive type as a type parameter. **C** is wrong because a Map takes two type parameters separated by a comma. **D** is wrong because an int can't autobox to a null, and **F** is wrong because a `null` can't unbox to 0. **H** is wrong because you can't autobox a primitive just by trying to invoke a method with it. (Objective 6.4)

9. Given a properly prepared String array containing five elements, which range of results could a proper invocation of `Arrays.binarySearch()` produce?

A. 0 through 4
B. 0 through 5
C. -1 through 4
D. -1 through 5
E. -5 through 4
F. -5 through 5
G. -6 through 4
H. -6 through 5

Answer:

☑ **G** is correct. If a match is found, `binarySearch()` will return the index of the element that was matched. If no match is found, `binarySearch()` will return a negative number that, if inverted and then decremented, gives you the insertion point (array index) at which the value searched on should be inserted into the array to maintain a proper sort.

☒ **A, B, C, D, E, F,** and **H** are incorrect based on the above. (Objective 6.5)

10. Given:

```
interface Hungry<E> { void munch(E x); }
interface Carnivore<E extends Animal> extends Hungry<E> {}
interface Herbivore<E extends Plant> extends Hungry<E> {}
abstract class Plant {}
class Grass extends Plant {}
abstract class Animal {}
class Sheep extends Animal implements Herbivore<Sheep> {
  public void munch(Sheep x) {}
}
class Wolf extends Animal implements Carnivore<Sheep> {
  public void munch(Sheep x) {}
}
```

Which of the following changes (taken separately) would allow this code to compile? (Choose all that apply.)

A. Change the Carnivore interface to
```
interface Carnivore<E extends Plant> extends Hungry<E> {}
```
B. Change the Herbivore interface to
```
interface Herbivore<E extends Animal> extends Hungry<E> {}
```
C. Change the Sheep class to
```
class Sheep extends Animal implements Herbivore<Plant> {
    public void munch(Grass x) {}
}
```
D. Change the Sheep class to
```
class Sheep extends Plant implements Carnivore<Wolf> {
    public void munch(Wolf x) {}
}
```
E. Change the Wolf class to
```
class Wolf extends Animal implements Herbivore<Grass> {
    public void munch(Grass x) {}
}
```
F. No changes are necessary.

Answer:

☑ **B** is correct. The problem with the original code is that Sheep tries to implement `Herbivore<Sheep>` and Herbivore declares that its type parameter `E` can be any type that extends Plant. Since a Sheep is not a Plant, `Herbivore<Sheep>` makes no sense—the type Sheep is outside the allowed range of Herbivore's parameter `E`. Only solutions that either alter the definition of a Sheep or alter the definition of Herbivore will be able to fix this. So **A, E,** and **F** are eliminated. **B** works, changing the definition of an Herbivore to allow it to eat Sheep solves the problem. **C** doesn't work because an `Herbivore<Plant>` must have a `munch(Plant)` method, not `munch(Grass)`. And **D** doesn't work, because in **D** we made Sheep extend Plant, now the Wolf class breaks because its `munch(Sheep)` method no longer fulfills the contract of Carnivore. (Objective 6.4)

11. Which collection class(es) allows you to grow or shrink its size and provides indexed access to its elements, but whose methods are not synchronized? (Choose all that apply.)

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

Answer:

☑ **D** is correct. 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, E,** and **F** are incorrect based on the logic described above; Notes: **C**, List is an interface, and **F**, `PriorityQueue` does not offer access by index. (Objective 6.1)

12. Given:

```
import java.util.*;
public class Group extends HashSet<Person> {
    public static void main(String[] args) {
        Group g = new Group();
        g.add(new Person("Hans"));
        g.add(new Person("Lotte"));
        g.add(new Person("Jane"));
        g.add(new Person("Hans"));
        g.add(new Person("Jane"));
        System.out.println("Total: " + g.size());
    }
    public boolean add(Object o) {
        System.out.println("Adding: " + o);
        return super.add(o);
    }
}
class Person {
    private final String name;
    public Person(String name) { this.name = name; }
    public String toString() { return name; }
}
```

Which of the following occur at least once when the code is compiled and run? (Choose all that apply.)

A. Adding `Hans`
B. Adding `Lotte`
C. Adding `Jane`
D. Total: 3
E. Total: 5
F. The code does not compile.
G. An exception is thrown at runtime.

Answer:

☑ **F** is correct. The problem here is in Group's `add()` method—it should have been `add(Person)`, since the class extends `HashSet<Person>`. So this doesn't compile. Pop Quiz: What would happen if you fixed this code, changing `add(Object)` to `add(Person)`? Try running the code to see if the results match what you thought.

