

Remember, when an object is constructed using `new` (as opposed to being deserialized), the following things happen (in this order):

1. All instance variables are assigned default values.
2. The constructor is invoked, which immediately invokes the superclass constructor (or another overloaded constructor, until one of the overloaded constructors invokes the superclass constructor).
3. All superclass constructors complete.
4. Instance variables that are initialized as part of their declaration are assigned their initial value (as opposed to the default values they're given prior to the superclass constructors completing).
5. The constructor completes.

*But these things do not happen when an object is serialized.* When an instance of a serializable class is serialized, the constructor does not run and instance variables are not given their initially assigned values! Think about it—if the constructor were invoked and/or instance variables were assigned the values given in their declarations, the object you’re trying to restore would revert back to its original state, rather than coming back reflecting the changes in its state that happened sometime after it was created. For example, imagine you have a class that declares an instance variable and assigns it the `int` value `3` and includes a method that changes the instance variable value to `10`:

```
class Foo implements Serializable {  
    int num = 3;  
    void changeNum() { num = 10; }  
}
```

Obviously, if you serialize a `Foo` instance *after* the `changeNum()` method runs, the value of the `num` variable should be `10`. When the `Foo` instance is serialized, you want the `num` variable to still be `10`! You obviously don’t want the initialization (in this case, the assignment of the value `3` to the variable `num`) to happen. Think of constructors and instance variable assignments together as part of one complete object initialization process (and, in fact, they do become one initialization method in the bytecode). The point is, when an object is serialized we do not want any of the normal initialization to happen. We don’t

want the constructor to run, and we don't want the explicitly declared values to be assigned. We want only the values saved as part of the serialized state of the object to be reassigned.

Of course, if you have variables marked `transient`, they will not be restored to their original state (unless you implement `readObject()`), but will instead be given the default value for that data type. In other words, even if you say

```
class Bar implements Serializable {  
    transient int x = 42;  
}
```

when the `Bar` instance is deserialized, the variable `x` will be set to a value of `0`. Object references marked `transient` will always be reset to `null`, regardless of whether they were initialized at the time of declaration in the class.

So, that's what happens when the object is deserialized, and the class of the serialized object directly extends `Object`, or has only serializable classes in its inheritance tree. It gets a little trickier when the serializable class has one or more non-serializable superclasses.

Getting back to our non-serializable `Animal` class with a serializable `Dog` subclass example:

```
class Animal {  
    public String name;  
}  
class Dog extends Animal implements Serializable {  
    // the rest of the Dog code  
}
```

Because `Animal` is not serializable, any state maintained in the `Animal` class, even though the state variable is inherited by the `Dog`, isn't going to be restored with the `Dog` when it's deserialized! The reason is, the (unserialized) `Animal` part of the `Dog` is going to be reinitialized, just as it would be if you were making a new `Dog` (as opposed to deserializing one). That means all the things that happen to an object during construction will happen—but only to the `Animal` parts of a `Dog`. In other words, the instance variables from the `Dog`'s class will be serialized

and deserialized correctly, but the inherited variables from the non-serializable Animal superclass will come back with their default/initially assigned values rather than the values they had at the time of serialization.

If you are a serializable class but your superclass is *not* serializable, then any instance variables you inherit from that superclass will be reset to the values they were given during the original construction of the object. This is because the non-serializable class constructor *will* run!

In fact, every constructor above the first non-serializable class constructor will also run, no matter what, because once the first super constructor is invoked (during deserialization), it, of course, invokes its super constructor and so on, up the inheritance tree.

For the exam, you'll need to be able to recognize which variables will and will not be restored with the appropriate values when an object is deserialized, so be sure to study the following code example and the output:

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

        Dog d = new Dog(35, "Fido");
        System.out.println("before: " + d.name + " "
                           + d.weight);
        try {
            FileOutputStream fs = new FileOutputStream("testSer.ser");
            ObjectOutputStream os = new ObjectOutputStream(fs);
            os.writeObject(d);
            os.close();
        } catch (Exception e) { e.printStackTrace(); }
        try {
            FileInputStream fis = new FileInputStream("testSer.ser");
            ObjectInputStream ois = new ObjectInputStream(fis);
            d = (Dog) ois.readObject();
            ois.close();
        } catch (Exception e) { e.printStackTrace(); }

        System.out.println("after: " + d.name + " "
                           + d.weight);
    }
}
class Dog extends Animal implements Serializable {
    String name;
    Dog(int w, String n) {
        weight = w;           // inherited
        name = n;             // not inherited
    }
}
class Animal {           // not serializable !
    int weight = 42;
}
```

which produces the output:

before: Fido 35

after: Fido 42

The key here is that because `Animal` is not serializable, when the `Dog` was deserialized, the `Animal` constructor ran and reset the `Dog`'s inherited `weight` variable.



*If you serialize a collection or an array, every element must be serializable! A single non-serializable element will cause serialization to fail. Note also that although the collection interfaces are not serializable, the concrete collection classes in the Java API are.*

## Serialization Is Not for Statics

Finally, you might have noticed that we've talked only about instance variables, not static variables. Should static variables be saved as part of the object's state? Isn't the state of a static variable at the time an object was serialized important? Yes and no. It might be important, but it isn't part of the instance's state at all. Remember, you should think of static variables purely as *class* variables. They have nothing to do with individual instances. But serialization applies only to *objects*. And what happens if you deserialize three different `Dog` instances, all of which were serialized at different times and all of which were saved when the value of a static variable in class `Dog` was different? Which instance would "win"? Which instance's static value would be used to replace the one currently in the one and only `Dog` class that's currently loaded? See the problem?

Static variables are *never* saved as part of the object's state...because they do not belong to the object!



**As simple as serialization code is to write, versioning problems can occur in the real world. If you save a *Dog* object using one version of the class, but attempt to deserialize it using a newer different version of the class, deserialization might fail. See the Java API for details about versioning issues and solutions.**

## CERTIFICATION SUMMARY

---

**File I/O** Remember that objects of type `File` can represent either files or directories, but that until you call `createNewFile()` or `mkdir()`, you haven't actually created anything on your hard drive. Classes in the `java.io` package are designed to be chained together. You will rarely use a `FileReader` or a `FileWriter` without "wrapping" them with a `BufferedReader` or `BufferedWriter` object, which gives you access to more powerful, higher-level methods. As of Java 5, the `PrintWriter` class has been enhanced with advanced `append()`, `format()`, and `printf()` methods, and when you couple that with new constructors that allow you to create `PrintWriters` directly from a `String` name or a `File` object, you may use `BufferedWriters` a lot less. The `Console` class allows you to read nonechoed input (returned in a `char[?]`) and is instantiated using `System.console()`.

NIO.2 objects of type `Path` can be files or directories and are a replacement of type `File`. `Paths` are created with `Paths.get()`. Utility methods in `Files` allow you to create, delete, move, copy, or check information about a `Path`. In addition, `BasicFileAttributes`, `DosFileAttributes` (Windows), and `PosixFileAttributes` (UNIX/Linux/Mac) allow you to check more advanced information about a `Path`. `BasicFileAttributeView`, `DosFileAttributeView`, and `PosixFileAttributeView` allow you to update advanced `Path` attributes.

Using a `DirectoryStream` allows you to iterate through a directory. Extending `SimpleFileVisitor` lets you walk a directory tree recursively looking at files and/or directories. With a `PathMatcher`, you can search directories for files using regex-esque expressions called globs.

Finally, registering a `WatchService` provides notifications for new/changed/removed files or directories.

**Serialization** Serialization lets you save, ship, and restore everything you need to know about a *live* object. And when your object points to other objects, they get saved too. The `java.io.ObjectOutputStream` and

`java.io.ObjectInputStream` classes are used to serialize and deserialize objects. Typically, you wrap them around instances of `FileOutputStream` and `FileInputStream`, respectively.

The key method you invoke to serialize an object is `writeObject()`, and to deserialize an object invoke `readObject()`. In order to serialize an object, it must implement the `Serializable` interface. Mark instance variables `transient` if you don't want their state to be part of the serialization process. You can augment the serialization process for your class by implementing `writeObject()` and `readObject()`. If you do that, an embedded call to `defaultReadObject()` and `defaultWriteObject()` will handle the normal serialization tasks, and you can augment those invocations with manual *reading from* and *writing to* the stream.

If a superclass implements `Serializable` then all of its subclasses do too. If a superclass doesn't implement `Serializable`, then when a subclass object is deserialized, the unserializable superclass's constructor runs—be careful! Finally, remember that serialization is about instances, so static variables aren't serialized.



## TWO-MINUTE DRILL

Here are some of the key points from the certification objectives in this chapter.

### File I/O (OCP Objectives 8.1 and 8.2)

- The classes you need to understand in `java.io` are `File`, `FileReader`, `BufferedReader`, `FileWriter`, `BufferedWriter`, `PrintWriter`, and `Console`.
- A new `File` object doesn't mean there's a new file on your hard drive.
- `File` objects can represent either a file or a directory.
- The `File` class lets you manage (add, rename, and delete) files and directories.
- The methods `createNewFile()` and `mkdir()` add entries to your file system.
- `FileWriter` and `FileReader` are low-level I/O classes. You can use them to write and read files, but they should usually be wrapped.

- `FileOutputStream` and `FileInputStream` are low-level I/O classes. You can use them to write and read bytes to and from files, but they should usually be wrapped.
- Classes in `java.io` are designed to be “chained” or “wrapped.” (This is a common use of the decorator design pattern.)
- It’s very common to “wrap” a `BufferedReader` around a `FileReader` or a `BufferedWriter` around a `FileWriter` to get access to higher-level (more convenient) methods.
- `Printwriters` can be used to wrap other `Writers`, but as of Java 5, they can be built directly from `Files` or `Strings`.
- As of Java 5, `Printwriters` have `append()`, `format()`, and `printf()` methods.
- `Console` objects can read nonechoed input and are instantiated using `System.console()`.

## Path, Paths, File, and Files (OCP Objectives 9.1 and 9.2)

- NIO.2 was introduced in Java 7.
- `Path` replaces `File` for a representation of a file or directory.
- `Paths.get()` lets you create a `Path` object.
- Static methods in `Files` let you work with `Path` objects.
- A `Path` object doesn’t mean the file or directory exists on your hard drive.
- The methods `Files.createFile()` and `Files.createDirectory()` add entries to your file system.
- The `Files` class provides methods to move, copy, and delete `Path` objects.
- `Files.delete()` throws an exception and `Files.deleteIfExists()` returns false if the file does not exist.
- On `Path`, `normalize()` simplifies the path representation.
- On `Path`, `resolve()` and `relativize()` work with the relationship between two path objects.

## File Attributes (OCP Objective 9.2)

- The `Files` class provides methods for common attributes, such as whether the file is executable and when it was last modified.
- For less common attributes the classes `BasicFileAttributes`, `DosFileAttributes`, and `PosixFileAttributes` read the attributes.
- `DosFileAttributes` works on Windows operating systems.
- `PosixFileAttributes` works on UNIX, Linux, and Mac operating systems.
- Attributes that can't be updated via the `Files` class are set using these classes: `BasicFileAttributeView`, `DosFileAttributeView`, `PosixFileAttributeView`, `FileOwnerAttributeView`, and `AclFileAttributeView`.

## Directory Trees, Matching, and Watching for Changes (OCP Objective 9.2)

- `DirectoryStream` iterates through immediate children of a directory using glob patterns.
- `FileVisitor` walks recursively through a directory tree.
- You can override one or all of the methods of `SimpleFileVisitor`—`preVisitDirectory`, `visitFile`, `visitFileFailed`, and `postVisitDirectory`.
- You can change the flow of a file visitor by returning one of the `FileVisitResult` constants: `CONTINUE`, `SKIP_SUBTREE`, `SKIP_SIBLINGS`, or `TERMINATE`.
- `PathMatcher` checks if a path matches a glob pattern.
- Know what the following expressions mean for globs: `*`, `**`, `?`, and `{a, b}`.
- Directories register with `WatchService` to be notified about creation, deletion, and modification of files or immediate subdirectories.
- `PathMatcher` and `WatchService` use `FileSystems`-specific implementations.

## Serialization (Objective 8.2)

- The classes you need to understand are all in the `java.io` package; they

include `ObjectOutputStream` and `ObjectInputStream`, primarily, and `FileOutputStream` and `FileInputStream` because you will use them to create the low-level streams that the `ObjectXxxStream` classes will use.

- A class must implement `Serializable` before its objects can be serialized.
- The `ObjectOutputStream.writeObject()` method serializes objects, and the `ObjectInputStream.readObject()` method deserializes objects.
- If you mark an instance variable `transient`, it will not be serialized even though the rest of the object's state will be.
- You can supplement a class's automatic serialization process by implementing the `writeObject()` and `readObject()` methods. If you do this, embedding calls to `defaultWriteObject()` and `defaultReadObject()`, respectively, will handle the part of serialization that happens normally.
- If a superclass implements `Serializable`, then its subclasses do automatically.
- If a superclass doesn't implement `Serializable`, then, when a subclass object is deserialized, the superclass constructor will be invoked along with its superconstructor(s).

## SELF TEST

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

1. Note: The use of “drag-and-drop” questions has come and gone over the years. In case Oracle brings them back into fashion, we threw a couple of them in the book.

Using the fewest fragments possible (and filling the fewest slots possible), complete the following code so that the class builds a directory named “dir3” and creates a file named “file3” inside “dir3.” Note you can use each fragment either zero or one times.

Code:

```
import java.io.
```

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

} }

## Fragments:

```
File;      FileDescriptor;    FileWriter;     Directory;
try {      .createNewDir();      File dir        File
{ }        (Exception x)       ("dir3");      file
file       .createNewFile();    = new File      = new File
dir        (dir, "file3");    (dir, file);   .createFile();
} catch   ("dir3", "file3"); .mkdir();        File file
```

2. Given:

```
import java.io.*;

class Directories {
    static String [] dirs = {"dir1", "dir2"};
    public static void main(String [] args) {
        for (String d : dirs) {

            // insert code 1 here

            File file = new File(path, args[0]);

            // insert code 2 here
        }
    }
}
```

and that the invocation

```
java Directories file2.txt
```

is issued from a directory that has two subdirectories, “dir1” and “dir2,” and that “dir1” has a file “file1.txt” and “dir2” has a file “file2.txt,” and the output is “false true,” which set(s) of code fragments must be inserted? (Choose all that apply.)

- A. String path = d;  
System.out.print(file.exists() + " ");
- B. String path = d;  
System.out.print(file.isFile() + " ");
- C. String path = File.separator + d;  
System.out.print(file.exists() + " ");
- D. String path = File.separator + d;  
System.out.print(file.isFile() + " ");

3. Given:

```
import java.io.*;
public class ReadingFor {
    public static void main(String[] args) {
        String s;
        try {
            FileReader fr = new FileReader("myfile.txt");
            BufferedReader br = new BufferedReader(fr);
            while((s = br.readLine()) != null)
                System.out.println(s);
            br.flush();
        } catch (IOException e) { System.out.println("io error"); }
    }
}
```

And given that `myfile.txt` contains the following two lines of data:

`ab`

`cd`

What is the result?

- A. ab
- B. abcd
- C. ab  
cd
- D. a  
b  
c  
d

## E. Compilation fails

4. Given:

```
1. import java.io.*;  
2. public class Talker {  
3.     public static void main(String[] args) {  
4.         Console c = System.console();  
5.         String u = c.readLine("%s", "username: ");  
6.         System.out.println("hello " + u);  
7.         String pw;  
8.         if(c != null && (pw = c.readPassword("%s", "password: ")) != null)  
9.             // check for valid password  
10.    }  
11. }
```

If line 4 creates a valid `Console` object and if the user enters `fred` as a username and `1234` as a password, what is the result? (Choose all that apply.)

A. `username:`

`password:`

B. `username: fred`

`password:`

C. `username: fred`

`password: 1234`

D. Compilation fails

E. An exception is thrown at runtime

5. Given:

```
3. import java.io.*;
4. class Vehicle { }
5. class Wheels { }
6. class Car extends Vehicle implements Serializable { }
7. class Ford extends Car { }
8. class Dodge extends Car {
9.     Wheels w = new Wheels();
10. }
```

Instances of which class(es) can be serialized? (Choose all that apply.)

A. Car

B. Ford

C. Dodge

- D. Wheels
  - E. Vehicle
6. Which of the following creates a Path object pointing to c:/temp/exam? (Choose all that apply.)
- A. new Path("c:/temp/exam")
  - B. new Path("c:/temp", "exam")
  - C. Files.get("c:/temp/exam")
  - D. Files.get("c:/temp", "exam")
  - E. Paths.get("c:/temp/exam")
  - F. Paths.get("c:/temp", "exam")
7. Given a directory tree at the root of the C: drive and the fact that no other files exist:

```
dir x - |
..... | - dir y
..... | - file a
```

and these two paths:

```
Path one = Paths.get("c:/x");
Path two = Paths.get("c:/x/y/a");
```

- Which of the following statements prints out: y/a?
- A. System.out.println(one.relativize(two));
  - B. System.out.println(two.relativize(one));
  - C. System.out.println(one.resolve(two));
  - D. System.out.println(two.resolve(one));
  - E. System.out.println(two.resolve(two));
  - F. None of the above
8. Given the following statements:
- I. A nonempty directory can usually be deleted using Files.delete
  - II. A nonempty directory can usually be moved using Files.move

- III. A nonempty directory can usually be copied using `Files.copy`

Which of the following is true?

- A. I only
- B. II only
- C. III only
- D. I and II only
- E. II and III only
- F. I and III only
- G. I, II, and III

**9.** Given:

```
new File("c:/temp/test.txt").delete();
```

How would you write this line of code using Java 7 APIs?

- A. `Files.delete(Paths.get("c:/temp/test.txt"));`
- B. `Files.deleteIfExists(Paths.get("c:/temp/test.txt"));`
- C. `Files.deleteOnExit(Paths.get("c:/temp/test.txt"));`
- D. `Paths.get("c:/temp/test.txt").delete();`
- E. `Paths.get("c:/temp/test.txt").deleteIfExists();`
- F. `Paths.get("c:/temp/test.txt").deleteOnExit();`

**10.** Given:

```
public void read(Path dir) throws IOException {  
    // CODE HERE  
    System.out.println(attr.creationTime());  
}
```

Which code inserted at // CODE HERE will compile and run without error on Windows? (Choose all that apply.)

- A. BasicFileAttributes attr = Files.readAttributes(dir,  
BasicFileAttributes.class);
  - B. BasicFileAttributes attr = Files.readAttributes(dir,  
DosFileAttributes.class);
  - C. DosFileAttributes attr = Files.readAttributes(dir,  
BasicFileAttributes.class);
  - D. DosFileAttributes attr = Files.readAttributes(dir,  
DosFileAttributes.class);
  - E. PosixFileAttributes attr = Files.readAttributes(dir,  
PosixFileAttributes.class);
  - F. BasicFileAttributes attr = new BasicFileAttributes(dir);
  - G. BasicFileAttributes attr =dir.getBasicFileAttributes();
- 11.** Which of the following are true? (Choose all that apply.)
- A. The class AbstractFileAttributes applies to all operating systems
  - B. The class BasicFileAttributes applies to all operating systems
  - C. The class DosFileAttributes applies to Windows-based operating systems
  - D. The class WindowsFileAttributes applies to Windows-based operating systems
  - E. The class PosixFileAttributes applies to all Linux/UNIX-based operating systems
  - F. The class UnixFileAttributes applies to all Linux/UNIX-based

## operating systems

12. Given a partial directory tree:

```
dir x - |
..... | - dir y
..... | - file a
```

In what order can the following methods be called if walking the directory tree from x? (Choose all that apply.)

- I: preVisitDirectory x
- II: preVisitDirectory x/y
- III: postVisitDirectory x/y
- IV: postVisitDirectory x
- V: visitFile x/a

- A. I, II, III, IV, V
- B. I, II, III, V, IV
- C. I, V, II, III, IV
- D. I, V, II, IV, III
- E. V, I, II, III, IV
- F. V, I, II, IV, III

13. Given:

```
public class MyFileVisitor extends SimpleFileVisitor<Path> {  
    // more code here  
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)  
        throws IOException {  
        System.out.println("File " + file);  
        if (file.getFileName().endsWith("Test.java")) {  
            // CODE HERE  
        }  
        return FileVisitResult.CONTINUE;  
    }  
    // more code here  
}
```

Which code inserted at `// CODE HERE` would cause the `FileVisitor` to stop visiting files after it sees the file `Test.java`?

- A. `return FileVisitResult.CONTINUE;`
- B. `return FileVisitResult.END;`
- C. `return FileVisitResult.SKIP_SIBLINGS;`
- D. `return FileVisitResult.SKIP_SUBTREE;`
- E. `return FileVisitResult.TERMINATE;`
- F. `return null;`

14. Assume all the files referenced by these paths exist:

```
Path a = Paths.get("c:/temp/dir/a.txt");  
Path b = Paths.get("c:/temp/dir/subdir/b.txt");
```

What is the correct string to pass to PathMatcher to match both these files?

- A. "glob:/\*/\*.txt"
- B. "glob:\*\*.txt"
- C. "glob:\*.txt"
- D. "glob:/\*\*/\*.txt"
- E. "glob:/\*\*.txt"
- F. "glob:/\*.txt"
- G. None of the above

15. Given a partial directory tree at the root of the drive:

```
dir x - |
.....| - file a.txt
.....| - dir y
.....|   | - file b.txt
.....|   | - dir y
.....|   | - file c.txt
```

And the following snippet:

```
Path dir = Paths.get("c:/x");
try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir, "**/*.txt")) {
    for (Path path : stream) {
        System.out.println(path);
    }
}
```

What is the result?

- A. c:/x/a.txt
- B. c:/x/a.txt
  - c:/x/y/b.txt
  - c:/x/y/z/c.txt
- C. Code compiles but does not output anything
- D. Does not compile because DirectoryStream comes from FileSystems, not Files
- E. Does not compile for another reason

**16.** Given a partial directory tree:

```
dir x - |
..... | - dir y
..... | -file a
```

and given that a valid Path object, dir, points to x, and given this snippet:

```
WatchKey key = dir.register(watcher, ENTRY_CREATE);
```

If a WatchService is set using the given WatchKey, what would be the result if a file is added to dir y?

- A. No notice is given
- B. A notice related to dir x is issued
- C. A notice related to dir y is issued
- D. Notices for both dir x and dir y are given
- E. An exception is thrown
- F. The behavior depends on the underlying operating system

**17.** Given:

```
import java.io.*;
class Player {
    Player() { System.out.print("p"); }
}
class CardPlayer extends Player implements Serializable {
    CardPlayer() { System.out.print("c"); }
    public static void main(String[] args) {
        CardPlayer c1 = new CardPlayer();
        try {
            FileOutputStream fos = new FileOutputStream("play.txt");
            ObjectOutputStream os = new ObjectOutputStream(fos);
            os.writeObject(c1);
            os.close();
            FileInputStream fis = new FileInputStream("play.txt");
            ObjectInputStream is = new ObjectInputStream(fis);
            CardPlayer c2 = (CardPlayer) is.readObject();
            is.close();
        } catch (Exception x) { }
    }
}
```

What is the result?

- A. pc
- B. pcc
- C. pcp
- D. pcpc
- E. Compilation fails

F. An exception is thrown at runtime

**18.** Given:

```
import java.io.*;  
  
class Keyboard { }  
public class Computer implements Serializable {  
    private Keyboard k = new Keyboard();  
    public static void main(String[] args) {  
        Computer c = new Computer();  
        c.storeIt(c);  
    }  
    void storeIt(Computer c) {  
        try {  
            ObjectOutputStream os = new ObjectOutputStream(  
                new FileOutputStream("myFile"));  
            os.writeObject(c);  
            os.close();  
            System.out.println("done");  
        } catch (Exception x) {System.out.println("exc");}  
    }  
}
```

What is the result? (Choose all that apply.)

- A. exc
- B. done
- C. Compilation fails
- D. Exactly one object is serialized

E. Exactly two objects are serialized

19. Given:

```
import java.io.*;

public class TestSer {
    public static void main(String[] args) {
        SpecialSerial s = new SpecialSerial();
        try {
            ObjectOutputStream os = new ObjectOutputStream(
                new FileOutputStream("myFile"));
            os.writeObject(s);  os.close();
            System.out.print(++s.z + " ");

            ObjectInputStream is = new ObjectInputStream(
                new FileInputStream("myFile"));
            SpecialSerial s2 = (SpecialSerial)is.readObject();
            is.close();
            System.out.println(s2.y + " " + s2.z);
        } catch (Exception x) {System.out.println("exc"); }
    }
}

class SpecialSerial implements Serializable {
    transient int y = 7;
    static int z = 9;
}
```

Which are true? (Choose all that apply.)

- A. Compilation fails
- B. The output is 10 0 9
- C. The output is 10 0 10
- D. The output is 10 7 9
- E. The output is 10 7 10
- F. In order to alter the standard deserialization process, you would implement the `readObject()` method in `SpecialSerial`
- G. In order to alter the standard deserialization process, you would implement the `defaultReadObject()` method in `SpecialSerial`

## A SELF TEST ANSWERS

### 1. Answer:

```
import java.io.File;
class Maker {
    public static void main(String[] args) {
        try {
            File dir = new File("dir3");
            dir.mkdir();
            File file = new File(dir, "file3");
            file.createNewFile();
        } catch (Exception x) { }
    }
}
```

Notes: The `new File` statements don't make actual files or directories, just objects. You need the `mkdir()` and `createNewFile()` methods to actually create the directory and the file. While drag-and-drop questions are no

longer on the exam, it is still good to be able to complete them. (OCP Objective 8.2)

2.  **A** and **B** are correct. Because you are invoking the program from the directory whose direct subdirectories are to be searched, you don't start your path with a `File.separator` character. The `exists()` method tests for either files or directories; the `isFile()` method tests only for files. Since we're looking for a file, both methods work.  
 **C** and **D** are incorrect based on the above. (OCP Objective 8.2)
3.  **E** is correct. You need to call `flush()` only when you're writing data. Readers don't have `flush()` methods. If not for the call to `flush()`, answer **C** would be correct.  
 **A, B, C, and D** are incorrect based on the above. (OCP Objective 8.2)
4.  **D** is correct. The `readPassword()` method returns a `char[]`. If a `char[]` were used, answer **B** would be correct.  
 **A, B, C, and E** are incorrect based on the above. (OCP Objective 8.1)
5.  **A** and **B** are correct. `Dodge` instances cannot be serialized because they "have" an instance of `Wheels`, which is not serializable. `Vehicle` instances cannot be serialized even though the subclass `Car` can be.  
 **C, D, and E** are incorrect based on the above. (Pre-OCPJP 7 only)
6.  **E** and **F** are correct since `Paths` must be created using the `Paths.get()` method. This method takes a varargs `String` parameter, so you can pass as many path segments to it as you like.  
 **A** and **B** are incorrect because you cannot construct a `Path` directly. **C** and **D** are incorrect because the `Files` class works with `Path` objects but does not create them from `Strings`. (Objective 9.1)
7.  **A** is correct because it prints the path to get to two from one.  
 **B** is incorrect because it prints out `../. . .`, which is the path to navigate to one from two. This is the reverse of what we want. **C, D, and E** are incorrect because it does not make sense to call `resolve` with absolute paths. They **might** print out `c:/x/c:/x/y/a`, `c:/x/y/a/c:/x`, and `c:/x/y/a/c:/x/y/a`, respectively. **F** is incorrect because of the above. Note that the directory structure provided is redundant. Neither `relativize()` nor `resolve()` requires either path to actually exist. (OCP Objective 9.1)

8.  **E** is correct because a directory containing files or subdirectories is copied or moved in its entirety. Directories can only be deleted if they are empty. Trying to delete a nonempty directory will throw a `DirectoryNotEmptyException`. The question says “usually” because copy and move success depends on file permissions. Think about the most common cases when encountering words such as “usually” on the exam.
- A, B, C, D, F, and G** are incorrect because of the above. (OCP Objective 9.2)
9.  **B** is correct because, like the Java 7 code, it returns `false` if the file does not exist.
- A** is incorrect because this code throws an exception if the file does not exist. **C, D, E, and F** are incorrect because they do not compile. There is no `deleteOnExit()` method, and file operations such as `delete` occur using the `Files` class rather than the `Path` object directly. (OCP Objective 9.2)
10.  **A, B, and D** are correct. Creation time is a basic attribute, which means you can read `BasicFileAttributes` or any of its subclasses to read it. `DosFileAttributes` is one such subclass.
- C** is incorrect because you cannot cast a more general type to a more specific type. **E** is incorrect because this example specifies it is being run on Windows. Although it would work on UNIX, it throws an `UnsupportedOperationException` on Windows due to requesting the `WindowsFileSystemProvider` to get a POSIX class. **F** and **G** are incorrect because those methods do not exist. You must use the `Files` class to get the attributes. (OCP Objective 9.2)
11.  **B, C, and E** are correct. `BasicFileAttributes` is the general superclass. `DosFileAttributes` subclasses `BasicFileAttributes` for Windows operating systems. `PosixFileAttributes` subclasses `BasicFileAttributes` for UNIX/Linux/Mac operating systems.
- A, D, and F** are incorrect because no such classes exist. (OCP Objective 9.2)
12.  **B and C** are correct because file visitor does a depth-first search. When files and directories are at the same level of the file tree, they can be visited in either order. Therefore, “y” and “a” could be reversed. All of the subdirectories and files are visited before `postVisit` is called on the directory.

**A, D, E, and F** are incorrect because of the above. (OCP Objective 9.2)

- 13.**  **E** is correct because it is the correct constant to end the `FileVisitor`.

**B** is incorrect because `END` is not defined as a result constant. **A, C, and D** are incorrect. Although they are valid constants, they do not end file visiting. `CONTINUE` proceeds as if nothing special has happened.

`SKIP_SUBTREE` skips the subdirectory, which doesn't even make sense for a Java file. `SKIP_SIBLINGS` would skip any files in the same directory. Since we weren't told what the file structure is, we can't assume there weren't other directories or subdirectories. Therefore, we have to choose the most general answer of `TERMINATE`. **F** is incorrect because file visitor throws a `NullPointerException` if null is returned as the result. (OCP Objective 9.2)

- 14.**  **B** is correct. `**` matches zero or more characters, including multiple directories.

**A** is incorrect because `*/` only matches one directory. It will match "temp" but not "c:/temp," let alone "c:/temp/dir." **C** is incorrect because `*.txt` only matches filenames and not directory paths. **D, E, and F** are incorrect because the paths we want to match do not begin with a slash. **G** is incorrect because of the above. (OCP Objective 9.2)

- 15.**  **C** is correct because `DirectoryStream` only looks at files in the immediate directory. `**/* .txt` means zero or more directories followed by a slash, followed by zero or more characters followed by `.txt`. Since the slash is in there, it is required to match, which makes it mean one or more directories. However, this is impossible because `DirectoryStream` only looks at one directory. If the expression were simply `*.txt`, answer **A** would be correct.

**A, B, D, and E** are incorrect because of the above. (OCP Objective 9.2).

- 16.**  **A** is correct because `WatchService` only looks at a single directory. If you want to look at subdirectories, you need to set recursive watch keys. This is usually done using a `FileVisitor`.

**B, C, D, E, and F** are incorrect because of the above. (OCP Objective 9.2).

- 17.**  **C** is correct. It's okay for a class to implement `Serializable` even if its superclass doesn't. However, when you deserialize such an object, the non-serializable superclass must run its constructor. Remember, constructors don't run on serialized classes that implement `Serializable`.

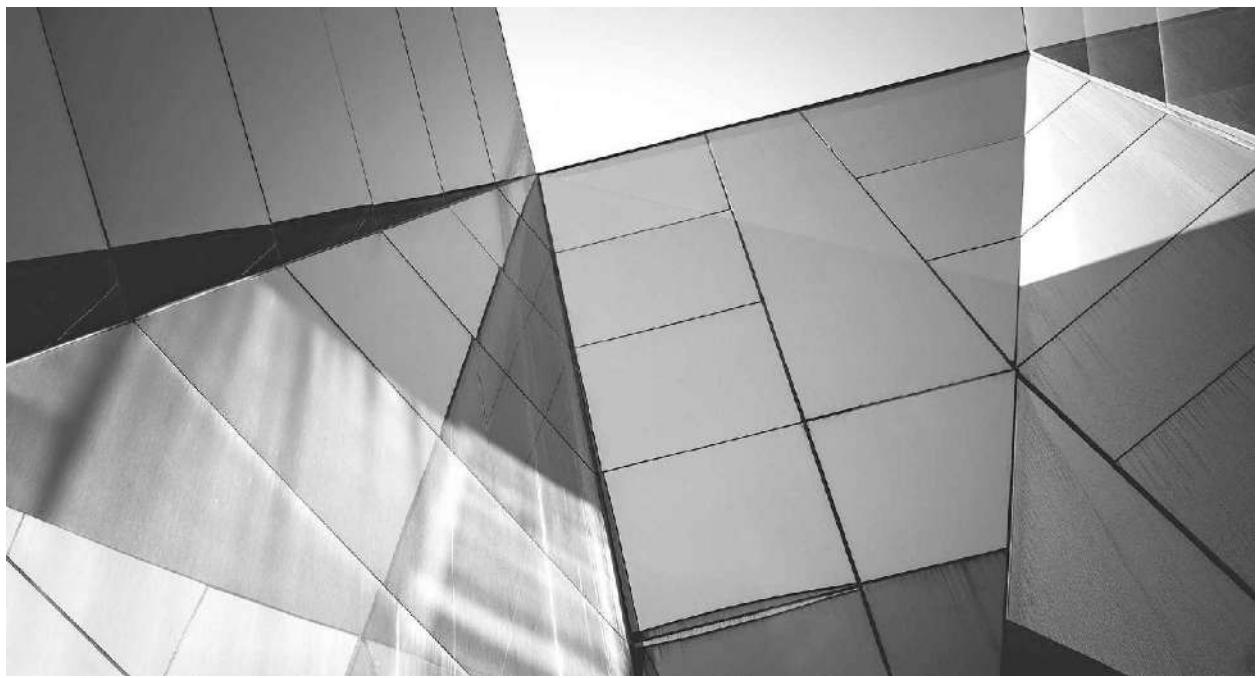
**A, B, D, E, and F** are incorrect based on the above. (OCP Objective 8.2)

- 18.**  **A** is correct. An instance of type Computer Has-a Keyboard. Because Keyboard doesn't implement Serializable, any attempt to serialize an instance of Computer will cause an exception to be thrown.

**B, C, D, and E** are incorrect based on the above. If Keyboard did implement Serializable, then two objects would have been serialized. (OCP Objective 8.2)

- 19.**  **C and F** are correct. **C** is correct because static and transient variables are not serialized when an object is serialized. **F** is a valid statement.

**A, B, D, and E** are incorrect based on the above. **G** is incorrect because you don't implement the defaultReadObject() method; you call it from within the readObject() method, along with any custom read operations your class needs. (OCP Objective 8.2)



# 6

## Generics and Collections

### CERTIFICATION OBJECTIVES

- Override hashCode, equals, and toString Methods from Object Class
  - Create and Use a Generic Class
  - Create and Use ArrayList, TreeSet, TreeMap, and ArrayDeque Objects
  - Use java.util.Comparator and java.lang.Comparable Interfaces
  - Create and Use Lambda Expressions
- ✓ Two-Minute Drill

#### Q&A Self Test

Generics were 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. We’ll also spend time looking at Java’s rich set of classes that allow you to create collections of objects—you know, lists, sets, maps, and queues. It’s safe to say that the care and feeding of collections of data is one of the most common programming activities that programmers perform, and the Java API provides a rich and powerful set of classes dedicated to collections.

### CERTIFICATION OBJECTIVE

## Override hashCode(), equals(), and toString() (OCP Objective 1.4)

### 1.4 Override hashCode, equals, and toString methods from Object class.

It might not be immediately obvious, but understanding `hashCode()` and `equals()` is essential to working with Java collections, especially when using Maps and when searching and sorting in general.

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 class `Object`, but you will need to know about the methods listed in [Table 6-1](#).

**TABLE 6-1** Methods of Class Object Covered on the Exam

Method	Description
<code>boolean equals (Object obj)</code>	Decides whether two objects are meaningfully equivalent
<code>void finalize()</code>	Called by the garbage collector when the garbage collector sees that the object cannot be referenced (rarely used, and deprecated in Java 9)
<code>int hashCode()</code>	Returns a hashcode int value for an object so that the object can be used in Collection classes that use hashing, including <code>Hashtable</code> , <code>HashMap</code> , and <code>HashSet</code>
<code>final void notify()</code>	Wakes up a thread that is waiting for this object's lock
<code>final void notifyAll()</code>	Wakes up <i>all</i> threads that are waiting for this object's lock
<code>final void wait()</code>	Causes the current thread to wait until another thread calls <code>notify()</code> or <code>notifyAll()</code> on this object
<code>String toString()</code>	Returns a "text representation" of the object

[Chapter 10](#) covers `wait()`, `notify()`, and `notifyAll()`. The `finalize()` method was covered in [Chapter 3](#) of the *OCA Java SE 8 Programmer 1 Exam Guide* (McGraw-Hill Education, 2017). In this section, we'll look at the `hashCode()` and `equals()` methods because they are so often critical when using collections. Oh, that leaves `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. When you pass an object reference to the `System.out.println()` method, for example, the object's `toString()` method is called, and the return of `toString()` is shown in the following example:

```
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()

As we mentioned earlier, you might be wondering why we decided to talk about `Object.equals()` near the beginning of the chapter on collections. We'll be spending a lot of time answering that question over the next pages, but for now, it's enough to know that whenever you need to sort or search through a collection of objects, the `equals()` and `hashCode()` methods are essential. But before we go there, let's look at the more common uses of the `equals()` method.

You learned a bit about the `equals()` method in [Chapter 4](#) of the *OCA Java SE 8 Programmer 1 Exam Guide* (McGraw-Hill Education, 2017). 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 has overridden the `equals()` method (inherited from the class `Object`), so you could compare two different `String` objects to see if their contents are meaningfully equivalent. Later in this chapter, we'll be discussing the so-called wrapper classes when it's time to put primitive values into collections. For now, remember that there is a wrapper class for every kind of primitive. The folks who created the `Integer` class (to support `int` primitives) decided that 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. In the case of class `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 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 perform a search based on that object as the key. Of course, overriding the `equals()` method for car also allows the potential for more than one object representing a single unique car to 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 you could cast the object argument to the correct type so 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 `getMoofValue()` 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.



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

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

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

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

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

***knows about this new, different method that happens to also be named equals().***

## 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 implementation details of how the collection classes use hashing, 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.

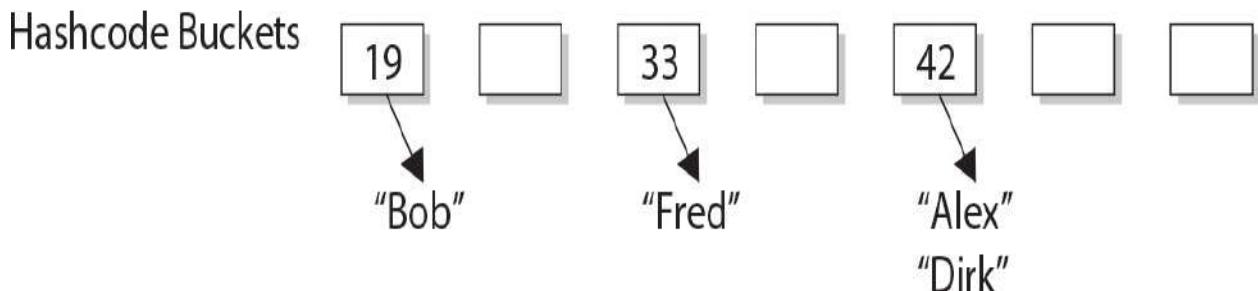
Imagine a set of buckets lined up on the floor. Someone hands you a piece of paper with a name on it. You take the name and calculate an integer code from it by using A is 1, B is 2, and so on, adding the numeric values of all the letters in the name together. A given name will always result in the same code; see [Figure 6-1](#).

### FIGURE 6-1

A simplified hashcode example

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

## HashMap Collection



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 which bucket you should look in 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 and not how to locate the name once we're in that bucket.

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 flip 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.



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

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

This distributed-across-the-buckets example is similar to the way hashcodes are used in collections. When you put an object in a collection that uses hashcodes, the collection uses the hashcode of the object to decide in which bucket/slot the object should land. Then when you want to fetch that object (or, for a hashtable, retrieve the associated value for that object), you have to give the collection a reference to an object, which it then 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.



**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; }
```

**This does 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 with a value of -920. This hashCode() method is horribly inefficient, remember, because it makes all objects land in the same bucket. 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 both have identical `hashCode` values and return `true` for the `equals()` method. 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 (drumroll) 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 that 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)
<code>x.equals(y) == true</code>	<code>x.hashCode() == y.hashCode()</code>	
<code>x.hashCode() == y.hashCode()</code>		<code>x.equals(y) == true</code>
<code>x.equals(y) == false</code>		No <code>hashCode()</code> requirements
<code>x.hashCode() != y.hashCode()</code>	<code>x.equals(y) == false</code>	

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? Although that's legal (the compiler won't complain), under some circumstances, an object you put in a collection won't be found. As you might know, serialization saves an object so it can be reanimated later by deserializing it back to full objectness. But danger, Will Robinson—**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 serialization, `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 them to determine hashcodes or equality.

## CERTIFICATION OBJECTIVE

### Collections Overview (OCP Objective 3.2)

3.2 *Create and use ArrayList, TreeSet, TreeMap, and ArrayDeque objects.*

In this section, we're going to present a relatively high-level discussion of the major categories of collections covered on the exam. We'll be looking at their characteristics and uses from an abstract level. In the section after this one, we'll dive into each category of collection and show concrete examples of using each.

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; there must be 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 and yet again in Java 6, 7, and 8, 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 good performance. And when you need something a little more custom, the Collections Framework in the `java.util` package is loaded with interfaces and utilities.

## So What Do You Do with a Collection?

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

- Add objects to the collection.
- Remove objects from the collection.
- Find out if an object (or group of objects) is in the collection.
- Retrieve an object from the collection without removing it.
- Iterate through the collection, looking at each element (object) one after another.

## Key Interfaces and Classes of the Collections Framework

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 for life in general) are the following nine:

Collection	Set	SortedSet
List	Map	SortedMap
Queue	NavigableSet	NavigableMap

In [Chapter 11](#), which deals with concurrency, we will discuss several classes related to the `Deque` interface. Other than those, there are 14 concrete implementation classes you need to know for the exam (there are others, but the exam doesn't specifically cover them).

Maps	Sets	Lists	Queues	Utilities
HashMap	HashSet	ArrayList	PriorityQueue	Collections
Hashtable	LinkedHashSet	Vector	ArrayDeque	Arrays
TreeMap	TreeSet	LinkedList		
LinkedHashMap				

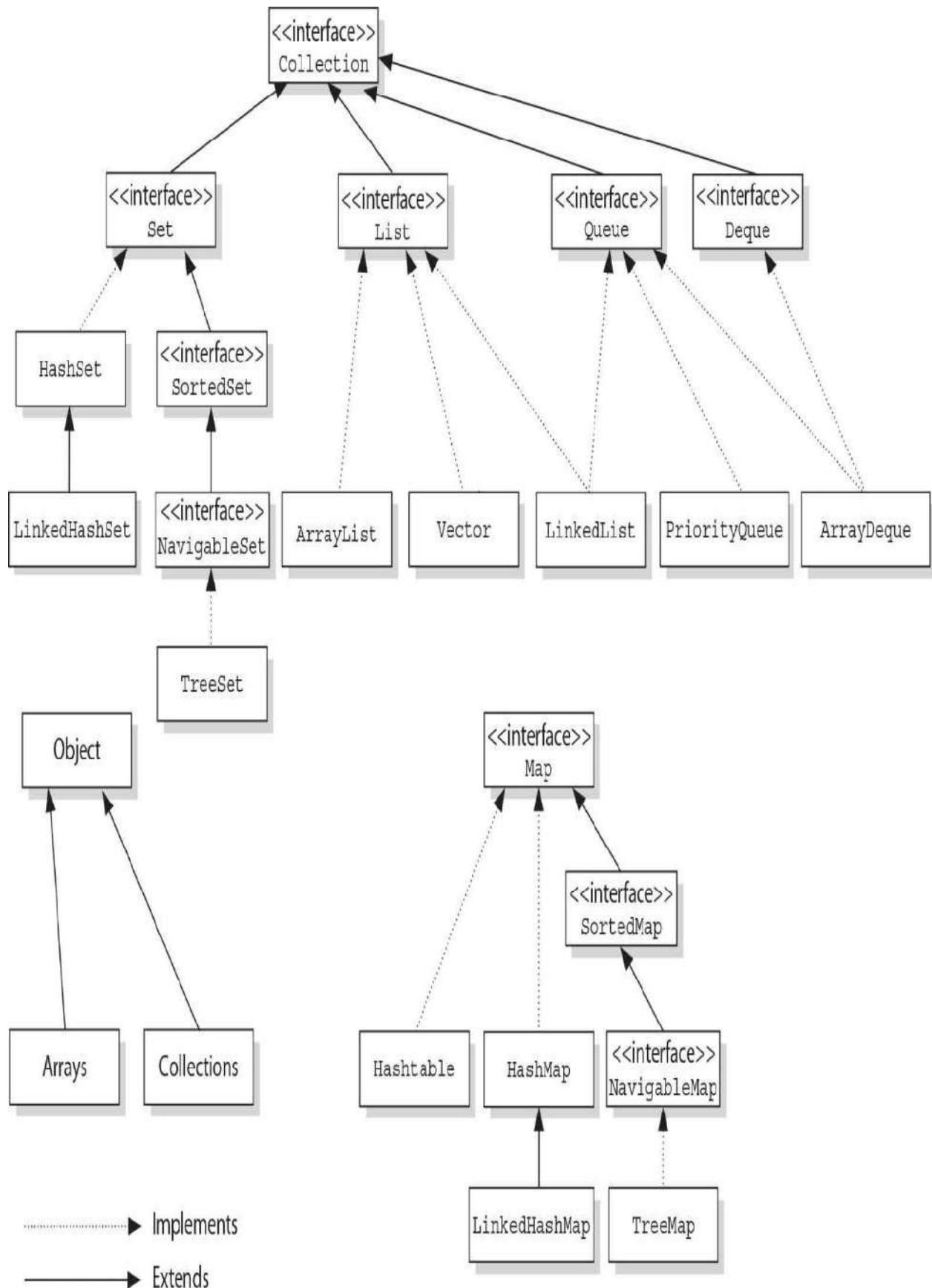
Note: In the table above, we listed more classes than are officially mentioned in the Oracle objectives. Oracle's objectives can be on the terse side! The classes we'll talk about fall into three categories:

- **Definitely on the exam:** ArrayList, ArrayDeque, TreeMap, TreeSet, and Arrays
- **Somewhat likely to be on the exam:** Collections, HashMap, Hashtable, and PriorityQueue
- **Unlikely to be on the exam:** HashSet, LinkedHashMap, LinkedHashSet, LinkedList, and Vector

We've included the "unlikelies" because understanding them will give you a better overview of how the key interfaces work. 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 6-2](#)). To make things a little more confusing, there are really three overloaded uses of the word "collection":

### FIGURE 6-2

The interface and class hierarchy for collections



- 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.



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

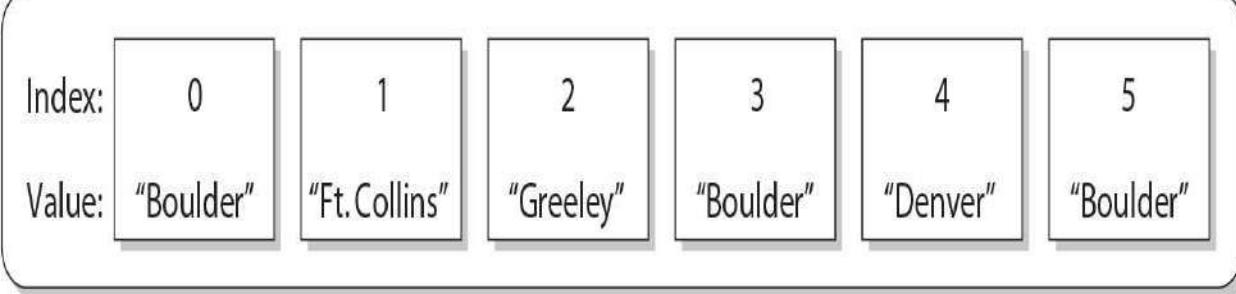
Collections come in four basic flavors:

- **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 in the order in which they are to be processed

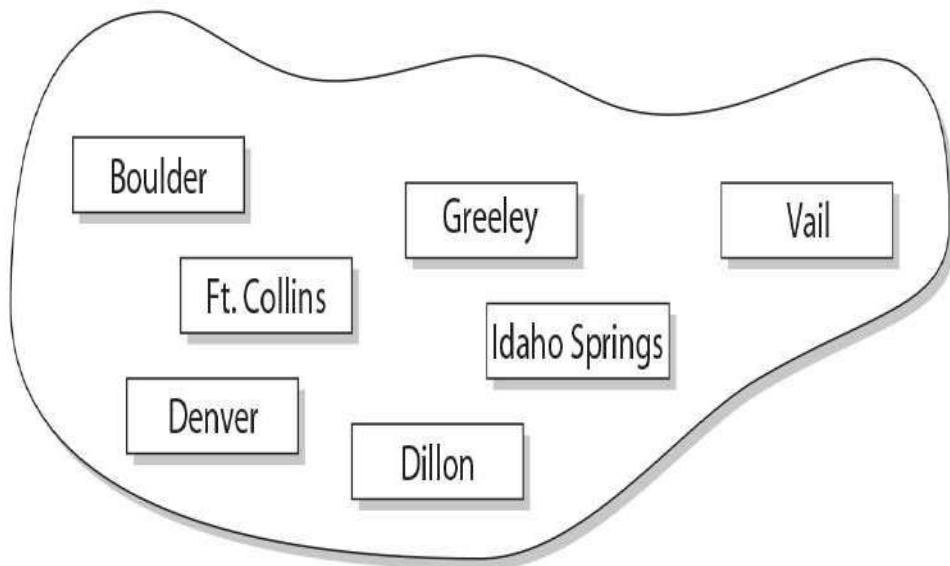
[Figure 6-3](#) illustrates the relative structures of a `List`, a `Set`, and a `Map`.

### FIGURE 6-3

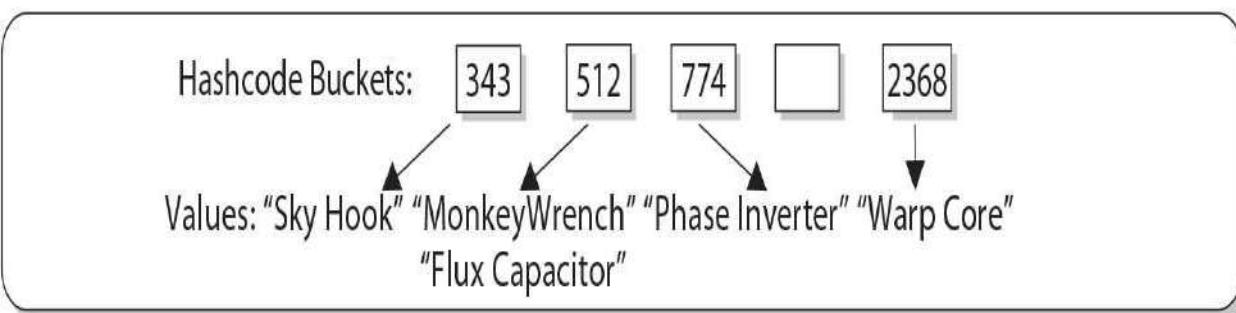
Examples of a `List`, a `Set`, and a `Map`



List: The salesman's itinerary (Duplicates allowed)



Set: The salesman's territory (No duplicates allowed)



HashMap: The salesman's products (Keys generated from product IDs)

But there are subflavors 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, whereas 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 it was last accessed or at what “position” it was added. 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 kind 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` come 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 nonlists don’t 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. `ArrayList` implements the `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 (Unlikely to Be on the Exam)** `vector` is a holdover from the earliest days of Java; `vector` and `Hashtable` were two of the original collections—the rest were added with later versions of Java. 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. Of the classes discussed here, only `vector` and `ArrayList` implement `RandomAccess`.

**LinkedList (Unlikely to Be on the Exam)** 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 (Unlikely to Be on the Exam)** 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 (Unlikely to Be on the Exam)** 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, whereas a `LinkedHashSet` lets you iterate through the elements in the order in which they were inserted.



*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 `Comparator`. As of Java 6, `TreeSet` implements `NavigableSet`.

## 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 (Somewhat Likely to Be on the Exam)** 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 (Somewhat Likely to Be on the Exam)** 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 although `HashMap` lets you have null values as well as one null key, a `Hashtable` doesn't let you have anything that's null.

**LinkedHashMap (Unlikely to Be on the Exam)** 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 `Comparator`) when you construct a `TreeMap` that specifies how the elements should be compared to one another when they're being ordered. As of Java 6, `TreeMap` implements `NavigableMap`.

## 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 have methods to add and subtract elements and review queue elements.

**PriorityQueue (Somewhat Likely to Be on the Exam)** 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.

**ArrayDeque** The `Deque` interface was added in Java 6. `Deque` (pronounced “deck”) is a double-ended queue, meaning you can add and remove items from both ends of the queue. `ArrayDeque` is one of the Collections that implements this interface, and it is a good choice for implementing either a queue or a stack because it is resizable with no capacity restrictions and it is designed to be high performance (but not thread safe).



*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 6-2 summarizes 12 of the 14 concrete collection-oriented classes you’ll need to understand for the exam. Even though not all of these classes are listed in the Section 3 objectives, you may encounter them on the exam so it’s a good idea to familiarize yourself with the classes in this table. (Arrays and Collections are coming right up!)

**TABLE 6-2** Collection Interface Concrete Implementation Classes

Class	Map	Set	List	Ordered	Sorted
HashMap	X			No	No
Hashtable	X			No	No
TreeMap	X			Sorted	By <i>natural order</i> or custom comparison rules
LinkedHashMap	X			By insertion order or last access order	No
HashSet		X		No	No
TreeSet		X		Sorted	By <i>natural order</i> 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
ArrayDeque				By position	No

*Be sure you know how to interpret [Table 6-2](#) 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 Collections (OCP Objectives 2.6, 3.2, and 3.3)

- 2.6 Create and use Lambda expressions.
- 3.2 Create and use `ArrayList`, `TreeSet`, `TreeMap`, and `ArrayDeque` objects.
- 3.3 Use `java.util.Comparator` and `java.lang.Comparable` interfaces.
- 3.X Sort and search arrays and lists.

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

Let's start with a quick review of what you learned about ArrayLists from the previous book, *OCA Java SE 8 Programmer 1 Exam Guide* (McGraw-Hill Education, 2017). The `java.util.ArrayList` class is one of the most commonly used 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();
```