☒ **A, B, C, D, E,** and **G** are incorrect based on the above. (Objective 6.4)

13. Given:

```
import java.util.*;
class AlgaeDiesel {
  public static void main(String[] args) {
    String[] sa = {"foo", "bar", "baz" };
    // insert method invocations here
  }
}
```

What `java.util.Arrays` and/or `java.util.Collections` methods could you use to convert `sa` to a List and then search the List to find the index of the element whose value is `"foo"`? (Choose from one to three methods.)

A. `sort()`

B. `asList()`

C. `toList()`

D. `search()`

E. `sortList()`

F. `contains()`

G. `binarySearch()`

Answer:

☑ **A, B,** and **G** are required. The as `List()` method converts an array to a List. You can find the index of an element in a List with the `binarySearch()` method, but before you do that you must sort the list using `sort()`.

☒ **F** is incorrect because `contains()` returns a boolean, not an index. **C, D,** and **E** are incorrect, because these methods are not defined in the List interface. (Objective 6.5)

14. Given that String implements java.lang.CharSequence, and:

```
import java.util.*;
public class LongWordFinder {
    public static void main(String[] args) {
        String[] array = { "123", "12345678", "1", "12", "1234567890"};
        List<String> list = Arrays.asList(array);
        Collection<String> resultList = getLongWords(list);
    }
    // INSERT DECLARATION HERE
    {
        Collection<E> longWords = new ArrayList<E>();
        for (E word : coll)
            if (word.length() > 6)  longWords.add(word);
        return longWords;
}   }
```

Which declarations could be inserted at `// INSERT DECLARATION HERE` so that the program will compile and run? (Choose all that apply.)

A. `public static <E extends CharSequence> Collection<? extends CharSequence> getLongWords(Collection<E> coll)`

B. `public static <E extends CharSequence> List<E> getLongWords(Collection<E> coll)`

C. `public static Collection<E extends CharSequence> getLongWords(Collection<E> coll)`

D. `public static List<CharSequence> getLongWords(Collection<CharSequence> coll)`

E. `public static List<? extends CharSequence> getLongWords(Collection<? extends CharSequence> coll)`

F. `static public <E extends CharSequence> Collection<E> getLongWords(Collection<E> coll)`

G. `static public <E super CharSequence> Collection<E> getLongWords(Collection<E> coll)`

Answer:

☑ **F** is correct.

☒ **A** is close, but it's wrong because the return value is too vague. The last line of the method expects the return value to be `Collection<String>`, not `Collection<? extends CharSequence>`. **B** is wrong because `longWords` has been declared as a `Collection<E>`, and that can't be implicitly converted to a `List<E>` to match the declared return value. (Even though we know that `longWords` is really an `ArrayList<E>`, the compiler only know what it's been declared as.) **C, D,** and **E** are wrong because they do not declare a type variable E (there's no <> before the return value) so the

`getLongWords()` method body will not compile. **G** is wrong because `E super CharSequence` makes no sense—`super` could be used in conjunction with a wildcard but not a type variable like `E`. (Objective 6.3)

15. Given:

```
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. Compilation fails.

B. `one two three four`

C. `four three two one`

D. `four one three two`

E. `one two three four one`

F. `one four three two one`

G. An exception is thrown at runtime.

H. The print order is not guaranteed.

Answer:

☑ **D** is correct. TreeSet assures no duplicate entries; also, when it is accessed it will return elements in natural order, which for Strings means alphabetical.

☒ **A, B, C, E, F, G,** and **H** are incorrect based on the logic described above. Note, even though as of Java 5 you don't have to use an Iterator, you still can. (Objective 6.5)

16. Given a method declared as:

```
public static <E extends Number> List<? super E> process(List<E> nums)
```

A programmer wants to use this method like this:

```
// INSERT DECLARATIONS HERE
output = process(input);
```

Which pairs of declarations could be placed at `// INSERT DECLARATIONS HERE` to allow the code to compile? (Choose all that apply.)

A.
```
ArrayList<Integer> input = null;
ArrayList<Integer> output = null;
```

B.
```
ArrayList<Integer> input = null;
List<Integer> output = null;
```

C.
```
ArrayList<Integer> input = null;
List<Number> output = null;
```

D.
```
List<Number> input = null;
ArrayList<Integer> output = null;
```

E.
```
List<Number> input = null;
List<Number> output = null;
```

F.
```
List<Integer> input = null;
List<Integer> output = null;
```

G. None of the above.

Answer:

☑ **B, E,** and **F** are correct.

☒ The return type of process is definitely declared as a `List`, not an `ArrayList`, so **A** and **D** are wrong. **C** is wrong because the return type evaluates to `List<Integer>`, and that can't be assigned to a variable of type `List<Number>`. Of course all these would probably cause a `NullPointerException` since the variables are still null—but the question only asked us to get the code to compile.