Then as of Java 5 (yes, you might still encounter pre-Java 5 code out in the wild), you'd 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, starting with 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.)



**Why we're still talking about Java 5:**

- ***Understanding how collections worked before Java 5 makes generics easier to understand now.***
- ***On the exam, and in the real world, you'll have to understand how code written before Java 5 works and how such code interacts with more recent code.***

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:

```
List<String> test = new ArrayList<String>(); // declare the ArrayList  
String s = "hi";  
test.add("string"); // add some strings  
test.add(s);  
test.add(s+s);  
System.out.println(test.size()); // use ArrayList methods  
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 a lot 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 whether it contained specific objects; we removed an object right out from the middle of it; and then we rechecked its size.

## Autoboxing with Collections

In general, collections can hold objects but not primitives. Prior to Java 5, a

common use for the so-called wrapper classes (e.g., `Integer`, `Float`, `Boolean`, and so on) was to provide a way to get primitives into and out of collections. Prior to Java 5, you had to “wrap” a primitive manually before you could put it into a collection. Starting 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));    // Use Integer to "wrap" an int
```

In the previous example, we create an instance of class `Integer` with a value of 42. We’ve created an entire object to “wrap around” a primitive value. 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. There are some sneaky implications when we need to use wrapper objects; let’s take a closer look...

In the old, pre-Java 5 days, if you wanted to make a wrapper, unwrap it, use it, and then rewrap it, you might do something like this:

```
Integer y = new Integer(567);      // make it
int x = y.intValue();             // unwrap it
x++;
y = new Integer(x);              // rewrap it
System.out.println("y = " + y);   // print it
```

As of Java 5, you can say

```
Integer y = new Integer(567);      // make it
y++;                            // unwrap it, increment it,
                                // rewrap it
System.out.println("y = " + y);   // print it
```

Both examples produce the following output:

y = 568

And yes, you read that correctly. The code appears to be using the postincrement operator on an object reference variable! But it's simply a convenience. Behind the scenes, the compiler does the unboxing and reassignment for you. Earlier, we mentioned that wrapper objects are immutable...this example appears to contradict that statement. It sure looks like y's value changed from 567 to 568. What actually happened, however, is that a second wrapper object was created and its value was set to 568. If only we could access that first wrapper object, we could prove it...

Let's try this:

```

Integer y = 567;                      // make a wrapper
Integer x = y;                        // assign a second ref
                                         // var to THE wrapper

System.out.println(y==x);              // verify that they refer
                                         // to the same object
y++;                                 // unwrap, use, "rewrap"
System.out.println(x + " " + y);      // print values

System.out.println(y==x);              // verify that they refer
                                         // to different objects

```

which produces the output:

```

true
567 568
false

```

So, under the covers, when the compiler got to the line `y++;` it had to substitute something like this:

```

int x2 = y.intValue();                // unwrap it
x2++;
y = new Integer(x2);                // rewrap it

```

Just as we suspected, there's gotta be a call to `new` in there somewhere.

## **Boxing, ==, and equals()**

We just used `==` to do a little exploration of wrappers. Let's take a more thorough look at how wrappers work with `==`, `!=`, and `equals()`. The API developers decided that for all the wrapper classes, two objects are equal if they are of the same type and have the same value. It shouldn't be surprising that

```
Integer i1 = 1000;  
Integer i2 = 1000;  
if(i1 != i2) System.out.println("different objects");  
if(i1.equals(i2)) System.out.println("meaningfully equal");
```

produces the output

```
different objects  
meaningfully equal
```

It's just two wrapper objects that happen to have the same value. Because they have the same `int` value, the `equals()` method considers them to be "meaningfully equivalent" and, therefore, returns `true`. How about this one:

```
Integer i3 = 10;  
Integer i4 = 10;  
if(i3 == i4) System.out.println("same object");  
if(i3.equals(i4)) System.out.println("meaningfully equal");
```

This example produces the output:

```
same object  
meaningfully equal
```

Yikes! The `equals()` method seems to be working, but what happened with `==` and `!=`? Why is `!=` telling us that `i1` and `i2` are different objects, when `==` is saying that `i3` and `i4` are the same object? In order to save memory, two instances of the following wrapper objects (created through boxing) will always be `==` when their primitive values are the same:

- Boolean
- Byte
- Character from \u0000 to \u007f (7f is 127 in decimal)
- Short and Integer from –128 to 127

**When == is used to compare a primitive to a wrapper, the wrapper will be unwrapped and the comparison will be primitive to primitive.**

## Where Boxing Can Be Used

As we discussed earlier, it's common to use wrappers in conjunction with collections. Any time you want your collection to hold objects and primitives, you'll want to use wrappers to make those primitives collection-compatible. The general rule is that boxing and unboxing work wherever you can normally use a primitive or a wrapped object. The following code demonstrates some legal ways to use boxing:

```
class UseBoxing {  
    public static void main(String [] args) {  
        UseBoxing u = new UseBoxing();  
        u.go(5);  
    }  
  
    boolean go(Integer i) {      // boxes the int it was passed  
        Boolean ifSo = true;      // boxes the literal  
        Short s = 300;            // boxes the primitive  
        if(ifSo) {                // unboxing  
            System.out.println(++s); // unboxes, increments, reboxes  
        }  
        return !ifSo;             // unboxes, returns the inverse  
    }  
}
```



***Remember, wrapper reference variables can be null. That means you***

**have to watch out for code that appears to be doing safe primitive operations but that could throw a `NullPointerException`:**

```
class Boxing2 {  
    static Integer x;  
    public static void main(String [] args) {  
        doStuff(x);  
    }  
    static void doStuff(int z) {  
        int z2 = 5;  
        System.out.println(z2 + z);  
    } }
```

**This code compiles fine, but the JVM throws a `NullPointerException` when it attempts to invoke `doStuff(x)` because `x` doesn't refer to an `Integer` object, so there's no value to unbox.**

## The Java 7 “Diamond” Syntax

In the OCA book (*OCA Java SE 8 Programmer 1 Exam Guide* [McGraw-Hill Education, 2017]), we discussed several small additions/improvements to the language that were added under the name “Project Coin.” The last Project Coin improvement we’ll discuss in this book is the “diamond syntax.” We’ve already seen several examples of declaring type-safe collections, and as we go deeper into collections, we’ll see lots more like this:

```
ArrayList<String> stuff = new ArrayList<String>();  
List<Dog> myDogs = new ArrayList<Dog>();  
Map<String, Dog> dogMap = new HashMap<String, Dog>();
```

Notice that the type parameters are duplicated in these declarations. As of Java 7, these declarations could be simplified to

```
ArrayList<String> stuff = new ArrayList<>();  
List<Dog> myDogs = new ArrayList<>();  
Map<String, Dog> dogMap = new HashMap<>();
```

Notice that in the simpler Java 7 declarations, the right side of the declaration included the two characters “<>,” which together make a diamond shape—doh!

You cannot swap these; for example, the following declaration is NOT legal:

```
List<> stuff = new ArrayList<String>(); // NOT a legal diamond syntax
```

For the purposes of the exam, that’s all you’ll need to know about the diamond operator. For the remainder of the book, we’ll use the pre-diamond syntax and the Java 7 diamond syntax somewhat randomly—just like the real world!

## Sorting Collections and Arrays

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? There are a couple of ways to sort an `ArrayList`; for now we’ll use the `java.util.Collections` class to sort, and return later to the `ArrayList`’s `sort()` method.

```
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/??  
Caddyshack/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 (shown next) 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 ??  
, Caddyshack 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 (`dvdlist`).

If you've already figured out the problem, our guess is that you did it without

the help of the obscure error message shown earlier... 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 the `Comparable` interface. 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.util.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. Because 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` using 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.



***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 ??
, Caddyshack 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 from 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 both 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
, Caddyshack 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 6-3](#) to burn into your mind the differences between these two interfaces.

**TABLE 6-3** Comparing Comparable to Comparator

<b>java.lang.Comparable</b>	<b>java.util.Comparator</b>
<code>int objOne.compareTo(objTwo)</code>	<code>int compare(objOne, objTwo)</code>
Returns  <b>negative</b> if <code>objOne &lt; objTwo</code> <b>zero</b> if <code>objOne == objTwo</code> <b>positive</b> if <code>objOne &gt; objTwo</code>	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 <b>one</b> sort sequence can be created.	<b>Many</b> sort sequences can be created.
Implemented frequently in the API by:  String, wrapper classes, LocalDate, LocalTime...	Meant to be implemented to sort instances of third-party classes.

## Creating a Comparator with a Lambda Expression

To sort the `dvdlist` using a `Comparator`, we created a class, `GenreSort`, that implemented the `Comparator` interface. This interface has just one abstract method, `compare()`, as we said earlier, and because of this, `Comparator` is what we call a “functional interface.” You’ll learn a whole lot more about functional interfaces in [Chapter 8](#), but for now, just know that a functional interface is an interface with one and only one abstract method. Because it’s abstract, you must implement it when you implement the `Comparator` interface, like you do with the `GenreSort` `Comparator`.

You might remember from the section “Using Simple Lambdas” in [Chapter 6](#) of the *OCA Java SE 8 Programmer 1 Exam Guide* (McGraw-Hill Education, 2017), that we used the `java.util.function.Predicate` interface to create a lambda expression. A `Predicate` is also an example of a functional interface: an

interface with one abstract method. This is one of the things you need to know about lambda expressions; a lambda expression is an instance of a functional interface. That means whenever you're dealing with a class implementing a functional interface, like `Comparator`, you have an opportunity to use a lambda expression instead. Using lambda expressions certainly isn't required, but the nifty thing about using them is they can make your code more concise. Let's take another look at the `GenreSort` class:

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

As it must, the class implements the `compare()` method. To use it to sort our DVDs by genre, we passed an instance of this class to `Collections.sort()` as the second argument, the `comparator` argument:

```
GenreSort gs = new GenreSort();  
Collections.sort(dvdlist, gs);
```

Because `GenreSort` is implementing a functional interface, we can actually replace the entire class with a lambda expression. This lambda expression will look a bit different from the `Predicate` lambda we used in [Chapter 6](#) of *OCA Java SE 8 Programmer 1 Exam Guide*; this lambda needs to look like the `compare()` method of the `GenreSort` comparator. To turn that comparator into a lambda expression, here's what we do:

1. First we get the two parameters from the `compare()` method; those become the parameters for the lambda:  
  
(one, two)  
  
Because there are two parameters, we need to put them in parentheses.
4. Then, we write an arrow:  
  
(one, two) ->  
  
4. Then we write the body of the lambda expression. What should the body

be? Almost exactly what the body of the `compare()` method in `GenreSort` is:

```
(one, two) -> one.getGenre().compareTo(two.getGenre())
```

We don't need to use the curly braces and a `return` because there's only one statement in the body of `compare()`, so we can just copy the expression part after the `return`, and the lambda will automatically return the value for us.

Now we have a lambda expression for the `Comparator`, so how do we use it? All we have to do is replace the `GenreSort` instance in our code with the lambda:

```
Collections.sort(dvdlist,  
(one, two) -> one.getGenre().compareTo(two.getGenre()));
```

Now, we can rewrite the test program and completely eliminate the need for a separate `GenreSort` class:

```

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
        // Use a lambda expression as a Comparator
        Collections.sort(dvdlist,
            (one, two) ->
                one.getGenre().compareTo(two.getGenre())));
        System.out.println("--- sorted by genre, using Comparator lambda ---");
        System.out.println(dvdlist);
    }
}

```

If we want to delete the old code and delete the `GenreSort` class, we can because now we don't need that class at all to sort the `dvdlist`; we're using the lambda expression as the `Comparator` and writing the `Comparator` inline, where we need it when we call the `Collections.sort()` method.

You've seen the first three output lists; now here's a fourth:

```
--- sorted by genre, using Comparator lambda ---
[Patriot Games / action / Ford, Harrison
, Raiders of the Lost Ark / action / Ford, Harrison
, Caddyshack / comedy / Murray, Bill
, Lost in Translation / comedy / Murray, Bill
, 2001 / sci-fi / ???
, Donnie Darko / sci-fi / Gyllenhall, Jake
, Star Wars / sci-fi / Ford, Harrison
]
```

This is exactly the same output as we got using `GenreSort`, but now using a lambda expression instead. Just as you learned in [Chapter 6](#) of *OCA Java SE 8 Programmer 1 Exam Guide*, a lambda expression passes code as an argument. This time, the code is a `Comparator` method. You probably have a boatload of questions about lambda expressions at this point, but hang on just a bit longer. We'll cover all this in a lot more detail in [Chapter 8](#). The thing to know about lambdas at this stage is that they make convenient shortcuts for comparators. `Comparator` is a functional interface, so the entire class implementing the interface and the `compare()` method can be replaced by a lambda.

## Sorting ArrayLists Using the `sort()` Method

We've been using the `java.util.Collections` class to sort collections using the class method `sort()`, passing in the collection to sort, in this instance, the `dvdlist` `ArrayList`. `ArrayLists` can also be sorted directly by calling the `sort()` method on the list. And just like `Collections.sort()`, the `ArrayList` `sort()` method takes a `Comparator`. So we can pass in an instance of a class that implements `Comparator`, like we did for our `GenreSort` class:

```
GenreSort gs = new GenreSort();
dvdlist.sort(gs);           // sort using GenreSort comparator
```

Or we can pass in a `Comparator` made from a lambda expression:

```
// sort by genre, using a lambda comparator  
dvdlist.sort((one, two) -> one.getGenre().compareTo(two.getGenre()));
```

Either way we get the same results.

## Sorting with the Arrays Class

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 overloaded in the same way the `Collections.sort()` method is:

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

In addition, the `Arrays.sort()` method (the one-argument version) is overloaded about a million times to provide a couple of sort methods for every type of primitive. The `Arrays.sort(myArray)` 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.



*We've talked a lot about sorting by natural order and using Comparators to sort. The last rule you'll need to burn into your mind 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 positive 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. (Usually, 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 third 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:

- #1 Sort the sa array alphabetically (the natural order).
- #2 Search for the location of element "one", which is 1.
- #3 Make a Comparator instance. On the next line, we re-sort the array using the Comparator.
- #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.



***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.***

- #5 Search again, passing the Comparator to `binarySearch()`. This time, we get the correct answer, 2.
- #6 We define the Comparator; it's okay for this to be an inner class. (We'll be discussing inner classes in [Chapter 7](#).)
- #7 By switching the use of the arguments in the invocation of `compareTo()`, we get an inverted sort.

## Converting Arrays to Lists to Arrays

A couple of methods 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 is 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("\ns1[1] " + sList.get(1));
```

This produces

```
size 4
idx2 three
one five three six
s1[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 through 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 lets 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. Later in the chapter, [Table 6-7](#) 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();
        for(Object o : s)
            System.out.print(o + " ");
    }
}
```

If you insert the following line of code, you'll get output that looks 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` 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 crude 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 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 = 6. 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* hashCode 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 hashCode, 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 because 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.

## Navigating (Searching) TreeSets and TreeMaps

Note: This section and the next are fairly complex, and there is a good chance that you won’t get any questions on these topics. But again, Oracle’s objectives are somewhat terse, and we’d rather be on the safe side.

We’ve talked about searching lists and arrays. Let’s turn our attention to searching TreeSets and TreeMaps. Java 6 introduced (among other things) two interfaces: `java.util.NavigableSet` and `java.util.NavigableMap`. For the purposes of the exam, you’re interested in how TreeSet and TreeMap implement these interfaces.

Imagine that the Santa Cruz–Monterey ferry has an irregular schedule. Let’s say that we have the daily Santa Cruz departure times stored in military time in a TreeSet. Let’s look at some code that determines two things:

1. The last ferry that leaves before 4 PM (1600 hours)
2. The first ferry that leaves after 8 PM (2000 hours)

```
import java.util.*;
public class Ferry {
    public static void main(String[] args) {
        TreeSet<Integer> times = new TreeSet<Integer>();
        times.add(1205); // add some departure times
        times.add(1505);
        times.add(1545);
        times.add(1830);
        times.add(2010);
        times.add(2100);

        // Java 5 version

        TreeSet<Integer> subset = new TreeSet<Integer>();
        subset = (TreeSet)times.headSet(1600);
        System.out.println("J5 - last before 4pm is: " + subset.last());

        TreeSet<Integer> sub2 = new TreeSet<Integer>();
        sub2 = (TreeSet)times.tailSet(2000);
        System.out.println("J5 - first after 8pm is: " + sub2.first());

        // Java 6 version using the new lower() and higher() methods

        System.out.println("J6 - last before 4pm is: " + times.lower(1600));
        System.out.println("J6 - first after 8pm is: " + times.higher(2000));
    }
}
```

This should produce the following:

```
J5 - last before 4pm is: 1545  
J5 - first after 8pm is: 2010  
J6 - last before 4pm is: 1545  
J6 - first after 8pm is: 2010
```

As you can see in the preceding code, before the addition of the `NavigableSet` interface, zeroing in on an arbitrary spot in a `Set`—using the methods available in Java 5—was a compute-expensive and clunky proposition. On the other hand, using the Java 6 methods `lower()` and `higher()`, the code became a lot cleaner.

For the purpose of the exam, the `NavigableSet` methods related to this type of navigation are `lower()`, `floor()`, `higher()`, and `ceiling()`, and the mostly parallel `NavigableMap` methods are `lowerKey()`, `floorKey()`, `ceilingKey()`, and `higherKey()`. The difference between `lower()` and `floor()` is that `lower()` returns the element less than the given element, and `floor()` returns the element less than *or equal to* the given element. Similarly, `higher()` returns the element greater than the given element, and `ceiling()` returns the element greater than *or equal to* the given element. [Table 6-4](#) summarizes the methods you should know for the exam.

**TABLE 6-4** Important “Navigation”-Related Methods

Method	Description
<code>TreeSet.ceiling(e)</code>	Returns the lowest element $\geq e$
<code>TreeMap.ceilingKey(key)</code>	Returns the lowest key $\geq key$
<code>TreeSet.higher(e)</code>	Returns the lowest element $> e$
<code>TreeMap.higherKey(key)</code>	Returns the lowest key $> key$
<code>TreeSet.floor(e)</code>	Returns the highest element $\leq e$
<code>TreeMap.floorKey(key)</code>	Returns the highest key $\leq key$
<code>TreeSet.lower(e)</code>	Returns the highest element $< e$
<code>TreeMap.lowerKey(key)</code>	Returns the highest key $< key$
<code>TreeSet.pollFirst()</code>	Returns and removes the first entry
<code>TreeMap.pollFirstEntry()</code>	Returns and removes the first key/value pair
<code>TreeSet.pollLast()</code>	Returns and removes the last entry
<code>TreeMap.pollLastEntry()</code>	Returns and removes the last key/value pair
<code>TreeSet.descendingSet()</code>	Returns a NavigableSet in reverse order
<code>TreeMap.descendingMap()</code>	Returns a NavigableMap in reverse order

## Other Navigation Methods

In addition to the methods we just discussed, there were a few more methods

new to Java 6 that could be considered “navigation” methods. (Okay, it’s a little bit of a stretch to call these “navigation” methods, but just play along.)

## Polling

The idea of polling is that we want both to retrieve and remove an element from either the beginning or the end of a collection. In the case of `TreeSet`, `pollFirst()` returns and removes the first entry in the set, and `pollLast()` returns and removes the last. Similarly, `TreeMap` now provides `pollFirstEntry()` and `pollLastEntry()` to retrieve and remove key/value pairs.

## Descending Order

Also added in Java 6 for `TreeSet` and `TreeMap` were methods that returned a collection in the reverse order of the collection on which the method was invoked. The important methods for the exam are `TreeSet.descendingSet()` and `TreeMap.descendingMap()`.

[Table 6-4](#) summarizes the “navigation” methods you’ll need to know for the exam.

## Backed Collections

Some of the classes in the `java.util` package support the concept of “backed collections.” We’ll use a little code to help explain the idea:

```

TreeMap<String, String> map = new TreeMap<String, String>();
map.put("a", "ant"); map.put("d", "dog"); map.put("h", "horse");

SortedMap<String, String> submap;
submap = map.subMap("b", "g");           // #1 create a backed collection

System.out.println(map + " " + submap);   // #2 show contents

map.put("b", "bat");                     // #3 add to original
submap.put("f", "fish");                 // #4 add to copy

map.put("r", "raccoon");                // #5 add to original - out of range
// submap.put("p", "pig");               // #6 add to copy - out of range

System.out.println(map + " " + submap);   // #7 show final contents

```

This should produce something like this:

```

{a=ant, d=dog, h=horse} {d=dog}
{a=ant, b=bat, d=dog, f=fish, h=horse, r=raccoon} {b=bat, d=dog, f=fish}

```

The important method in this code is the `TreeMap.subMap()` method. It's easy to guess (and it's correct) that the `subMap()` method is making a copy of a portion of the `TreeMap` named `map`. The first line of output verifies the conclusions we've just drawn.

What happens next is powerful and a little bit unexpected (now we're getting to why they're called *backed* collections). When we add key/value pairs to either the original `TreeMap` or the partial-copy `SortedMap`, the new entries were automatically added to the other collection—sometimes. When `submap` was created, we provided a value range for the new collection. This range defines not only what should be included when the partial copy is created, but also defines the range of values that can be added to the copy. As we can verify by looking at

the second line of output, we can add new entries to either collection within the range of the copy, and the new entries will show up in both collections. In addition, we can add a new entry to the original collection, even if it's outside the range of the copy. In this case, the new entry will show up only in the original—it won't be added to the copy because it's outside the copy's range. Notice that we commented out line 6. If you attempt to add an out-of-range entry to the copied collection, an exception will be thrown.

For the exam, you'll need to understand the basics just explained, plus a few more details about three methods from `TreeSet`—`headSet()`, `subSet()`, and `tailSet()`—and three methods from `TreeMap`—`headMap()`, `subMap()`, and `tailMap()`. As with the navigation-oriented methods we just discussed, we can see a lot of parallels between the `TreeSet` and the `TreeMap` methods. The `headSet()`/`headMap()` methods create a subset that starts at the beginning of the original collection and ends at the point specified by the method's argument. The `tailSet()`/`tailMap()` methods create a subset that starts at the point specified by the method's argument and goes to the end of the original collection. Finally, the `subSet()`/`subMap()` methods allow you to specify both the start and end points for the subset collection you're creating.

As you might expect, the question of whether the subsetted collection's end points are inclusive or exclusive is a little tricky. The good news is that for the exam you have to remember only that when these methods are invoked with end point *and* boolean arguments, the boolean always means "is inclusive." A little more good news is that all you have to know for the exam is that, unless specifically indicated by a boolean argument, a subset's starting point will always be inclusive. Finally, you'll notice when you study the API that all of the methods we've been discussing here have an overloaded version that was added in Java 6. The older methods return either a `SortedSet` or a `SortedMap`; the Java 6 and later methods return either a `NavigableSet` or a `NavigableMap`. [Table 6-5](#) summarizes these methods.

**TABLE 6-5** Important "Backed Collection" Methods for `TreeSet` and `TreeMap`

Method	Description
headSet (e, b*)	Returns a subset ending at element e and <i>exclusive</i> of e
headMap (k, b*)	Returns a submap ending at key k and <i>exclusive</i> of key k
tailSet (e, b*)	Returns a subset starting at and <i>inclusive</i> of element e
tailMap (k, b*)	Returns a submap starting at and <i>inclusive</i> of key k
subSet (s, b*, e, b*)	Returns a subset starting at element s and ending just before element e
subMap (s, b*, e, b*)	Returns a submap starting at key s and ending just before key e

\* Note: These boolean arguments are optional. If they exist, it's a Java 6 method that lets you specify whether the start point and/or end point are exclusive, and these methods return a NavigableXXX. If the boolean argument(s) don't exist, the method returns either a SortedSet or a SortedMap.



*Let's say that you've created a backed collection using either a tailXXX() or subXXX() method. Typically, in these cases, the original and copy collections have different "first" elements. For the exam, it's important that you remember that the pollFirstXXX() methods will always remove the first entry from the collection on which they're invoked, but they will remove an element from the other collection only if it has the same value. So it's most likely that invoking pollFirstXXX() on the copy will remove an entry from both collections, but invoking pollFirstXXX() on the original will remove only the entry from the original.*

## Using the PriorityQueue Class and the Deque

# Interface

For the exam, you need to understand several of the classes that implement the Deque interface. These classes will be discussed in more detail in [Chapter 11, the concurrency chapter](#).

Other than those concurrency-related classes, the last two collection classes you need to understand for the exam are PriorityQueue and ArrayDeque.

## 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.

## ArrayDeque

`ArrayDeque` is a `Collection` class that implements the `Deque` interface. As we described earlier, `Deque` is an interface for double-ended queues, with methods for adding and removing elements to and from the queue at either end. Whereas the `Deque` interface allows for capacity-limited implementations, `ArrayDeque` has no capacity restrictions so adding elements will not fail.

The main advantage of `ArrayDeque` over, say, a `List` (like `ArrayList` or `LinkedList`) is performance. There are plenty of methods in the `Deque` interface to allow easy access to elements in the collection, and so `ArrayDeque` is a great choice if you're implementing a stack or queue. Unlike `PriorityQueue`, there is no natural ordering in `ArrayDeque`; it is simply a collection of elements that are stored in the order in which you add them. You'll also notice that many of the methods in `ArrayDeque` have similar names to `PriorityQueue`, primarily because both constructs are designed to implement queues. `ArrayDeque`

implements both the Queue interface and the Deque interface, so there are several methods with different names that do the same thing.

Here's an example to demonstrate how the various methods in the `ArrayDeque` interface add and access elements and remove elements from the collection:

```
List<Integer> nums = Arrays.asList(10, 9, 8, 7, 6, 5); // Create several
ArrayDeques, each with space for 2 items

ArrayDeque<Integer> a = new ArrayDeque<>(2);
ArrayDeque<Integer> b = new ArrayDeque<>(2);
ArrayDeque<Integer> c = new ArrayDeque<>(2);
ArrayDeque<Integer> d = new ArrayDeque<>(2);
ArrayDeque<Integer> e = new ArrayDeque<>(2);

// add 6 items to each Deque, each using different methods
for (Integer n : nums) {
    a.offer(n);          // add on the end
    b.offerFirst(n);    // add on the front
    c.push(n);           // add on the front
    d.add(n);            // add on the end
    e.addFirst(n);       // add on the front
}

// display the deques
System.out.println("a: " + a);
System.out.println("b: " + b);
System.out.println("c: " + c);
System.out.println("d: " + d);
System.out.println("e: " + e);
```

In this example, we first create several empty `ArrayDeques` (a through e), each using the `ArrayDeque` constructor specifying the number of spaces to

allocate, in this case, two. However, remember that `ArrayDeque` is not size constrained, so in the `for` loop, where we iterate through the `List` of numbers and add six numbers to each of the `ArrayDeques`, no exceptions are thrown and no errors are created.

As you can see in the output:

```
a: [10, 9, 8, 7, 6, 5]
b: [5, 6, 7, 8, 9, 10]
c: [5, 6, 7, 8, 9, 10]
d: [10, 9, 8, 7, 6, 5]
e: [5, 6, 7, 8, 9, 10]
```

the methods we're using to add elements to the deques result in different orderings of the deques depending on whether the method adds an element to the front of the deque or the end of the deque. The methods `offerFirst()`, `push()`, and `addFirst()` add elements to the front of the deque, so we see the numbers in reverse order of the `List`. The methods `offer()` and `add()` add elements to the end of the deque, so we see the numbers in the same order as they appeared in the `List`.

Next, let's do some experiments accessing the elements in deque `e`. First, we `peek()`, which returns the first element (sometimes called the *head* of the queue) without removing it from the deque:

```
System.out.println("First element of e: " + e.peek());
System.out.println("e hasn't changed: " + e);
```

This code produces the output:

```
First element of e: 5
e hasn't changed: [5, 6, 7, 8, 9, 10]
```

Then we `poll()`, which removes the first element from the deque and returns it:

```
System.out.println("First element of e: " + e.poll());
System.out.println("e has been modified: " + e);
```

We see the output:

```
First element of e: 5  
e has been modified: [6, 7, 8, 9, 10]
```

Then we `pop()`, which also removes the first element from the deque and returns it:

```
System.out.println("First element of e: " + e.pop());  
System.out.println("e has been modified: " + e);
```

So we get the output:

```
First element of e: 6  
e has been modified: [7, 8, 9, 10]
```

Then we `pollLast()`, which removes the last element from the deque (sometimes called the *tail* of the queue) and returns it:

```
System.out.println("Last element of e: " + e.pollLast());  
System.out.println("e has been modified: " + e);
```

This code produces:

```
Last element of e: 10  
e has been modified: [7, 8, 9]
```

Then we call `removeLast()` three times to remove the three remaining elements from the end of the deque, returning each one, so we see the elements in reverse order from the end of the deque:

```
System.out.println("Remove all remaining elements of e: " +  
    e.removeLast() + " " + e.removeLast() + " " + e.removeLast());  
System.out.println("e has been modified: " + e);
```

The output is

```
Remove all remaining elements of e: 9 8 7  
e has been modified: []
```

Now, e is empty.

Next, let's test to see what happens if we use various methods to try to remove another item from the deque. If we `pop()` or `remove()`, we get a `java.util.NoSuchElementException` error:

```
// calling pop() throws a java.util.NoSuchElementException  
System.out.println("Try to pop one more item: " + e.pop());  
  
// calling remove() throws a java.util.NoSuchElementException  
System.out.println("Try to remove one more item: " + e.remove());
```

But if we `poll()`, we get `null`:

```
// There's nothing left, so calling poll() returns null  
System.out.println("Try to poll one more item: " + e.poll());
```

So we see the output:

```
Try to poll one more item: null
```

Remember that, unlike many of the `Deque` classes we'll look at in [Chapter 11](#), `ArrayDeque` is not thread safe, so you have to synchronize on classes that access a shared `ArrayDeque` in a multithreaded environment.

## Method Overview for Arrays and Collections

For these two classes, we've already covered the trickier methods you might encounter on the exam. [Table 6-6](#) 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 6-6** | Key Methods in Arrays and Collections

Key Methods in <code>java.util.Arrays</code>	Descriptions
<code>static List asList(T[])</code>	Convert an array to a List (and bind them).
<code>static int binarySearch(Object[], key)</code> <code>static int binarySearch(primitive[], key)</code>	Search a sorted array for a given value; return an index or insertion point.
<code>static int binarySearch(T[], key, Comparator)</code>	Search a Comparator-sorted array for a value.
<code>static boolean equals(Object[], Object[])</code> <code>static boolean equals(primitive[], primitive[])</code>	Compare two arrays to determine if their contents are equal.
<code>static void sort(Object[])</code> <code>static void sort(primitive[])</code>	Sort the elements of an array by natural order.
<code>static void sort(T[], Comparator)</code>	Sort the elements of an array using a Comparator.
<code>static String toString(Object[])</code> <code>static String toString(primitive[])</code>	Create a String containing the contents of an array.
Key Methods in <code>java.util.Collections</code>	Descriptions
<code>static int binarySearch(List, key)</code> <code>static int binarySearch(List, key, Comparator)</code>	Search a "sorted" List for a given value; return an index or insertion point.
<code>static void reverse(List)</code>	Reverse the order of elements in a List.
<code>static Comparator reverseOrder()</code> <code>static Comparator reverseOrder(Comparator)</code>	Return a Comparator that sorts the reverse of the collection's current sort sequence.
<code>static void sort(List)</code> <code>static void sort(List, Comparator)</code>	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 6-7](#) lists a summary of the List, Set, and Map methods you should be aware of.

**TABLE 6-7** Key Methods in List, Set, and Map

Key Interface Methods	List	Set	Map	Descriptions
boolean <b>add(element)</b> boolean <b>add(index, element)</b>	X X	X		Add an element. For Lists, optionally add the element at an index point.
boolean <b>contains(object)</b> boolean <b>containsKey(object key)</b> boolean <b>containsValue(object value)</b>	X	X	X X	Search a collection for an object (or, optionally for Maps, a key); return the result as a boolean.
object <b>get(index)</b> object <b>get(key)</b>	X		X	Get an object from a collection via an index or a key.
int <b>indexOf(object)</b>	X			Get the location of an object in a List.
Iterator <b>iterator()</b>	X	X		Get an Iterator for a List or a Set.
Set <b>keySet()</b>			X	Return a Set containing a Map's keys.
put(key, value)			X	Add a key/value pair to a Map.
element <b>remove(index)</b> element <b>remove(object)</b> element <b>remove(key)</b>	X X X	X		Remove an element via an index, or via the element's value, or via a key.
int <b>size()</b>	X	X	X	Return the number of elements in a collection.
Object[] <b>toArray()</b> T[] <b>toArray(T[])</b>	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).

The corresponding `ArrayDeque` methods do the same things. We also talked about the `ArrayDeque` methods `pop()`, `pollLast()`, `remove()`, and `removeLast()`. `pop()` and `remove()` remove the first element from the deque, and `pollLast()` and `removeLast()` remove the last element from the deque. Just remember that elements in the `ArrayDeque` are simply ordered by their position in the deque.



*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 (OCP Objective 3.1)

#### 3.1 Create and use a generic class.

Now would be a great time to take a break. This innocent-sounding objective unpacks into a world of complexity. When you’re well rested, come on back and strap yourself in—the next several pages might get bumpy.

*Arrays in Java have always been type-safe—an array declared as type String (String []) can’t accept Integers (or ints), 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, although 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 the Java engineers 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. The 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 nontype-safe (nongeneric/pre-Java 5) collections to

still work together. Their solution isn't the friendliest, but it does let you use older nongeneric code, as well as use generic code that plays with nongeneric code. But notice we said "plays" and not "plays WELL."

While you can integrate Java 5 and later generic code with legacy, nongeneric 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 nongeneric) collections.

The Java 7 exam covered both pre-Java 5 (nongeneric) and generic-style collections. You may still see questions on the Java 8 exam that expect you to understand the tricky problems that can come from mixing nongeneric and generic code together, although it is less likely. 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.

## 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 nongeneric collection can hold any kind of object! A nongeneric 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` 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 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 the “parameterized type,” “type parameter,” or, of course, just old-fashioned “type.” In this chapter, we’ll refer to it both 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 previous method 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 List<Dog> getDogList() {  
    List<Dog> dogs = new ArrayList<Dog>();  
    // more code to insert dogs  
    return dogs;  
}
```

The compiler will stop you from returning anything not compatible with a `List<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 List is returned, those calling the method won't need a cast to take Dogs

from the `List`:

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

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

```
public List getDogList() {  
    List dogs = new ArrayList();  
    // code to add only Dogs... fingers crossed...  
    return dogs; // a List 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 `List`'s `get()` method; we aren't casting what is returned from the `getDogList()` method, which is a `List`.)

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 nongeneric collection—you can put ANY object type into the collection. You'll see a little later that nongeneric 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

nongeneric code, so we're just getting warmed up here.

## Generics and Legacy Code

The easiest thing about generics you'll need to know for the exam is how to update nongeneric 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 (or you use the Java 7 diamond syntax), 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>(); // (or the J7 diamond!)
```

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 nongeneric 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 Nongeneric 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 or later 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 nongenerics 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 nongeneric 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 and later allowed your newer type-safe code to make use of older code (it would have been a nightmare to ask several million Java developers to modify all their existing code).

So, the Java 5 or later compiler (from now on “the Java 5 compiler”) was *forced* into letting you compile your new type-safe code even though your code invokes a method of an older class that takes a nontype-safe argument and does who knows what with it.

However, just because **the Java 5 compiler** (remember this means Java 5 and later), 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>`.“



***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 versus 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 the 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.



***For the purposes of the exam, unless the question includes an answer that mentions warnings, even if you know the 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 nontype-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 Java 5 and later code you write using generics—looks exactly like the pregeneric 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 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 nongeneric 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 nontype-safe), so you took appropriate defensive steps in your code. The problem with mixing generic and nongeneric 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 nontype-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.



***When using legacy (nontype-safe) collections, watch out for unboxing problems! If you declare a nongeneric 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 refers to an `Integer` (or some other wrapped primitive) on the heap. Unboxing converts only from a wrapper class reference (like an `Integer` or a `Long`) to a primitive.***

***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 nongeneric 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> bList = new ArrayList<JButton>(); // yes
List<Object> oList = new ArrayList<Object>();   // yes
List<Integer> iList = 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 a 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. 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 Java 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[]
}
```

If this is true and 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 previous code 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 versus 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 that 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 that 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 that 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, nontype-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> 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 shown previously 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,” which you aren’t required to know any details of.)

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 your brain!

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 okay.”

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 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.

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 you can add to the list because 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:

```
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<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 is 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 (3)** `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 (4)** `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 (6)** `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, the Strategy design pattern isn'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:

```
    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, U> {
    T one;
    U two;
    UseTwo(T one, U two) {
        this.one = one;
        this.two = two;
    }
    T getT() { return one; }
    X getU() { 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 theU = twos.getU();   // 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.*



*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
```

In practice, **98 percent** 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 work.

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 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 `Object.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 hashCodeing. 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. Along the way, we introduced the “diamond” syntax, and we talked about autoboxing primitives into and out of wrapper class objects.

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 very limited 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() (OCP Objectives 1.4, 3.3, and 3.X)

- 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, and 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 (OCP Objective 3.2)

- ❑ 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, and 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 subflavors of collections: Sorted, Unsorted, Ordered, and Unordered:
  - ❑ **Ordered** Iterating through a collection in a specific, nonrandom order
  - ❑ **Sorted** Iterating through a collection in a sorted order
- ❑ Sorting can be alphabetic, numeric, or programmer-defined.

## Key Attributes of Common Collection Classes (OCP Objective 3.2)

- 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/values); 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.
- ArrayDeque** Like an **ArrayList** only better performance; ordered only by index. Good for stacks and queues.

## Using Collection Classes (OCP Objectives 3.2 and 3.3)

- Collections hold only objects, but primitives can be autoboxed.
- Java 7 and later allows “diamond” syntax: `List<Dog> d = new ArrayList<>();`.
- Iterate with the enhanced `for` or with an `Iterator` via `hasNext()` and `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 and Deques use `offer()` to add an element, `poll()` to remove the head of the queue, and `peek()` to look at the head of a queue.

- ❑ TreeSets and TreeMap have navigation methods like `floor()` and `higher()`.
- ❑ You can create/extend “backed” subcopies of TreeSets and TreeMap.

## Sorting and Searching Arrays and Lists (OCP Objectives 2.6 and 3.3)

- ❑ 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()`.
- ❑ Use a lambda expression as a shorthand to create a Comparator.
- ❑ To be sorted and searched, an array’s or List’s elements must be *comparable*.
- ❑ To be searched, an array or List must first be sorted.

## Utility Classes: Collections and Arrays (OCP Objectives 3.2 and 3.3)

- ❑ These `java.util` classes provide
  - ❑ A `sort()` method. Sort using a Comparator or sort using natural order.
  - ❑ A `binarySearch()` method. Search a presorted 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 (OCP Objective 3.1)

- ❑ 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 nongeneric 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 nongeneric 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 nontype-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); }
```

you'll get a warning because `add()` is potentially "unsafe."

- ❑ "Compiles without error" is not the same as "compiles without warnings." 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 apply 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 to 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>`, whereas `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) { }
```

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

## Q SELF TEST

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

- 1.** 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>();
2. 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

3. 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 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

4. 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.equals(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

5. 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.     }  
22.     public void setBalance(String accountName, int amount) {  
23.         accountTotals.put(accountName, Integer.valueOf(amount));  
24.     }  
25. }  
26. }
```

This class is going 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());
```

- 6.** 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
7. 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
  - G. java.util.ArrayDeque
8. Given a method declared as

```
public static <E extends Number> List<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;`

9. Given the proper import statement(s) and

- 13. `PriorityQueue<String> pq = new PriorityQueue<String>();`
- 14. `pq.add("2");`
- 15. `pq.add("4");`
- 16. `System.out.print(pq.peek() + " ");`
- 17. `pq.offer("1");`

```
18. pq.add("3");
19. pq.remove("1");
20. System.out.print(pq.poll() + " ");
21. if(pq.remove("2")) System.out.print(pq.poll() + " ");
22. System.out.println(pq.poll() + " " + pq.peek());
```

What is the result?

- A. 2 2 3 3
  - B. 2 2 3 4
  - C. 4 3 3 4
  - D. 2 2 3 3 3
  - E. 4 3 3 3 3
  - F. 2 2 3 3 4
  - G. Compilation fails
  - H. An exception is thrown at runtime
- 10.** Given the proper import statement(s) and

```
ArrayDeque<String> ad = new ArrayDeque<>();
ad.add("2");
ad.add("4");
System.out.print(ad.peek() + " ");
ad.offer("1");
ad.add("3");
ad.remove();
System.out.print(ad.poll() + " ");
if (ad.peek().equals("2")) System.out.print(ad.poll() + " ");
System.out.println(ad.poll() + " " + ad.peek());
```

What is the result?

- A. 2 2 3 3
- B. 2 2 3 4
- C. 4 3 3 4
- D. 2 2 4 3
- E. 2 4 1 3
- F. 2 2 3 3 4
- G. Compilation fails
- H. An exception is thrown at runtime

11. Given:

```
3. import java.util.*;  
4. public class Mixup {  
5.     public static void main(String[] args) {  
6.         Object o = new Object();  
7.         // insert code here  
8.         s.add("o");  
9.         s.add(o);  
10.    }  
11. }
```

And these three fragments:

```
I. Set s = new HashSet();  
II. TreeSet s = new TreeSet();  
III. LinkedHashSet s = new LinkedHashSet();
```

When fragments I, II, or III are inserted independently at line 7, which are true? (Choose all that apply.)

- A. Fragment I compiles
- B. Fragment II compiles
- C. Fragment III compiles

- D. Fragment I executes without exception
- E. Fragment II executes without exception
- F. Fragment III executes without exception

**12.** Given:

```
3. import java.util.*;  
4. class Turtle {  
5.     int size;  
6.     public Turtle(int s) { size = s; }  
7.     public boolean equals(Object o) { return (this.size == ((Turtle)o).size); }  
8.     // insert code here  
9. }  
10. public class TurtleTest {  
11.     public static void main(String[] args) {  
12.         LinkedHashSet<Turtle> t = new LinkedHashSet<Turtle>();  
13.         t.add(new Turtle(1));  t.add(new Turtle(2));  t.add(new Turtle(1));  
14.         System.out.println(t.size());  
15.     }  
16. }
```

And these two fragments:

```
I.    public int hashCode() { return size/5; }  
II.   // no hashCode method declared
```

If fragment I or II is inserted independently at line 8, which are true?  
(Choose all that apply.)

- A. If fragment I is inserted, the output is 2
- B. If fragment I is inserted, the output is 3
- C. If fragment II is inserted, the output is 2

- D. If fragment II is inserted, the output is 3
- E. If fragment I is inserted, compilation fails
- F. If fragment II is inserted, compilation fails

**13.** (OCJPJ 6 only) Given the proper import statement(s) and

```
13. TreeSet<String> s = new TreeSet<String>();  
14. TreeSet<String> subs = new TreeSet<String>();  
15. s.add("a"); s.add("b"); s.add("c"); s.add("d"); s.add("e");  
16.  
17. subs = (TreeSet)s.subSet("b", true, "d", true);  
18. s.add("g");  
19. s.pollFirst();  
20. s.pollFirst();  
21. s.add("c2");  
22. System.out.println(s.size() +" "+ subs.size());
```

Which are true? (Choose all that apply.)

- A. The size of s is 4
- B. The size of s is 5
- C. The size of s is 7
- D. The size of subs is 1
- E. The size of subs is 2
- F. The size of subs is 3
- G. The size of subs is 4
- H. An exception is thrown at runtime

**14.** (Note: Some of the classes used in this question are very unlikely to be on the real exam. Feel free to skip this question.)

Given:

```

3. import java.util.*;
4. public class Magellan {
5.     public static void main(String[] args) {
6.         TreeMap<String, String> myMap = new TreeMap<String, String>();
7.         myMap.put("a", "apple"); myMap.put("d", "date");
8.         myMap.put("f", "fig"); myMap.put("p", "pear");
9.         System.out.println("1st after mango: " + // sop 1
10.            myMap.higherKey("f"));
11.        System.out.println("1st after mango: " + // sop 2
12.            myMap.ceilingKey("f"));
13.        System.out.println("1st after mango: " + // sop 3
14.            myMap.floorKey("f"));
15.        SortedMap<String, String> sub = new TreeMap<String, String>();
16.        sub = myMap.tailMap("f");
17.        System.out.println("1st after mango: " + // sop 4
18.            sub.firstKey());
19.    }
20. }

```

Which of the `System.out.println` statements will produce the output `1st after mango: p?` (Choose all that apply.)

- A. sop 1
- B. sop 2
- C. sop 3
- D. sop 4
- E. None; compilation fails
- F. None; an exception is thrown at runtime

**15.** Given:

```
3. import java.util.*;
4. class Business { }
5. class Hotel extends Business { }
6. class Inn extends Hotel { }
7. public class Travel {
8.     ArrayList<Hotel> go() {
9.         // insert code here
10.    }
11. }
```

Which statement inserted independently at line 9 will compile? (Choose all that apply.)

- A. return new ArrayList<Inn>();
- B. return new ArrayList<Hotel>();
- C. return new ArrayList<Object>();
- D. return new ArrayList<Business>();

**16.** Given:

```
3. import java.util.*;
4. class Dog { int size; Dog(int s) { size = s; } }
5. public class FirstGrade {
6.     public static void main(String[] args) {
7.         TreeSet<Integer> i = new TreeSet<Integer>();
8.         TreeSet<Dog> d = new TreeSet<Dog>();
```

```
9.  
10.    d.add(new Dog(1));  d.add(new Dog(2));  d.add(new Dog(1));  
11.    i.add(1);  i.add(2);  i.add(1);  
12.    System.out.println(d.size() + " " + i.size());  
13. }  
14. }
```

What is the result?

- A. 1 2
- B. 2 2
- C. 2 3
- D. 3 2
- E. 3 3
- F. Compilation fails
- G. An exception is thrown at runtime

**17.** Given:

```
3. import java.util.*;
4. public class GeoCache {
5.     public static void main(String[] args) {
6.         String[] s = {"map", "pen", "marble", "key"};
7.         Othello o = new Othello();
8.         Arrays.sort(s,o);
9.         for(String s2: s) System.out.print(s2 + " ");
10.        System.out.println(Arrays.binarySearch(s, "map"));
11.    }
12.    static class Othello implements Comparator<String> {
13.        public int compare(String a, String b) { return b.compareTo(a); }
14.    }
15. }
```

Which are true? (Choose all that apply.)

- A. Compilation fails
- B. The output will contain a 1
- C. The output will contain a 2
- D. The output will contain a -1
- E. An exception is thrown at runtime
- F. The output will contain "key map marble pen"
- G. The output will contain "pen marble map key"

**18.** Given:

```

class DogSort implements Comparator<Dog> {
    public int compare(Dog one, Dog two) {
        return one.getName().compareTo(two.getName());
    }
}
Dog boi = new Dog("boi", 30, 6);
Dog clover = new Dog("clover", 35, 12);
Dog zooey = new Dog("zooey", 45, 8);
ArrayList<Dog> dogs = new ArrayList<>(Arrays.asList(zooey, clover, boi));

```

Which of the following code fragments sorts the dogs in ascending order by name? (Choose all that apply.)

- A. DogSort dogSorter = new DogSort();
 dogs.sort(dogSorter);
- B. dogs.sort((d1, d2) -> d1.getName().compareTo(d2.getName()));
- C. dogs.sort(DogSort);
- D. dogs.sort(int compare(d1, d2) ->
 d1.getName().compareTo(d2.getName()));
- E. dogs.sort((d1, d2) -> d2.getName().compareTo(d1.getName()));

## A SELF TEST ANSWERS

1.  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. (OCP Objective 3.2)

2.  **B** and **D** are correct. **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 hashCode value. **A** and **E** are a negation of the `hashCode()` and `equals()` contract. (OCP Objectives 1.4 and 3.3)
3.  **E** is correct. You can't put both `Strings` and `ints` 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. (OCP Objective 3.1)
4.  **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. (OCP Objectives 1.4, 3.3)
5.  **B**, **E**, and **G** are correct.  
☒ **A** is incorrect because you can't use a primitive type as a type parameter. **C** is incorrect because a `Map` takes two type parameters separated by a comma. **D** is incorrect because an `int` can't autobox to a `null`, and **F** is incorrect because a `null` can't unbox to `0`. **H** is incorrect because you can't autobox a primitive just by trying to invoke a method with it. (OCP Objectives 3.2)
6.  **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`. (OCP Objective 3.1)

7.  **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 earlier. **C**, `List`, is an interface, and **F** and **G**, `PriorityQueue` and `ArrayDeque`, do not offer access by index. (OCP Objective 3.2)
8.  **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 incorrect. **C** is incorrect 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. (OCP Objective 3.1)
9.  **B** is correct. For the sake of the exam, `add()` and `offer()` both add to (in this case) naturally sorted queues. The calls to `poll()` both return and then remove the first item from the queue, so the test fails.

☒ **A**, **C**, **D**, **E**, **F**, **G**, and **H** are incorrect based on the above. (OCP Objective 3.2)
10.  **E** is correct. `add()` and `offer()` both add to the end of the deque. The calls to `poll()` and `remove()` both return and then remove the first item from the queue. The calls to `peek()` return the first item in the queue

without removing it. The test fails because the first item is not "2"" (it's been removed).

**A, B, C, D, F, G, and H** are incorrect based on the above. (OCP Objective 3.2)

**11.**  **A, B, C, D, and F** are all correct.

Only **E** is incorrect. Elements of a `TreeSet` must in some way implement `Comparable`. (OCP Objective 3.3)

**12.**  **A and D** are correct. While fragment II wouldn't fulfill the `hashCode()` contract (as you can see by the results), it is legal Java. For the purpose of the exam, if you don't override `hashCode()`, every object will have a unique hashcode.

**B, C, E, and F** are incorrect based on the above. (OCP Objectives 1.4, 3.3)

**13.**  **B and F** are correct. After "g" is added, `TreeSet s` contains six elements and `TreeSet subs` contains three (b, c, d), because "g" is out of the range of `subs`. The first `pollFirst()` finds and removes only the "a". The second `pollFirst()` finds and removes the "b" from *both* `TreeSets` (remember they are backed). The final `add()` is in range of both `TreeSets`. The final contents are [c, c2, d, e, g] and [c, c2, d].

**A, C, D, E, G, and H** are incorrect based on the above. (OCP Objective 3.2)

**14.**  **A** is correct. The `ceilingKey()` method's argument is inclusive. The `floorKey()` method would be used to find keys before the specified key. The `firstKey()` method's argument is also inclusive.

**B, C, D, E, and F** are incorrect based on the above. (OCP Objective 3.2)

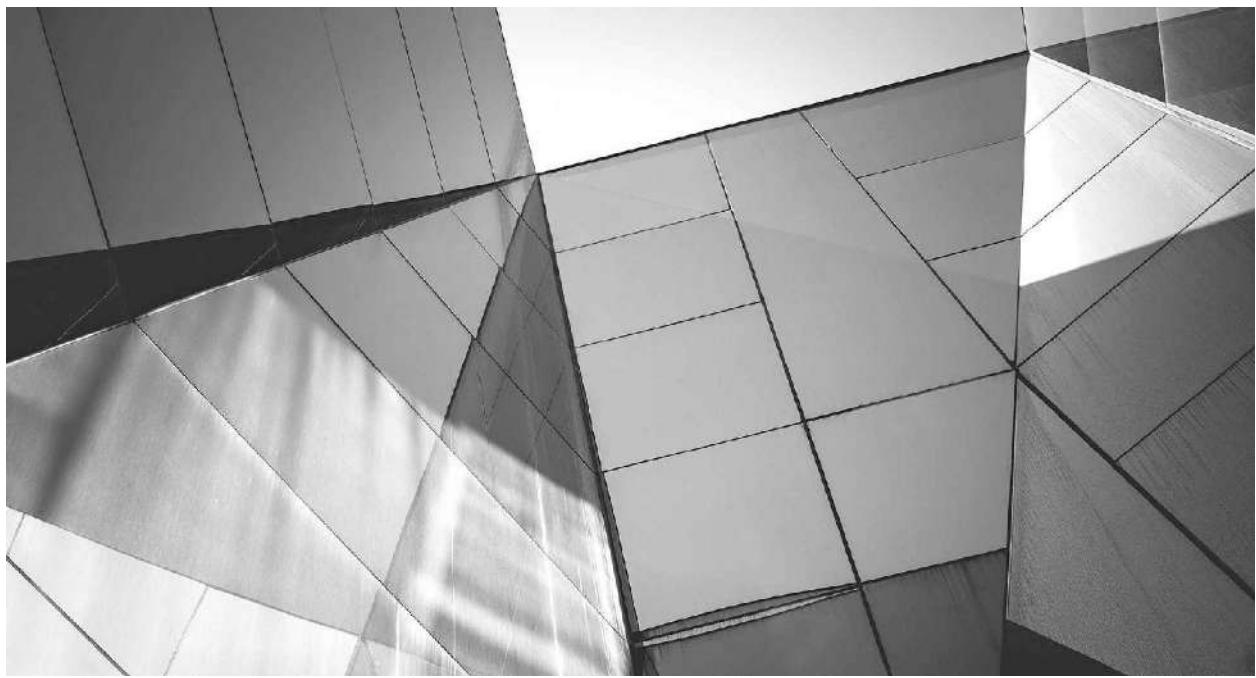
**15.**  **B** is correct.

**A** is incorrect because polymorphic assignments don't apply to generic type parameters. **C and D** are incorrect because they don't follow basic polymorphism rules. (OCP Objective 3.1)

**16.**  **G** is correct. Class `Dog` needs to implement `Comparable` in order for a `TreeSet` (which keeps its elements sorted) to be able to contain `Dog` objects.

**A, B, C, D, E, and F** are incorrect based on the above. (OCP Objectives 3.2 and 3.3)

- 17.**  **D** and **G** are correct. First, the `compareTo()` method will reverse the normal sort. Second, the `sort()` is valid. Third, the `binarySearch()` gives `-1` because it needs to be invoked using the same `Comparator (o)` as was used to sort the array. Note that when the `binarySearch()` returns an “undefined result,” it doesn’t officially have to be a `-1`, but it usually is, so if you selected only **G**, you get full credit!
- A, B, C, E, and F** are incorrect based on the above. (OCP Objective 3.3)
- 18.**  **A** and **B** are correct. **A** uses the `dogSorter` comparator made from the `DogSort Comparator` class, and **B** uses the lambda expression equivalent of `dogSorter`.
- C, D, and E** are incorrect. The argument to **C** is the class name for the `Comparator` rather than an instance. **D** uses the incorrect syntax for a lambda expression. **E** is almost right, but it sorts the dogs in the wrong order. (OCP Objective 2.6).



# 7

## Inner Classes

### CERTIFICATION OBJECTIVES

- Create Top-Level and Nested Classes
  - Create Inner Classes Including Static Inner Class, Local Class, Nested Class, and Anonymous Inner Class
  - Create and Use Lambda Expressions
- ✓ Two-Minute Drill

#### Q&A Self Test

Inner classes (including static nested classes) appear throughout the exam. The code used to represent questions on virtually *any* topic on the exam can involve inner (aka nested) classes. Unless you deeply understand the rules and syntax for inner classes, you're likely to miss questions you'd otherwise be able to answer. *As if the exam weren't already tough enough.*

This chapter looks at the ins and outs (inners andouters?) of inner classes and exposes you to the kinds of (often strange-looking) syntax examples you'll see scattered throughout the entire exam. So you've really got two goals for this chapter—to learn what you'll need to answer questions testing your inner-class knowledge and to learn how to read and understand inner-class code so you can handle questions testing your knowledge of *other* topics.

What's all the hoopla about inner classes? Before we get into it, we have to warn you (if you don't already know) that inner classes have inspired passionate love 'em or hate 'em debates since first introduced in version 1.1 of the

language. For once, we're going to try to keep our opinions to ourselves here and just present the facts as you'll need to know them for the exam. It's up to you to decide how—and to what extent—you should use inner classes in your own development. We mean it. We believe they have some powerful, efficient uses in very specific situations, including code that's easier to read and maintain, but they can also be abused and lead to code that's as clear as a cornfield maze and to the syndrome known as “reuseless”: *code that's useless over and over again*.

Inner classes let you define one class within another. They provide a type of scoping for your classes because you can make one class *a member of another class*. Just as classes have member *variables* and *methods*, a class can also have member *classes*. They come in several flavors, depending on how and where you define the inner class, including a special kind of inner class known as a “top-level nested class” (an inner class marked `static`), which technically isn't really an inner class. Because a static nested class is still a class defined within the scope of another class, we'll cover them in this chapter on inner classes. We'll also take another brief look at lambda expressions, which are often used as an alternative syntax (shorthand) for inner classes, so this is a good time to get more familiar with lambda expression syntax before we do a deep dive in the next chapter.

Many of the questions on the exam that make use of inner classes are focused on other certification topics and only use inner classes along the way. So in this chapter, we'll discuss the following four inner-class *topics*:

- Inner classes (“nested class” in the objective)
- Method-local inner classes (“local class” in the objective)
- Anonymous inner classes
- Static nested classes (“static inner class” in the objective)

The one certification objective directly related to inner classes is OCP Objective 2.3:

- Create inner classes including static inner class, local class, nested class, and anonymous inner class

which captures all the topics above in one objective.

## CERTIFICATION OBJECTIVE

## Nested Classes (OCP Objective 2.3)

2.3 *Create inner classes including static inner class, local class, nested class, and anonymous inner class.*

Note: As we've mentioned, mapping Objective 2.3 to this chapter is somewhat accurate, but it's also a bit misleading. You'll find inner classes used for many different exam topics. For that reason, we're not going to keep saying that this chapter is for Objective 2.3.

## Inner Classes

You're an OO programmer, so you know that for reuse and flexibility/extensibility, you need to keep your classes specialized. In other words, a class should have code *only* for the things an object of that particular type needs to do; any *other* behavior should be part of another class better suited for *that* job. Sometimes, though, you find yourself designing a class where you discover you need behavior that not only belongs in a separate specialized class, but also needs to be intimately tied to the class you're designing.

Event handlers are perhaps the best example of this (and are, in fact, one of the main reasons inner classes were added to the language in the first place). If you have a GUI class that performs some job, like, say, a chat client, you might want the chat-client-specific methods (accept input, read new messages from server, send user input back to server, and so on) to be in the class. But how do those methods get invoked in the first place? A user clicks a button. Or types some text in the input field. Or a separate thread doing the I/O work of getting messages from the server has messages that need to be displayed in the GUI. So you have chat-client-specific methods, but you also need methods for handling the “events” (button presses, keyboard typing, I/O available, and so on) that drive the calls on those chat-client methods. The ideal scenario—from an OO perspective—is to keep the chat-client-specific methods in the `ChatClient` class and put the event-handling *code* in a separate event-handling *class*.

*Nothing unusual about that so far; after all, that's how you're supposed to design OO classes. As specialists.* But here's the problem with the chat-client scenario: The event-handling code is intimately tied to the chat-client-specific code! Think about it: When users click a Send button (indicating that they want their typed-in message to be sent to the chat server), the chat-client code that sends the message needs to read from a *particular* text field. In other words, if

the user clicks Button A, the program is supposed to extract the text from the TextField B of a particular `ChatClient` instance. Not from some *other* text field from some *other* object, but specifically the text field that a specific instance of the `ChatClient` class has a reference to. So the event-handling code needs access to the members of the `ChatClient` object to be useful as a “helper” to a particular `ChatClient` instance.

And what if the `ChatClient` class needs to inherit from one class, but the event-handling code is better off inheriting from some *other* class? You can’t make a class extend more than one class, so putting all the code (the chat-client-specific code and the event-handling code) in one class won’t work in that case. So what you’d really like to have is the benefit of putting your event code in a separate class (better OO, encapsulation, and the ability to extend a class other than the class the `ChatClient` extends), but still allow the event-handling code to have easy access to the members of the `ChatClient` (so the event-handling code can, for example, update the `ChatClient`’s private instance variables). You *could* manage it by making the members of the `ChatClient` accessible to the event-handling class by, for example, marking them `public`. But that’s not a good solution either.

You already know where this is going—one of the key benefits of an inner class is the “special relationship” an *inner class instance* shares with *an instance of the outer class*. That “special relationship” gives code in the inner class access to members of the enclosing (outer) class, *as if the inner class were part of the outer class*. In fact, that’s exactly what it means: The inner class *is* a part of the outer class. Not just a “part,” but a full-fledged, card-carrying *member* of the outer class. Yes, an inner class instance has access to all members of the outer class, *even those marked private*. (Relax, that’s the whole point, remember? We want this separate inner class instance to have an intimate relationship with the outer class instance, but we still want to keep everyone *else* out. And besides, if you wrote the outer class, then you also wrote the inner class! So you’re not violating encapsulation; you *designed* it this way.)

## Coding a “Regular” Inner Class

We use the term *regular* here to represent inner classes that are not

- Static
- Method-local
- Anonymous

For the rest of this section, though, we'll just use the term "inner class" and drop the "regular." (When we switch to one of the other three types in the preceding list, you'll know it.) You define an inner class within the curly braces of the outer class:

```
class MyOuter {  
    class MyInner { }  
}
```

Piece of cake. And if you compile it:

```
%javac MyOuter.java
```

you'll end up with *two* class files:

```
MyOuter.class  
MyOuter$MyInner.class
```

The inner class is still, in the end, a separate class, so a separate class file is generated for it. But the inner class file isn't accessible to you in the usual way. You can't say

```
%java MyOuter$MyInner
```

in hopes of running the `main()` method of the inner class **because a regular inner class cannot have static declarations of any kind.** *The only way you can access the inner class is through a live instance of the outer class!* In other words, only at runtime, when there's already an instance of the outer class to tie the inner class instance to. You'll see all this in a moment. First, let's beef up the classes a little:

```
class MyOuter {  
    private int x = 7;  
  
    // inner class definition  
    class MyInner {
```

```
public void seeOuter() {  
    System.out.println("Outer x is " + x);  
}  
} // close inner class definition  
  
} // close outer class
```

The preceding code is perfectly legal. Notice that the inner class is, indeed, accessing a private member of the outer class. That's fine, because the inner class is also a member of the outer class. So just as any member of the outer class (say, an instance method) can access any other member of the outer class, private or not, the inner class—also a member—can do the same.

Okay, so now that we know how to write the code giving an inner class access to members of the outer class, how do you actually use it?

## Instantiating an Inner Class

To create an instance of an inner class, *you must have an instance of the outer class* to tie to the inner class. There are no exceptions to this rule: an inner class instance can never stand alone without a direct relationship to an instance of the outer class.

**Instantiating an Inner Class from Within the Outer Class** Most often, it is the outer class that creates instances of the inner class, since it is usually the outer class wanting to use the inner instance as a helper for its own personal use. We'll modify the `MyOuter` class to create an instance of `MyInner`:

```

class MyOuter {
    private int x = 7;
    public void makeInner() {
        MyInner in = new MyInner(); // make an inner instance
        in.seeOuter();
    }
}

class MyInner {
    public void seeOuter() {
        System.out.println("Outer x is " + x);
    }
}

```

You can see in the preceding code that the `MyOuter` code treats `MyInner` just as though `MyInner` were any other accessible class—it instantiates it using the class name (`new MyInner()`) and then invokes a method on the reference variable (`in.seeOuter()`). But the only reason this syntax works is because the outer class instance method code is doing the instantiating. In other words, *there's already an instance of the outer class—the instance running the `makeInner()` method.* So how do you instantiate a `MyInner` object from somewhere outside the `MyOuter` class? Is it even possible? (Well, since we're going to all the trouble of making a whole new subhead for it, as you'll see next, there's no big mystery here.)

### **Creating an Inner Class Object from Outside the Outer Class Instance**

**Code** Whew. Long subhead there, but it does explain what we're trying to do. If we want to create an instance of the inner class, we must have an instance of the outer class. You already know that, but think about the implications...it means that without a reference to an instance of the outer class, you can't instantiate the inner class from a static method of the outer class (because, don't forget, in static code, *there is no this reference*), or from any other code in any other class. Inner class instances are always handed an implicit reference to the outer class.

The compiler takes care of it, so you'll never see anything but the end result—the ability of the inner class to access members of the outer class. The code to make an instance from anywhere outside nonstatic code of the outer class is simple, but you must memorize this for the exam!

```
public static void main(String[] args) {  
    MyOuter mo = new MyOuter();      // gotta get an instance!  
    MyOuter.MyInner inner = mo.new MyInner();  
    inner.seeOuter();  
}
```

The preceding code is the same, regardless of whether the `main()` method is within the `MyOuter` class or some *other* class (assuming the other class has access to `MyOuter`, and since `MyOuter` has default access, that means the code must be in a class within the same package as `MyOuter`).

If you're into one-liners, you can do it like this:

```
public static void main(String[] args) {  
    MyOuter.MyInner inner = new MyOuter().new MyInner();  
    inner.seeOuter();  
}
```

You can think of this as though you're invoking a method on the outer instance, but the method happens to be a special inner class instantiation method, and it's invoked using the keyword `new`. Instantiating an inner class is the *only* scenario in which you'll invoke `new` *on* an instance as opposed to invoking `new` to *construct* an instance.

Here's a quick summary of the differences between inner class instantiation code that's *within* the outer class (but not `static`) and inner class instantiation code that's *outside* the outer class:

- From *inside* the outer class instance code, use the inner class name in the normal way:

```
MyInner mi = new MyInner();
```

- From *outside* the outer class instance code, the inner class name must now include the outer class's name:

`MyOuter.MyInner`

To instantiate it, you must use a reference to the outer class:

```
new MyOuter().new MyInner();
```

or

```
outerObjRef.new MyInner();
```

if you already have an instance of the outer class.

## Referencing the Inner or Outer Instance from Within the Inner Class

How does an object refer to itself normally? By using the `this` reference. Here is a quick review of this:

- The keyword `this` can be used only from within instance code. In other words, not within static code.
- The `this` reference is a reference to the currently executing object. In other words, the object whose reference was used to invoke the currently running method.
- The `this` reference is the way an object can pass a reference to itself to some other code as a method argument:

```
public void myMethod() {  
    MyClass mc = new MyClass();  
    mc.doStuff(this); // pass a ref to object running myMethod  
}
```

Within the inner class code, the `this` reference refers to the instance of the inner class, as you'd probably expect, since `this` always refers to the currently executing object. But what if the inner class code wants an explicit reference to

the outer class instance that the inner instance is tied to? In other words, *how do you reference the “outer this”?* Although normally, the inner class code doesn’t need a reference to the outer class, since it already has an implicit one it’s using to access the members of the outer class, it would need a reference to the outer class if it needed to pass that reference to some other code, as follows:

```
class MyInner {  
    public void seeOuter() {  
        System.out.println("Outer x is " + x);  
        System.out.println("Inner class ref is " + this);  
        System.out.println("Outer class ref is " + MyOuter.this);  
    }  
}
```

If we run the complete code as follows:

```

class MyOuter {
    private int x = 7;
    public void makeInner() {
        MyInner in = new MyInner();
        in.seeOuter();
    }
    class MyInner {
        public void seeOuter() {
            System.out.println("Outer x is " + x);
            System.out.println("Inner class ref is " + this);
            System.out.println("Outer class ref is " + MyOuter.this);
        }
    }
    public static void main (String[] args) {
        MyOuter.MyInner inner = new MyOuter().new MyInner();
        inner.seeOuter();
    }
}

```

the output is something like this:

```

Outer x is 7
Inner class ref is MyOuter$MyInner@113708
Outer class ref is MyOuter@33f1d7

```

So the rules for an inner class referencing itself or the outer instance are as follows:

- To reference the inner class instance itself from *within* the inner class

code, use `this`.

- To reference the “*outer this*” (the outer class instance) from within the inner class code, use `NameOfOuterClass.this` (example, `MyOuter.this`).

## Member Modifiers Applied to Inner Classes

A regular inner class is a member of the outer class just as instance variables and methods are, so the following modifiers can be applied to an inner class:

- `final`
- `abstract`
- `public`
- `private`
- `protected`
- `static`—*but static turns it into a static nested class, not an inner class*
- `strictfp`

## Method-Local Inner Classes

A regular inner class is scoped inside another class’s curly braces, but outside any method code (in other words, at the same level that an instance variable is declared). But you can also define an inner class within a method:

```
class MyOuter2 {  
    private String x = "Outer2";  
  
    void doStuff() {  
        class MyInner {  
            public void seeOuter() {  
                System.out.println("Outer x is " + x);  
            } // close inner class method  
        } // close inner class definition  
    } // close outer class method doStuff()  
  
} // close outer class
```

The preceding code declares a class, `MyOuter2`, with one method, `doStuff()`. But *inside* `doStuff()`, another class, `MyInner`, is declared, and it has a method of its own, `seeOuter()`. The previous code is completely useless, however, because *it never instantiates the inner class!* Just because you *declared* the class doesn't mean you created an *instance* of it. So to *use* the inner class, you must make an instance of it somewhere *within the method but below the inner class definition* (or the compiler won't be able to find the inner class). The following legal code shows how to instantiate and use a method-local inner class:

```
class MyOuter2 {  
    private String x = "Outer2";  
    void doStuff() {
```

```

class MyInner {
    public void seeOuter() {
        System.out.println("Outer x is " + x);
    } // close inner class method
} // close inner class definition

MyInner mi = new MyInner(); // This line must come
                           // after the class
mi.seeOuter();
} // close outer class method doStuff()
} // close outer class

```

## What a Method-Local Inner Object Can and Can't Do

A *method-local inner class can be instantiated only within the method where the inner class is defined*. In other words, no other code running in any other method—inside or outside the outer class—can ever instantiate the method-local inner class. Like regular inner class objects, the method-local inner class object shares a special relationship with the enclosing (outer) class object and can access its private (or any other) members. However, *the inner class object cannot use the local variables of the method the inner class is in*. Why not?

Think about it. The local variables of the method live on the stack and exist only for the lifetime of the method. You already know that the scope of a local variable is limited to the method the variable is declared in. When the method ends, the stack frame is blown away and the variable is history. But even after the method completes, the inner class object created within it might still be alive on the heap if, for example, a reference to it was passed into some other code and then stored in an instance variable. Because the local variables aren't guaranteed to be alive as long as the method-local inner class object is, the inner class object can't use them. *Unless the local variables are marked final or are effectively final!* The following code attempts to access a local variable from within a method-local inner class:

```
class MyOuter2 {  
    private String x = "Outer2";  
    void doStuff() {  
        String z = "local variable";  
        class MyInner {  
            public void seeOuter() {  
                System.out.println("Outer x is " + x);  
                System.out.println("Local var z is " + z);  
                z = "changing the local variable"; // Won't compile!  
            } // close inner class method  
        } // close inner class  
        x = "Changing Outer2";  
        MyInner mi = new MyInner();  
        mi.seeOuter();  
    } // close outer class doStuff() method  
    public static void main(String args[]) {  
        MyOuter2 mo2 = new MyOuter2();  
        mo2.doStuff();  
    }  
} // close outer class
```

Compiling the preceding code *really* upsets the compiler:

Local variable z defined in an enclosing scope must be final or effectively final

Removing the line that changes z fixes the problem, and marking the local

variable `z` as `final`, although optional, is a good reminder that we can't change it if we want to be able to use `z` in `seeOuter()`:

```
class MyOuter2 {  
    private String x = "Outer2";  
    void doStuff() {  
        final String z = "local variable"; // now MyInner can use z!  
        class MyInner {  
            public void seeOuter() {  
                System.out.println("Outer x is " + x);  
                System.out.println("Local var z is " + z);  
                // we removed the line that changed z  
            } // close inner class method  
        } // close inner class  
        x = "Changing Outer2";  
        MyInner mi = new MyInner();  
        mi.seeOuter();  
    } // close outer class doStuff() method  
} // close outer class doStuff() method  
public static void main(String args[]) {  
    MyOuter2 mo2 = new MyOuter2();  
    mo2.doStuff();  
}
```

Notice that even though `x` is not `final`, or effectively `final`, and we also use `x` in the `seeOuter()` method of `MyInner`, that's fine because `x` is a field of `MyOuter2`, not a local variable of the method `doStuff()`.

Just a reminder about modifiers within a method: The same rules apply to method-local inner classes as to local variable declarations. You can't, for example, mark a method-local inner class `public`, `private`, `protected`, `static`, `transient`, and the like. For the purpose of the exam, the only modifiers you *can* apply to a method-local inner class are `abstract` and `final`, but, as always, never both at the same time.



***Remember that a local class declared in a static method has access to only static members of the enclosing class, since there is no associated instance of the enclosing class. If you're in a static method, there is no this, so an inner class in a static method is subject to the same restrictions as the static method. In other words, no access to instance variables.***

## Anonymous Inner Classes

So far, we've looked at defining a class within an enclosing class (a regular inner class) and within a method (a method-local inner class). We're now going to look at the most unusual syntax you might ever see in Java: inner classes declared without any class name at all (hence, the word *anonymous*). And if that's not weird enough, you can define these classes, not just within a method, but even within an *argument* to a method. We'll look first at the *plain-old* (as if there is such a thing as a plain-old anonymous inner class) version (actually, even the plain-old version comes in two flavors), then at the argument-declared anonymous inner class, and finally at anonymous inner classes expressed with lambdas.

Perhaps your most important job here is to *learn to not be thrown when you see the syntax*. The exam is littered with anonymous inner class code—you might see it on questions about threads, wrappers, overriding, garbage collection, and...well, you get the idea.

### Plain-Old Anonymous Inner Classes, Flavor One

Check out the following legal-but-strange-the-first-time-you-see-it code:

```
class Popcorn {  
    public void pop() {  
        System.out.println("popcorn");  
    }  
}  
class Food {  
    Popcorn p = new Popcorn() {  
        public void pop() {  
            System.out.println("anonymous popcorn");  
        }  
    };  
}
```

Let's look at what's in the preceding code:

- We define two classes: Popcorn and Food.
- Popcorn has one method: `pop()`.
- Food has one instance variable, declared as type Popcorn. That's it for Food. Food has *no* methods.

And here's the big thing to get: the Popcorn reference variable refers, *not* to an instance of Popcorn, but to *an instance of an anonymous (unnamed) subclass of Popcorn*.

Let's look at just the anonymous class code:

```
2. Popcorn p = new Popcorn() {  
3.     public void pop() {  
4.         System.out.println("anonymous popcorn");  
5.     }  
6. };
```

**Line 2** Line 2 starts out as an instance variable declaration of type Popcorn. But instead of looking like this:

Popcorn p = new Popcorn(); // notice the semicolon at the end  
there's a curly brace at the end of line 2, where a semicolon would normally be.

Popcorn p = new Popcorn() { // a curly brace, not a semicolon

You can read line 2 as saying,

Declare a reference variable, p, of type Popcorn. Then declare a new class that has no name but that is a *subclass* of Popcorn. And here's the curly brace that opens the class definition...

**Line 3** Line 3, then, is actually the first statement within the new class definition. And what is it doing? Overriding the pop() method of the superclass Popcorn. This is the whole point of making an anonymous inner class—to *override one or more methods of the superclass!* (Or to implement methods of an interface, but we'll save that for a little later.)

**Line 4** Line 4 is the first (and, in this case, *only*) statement within the overriding pop() method. Nothing special there.

**Line 5** Line 5 is the closing curly brace of the pop() method. Nothing special.

**Line 6** Here's where you have to pay attention: Line 6 includes a *curly brace closing off the anonymous class definition* (it's the companion brace to the one on line 2), but there's more! Line 6 also has *the semicolon that ends the statement started on line 2*—the statement where it all began—the statement declaring and initializing the Popcorn reference variable. And what you're left with is a Popcorn reference to a brand-new *instance* of a brand-new, Just-In-

Time, anonymous (no name) *subclass* of Popcorn.



***The closing semicolon is hard to spot. Watch for code like this:***

```
2. Popcorn p = new Popcorn() {  
3.     public void pop() {  
4.         System.out.println("anonymous popcorn");  
5.     }  
6. } // Missing the semicolon needed to end  
      // the statement started on 2!  
7. Foo f = new Foo();
```

***You'll need to be especially careful about the syntax when inner classes are involved, because the code on line 6 looks perfectly natural. It's rare to see semicolons following curly braces.***

Polymorphism is in play when anonymous inner classes are involved. Remember that, as in the preceding Popcorn example, we're using a superclass reference variable type to refer to a subclass object. What are the implications? You can only call methods on an anonymous inner class reference that are defined in the reference variable type! This is no different from any other polymorphic references—for example,

```
class Horse extends Animal {  
    void buck() { }  
}  
class Animal {  
    void eat() { }  
}  
class Test {  
    public static void main (String[] args) {  
        Animal h = new Horse();  
        h.eat(); // Legal, class Animal has an eat() method  
        h.buck(); // Not legal! Class Animal doesn't have buck()  
    }  
}
```

So on the exam, you must be able to spot an anonymous inner class that—rather than overriding a method of the superclass—defines its own new method. The method definition isn't the problem, though; the real issue is, how do you invoke that new method? The reference variable type (the superclass) won't know anything about that new method (defined in the anonymous subclass), so the compiler will complain if you try to invoke any method on an anonymous inner class reference that is not in the superclass class definition.

Check out the following **illegal** code:

```
class Popcorn {
    public void pop() {
        System.out.println("popcorn");
    }
}

class Food {
    Popcorn p = new Popcorn() {
        public void sizzle() {
            System.out.println("anonymous sizzling popcorn");
        }
        public void pop() {
            System.out.println("anonymous popcorn");
        }
    };
}

public void popIt() {
    p.pop();      // OK, Popcorn has a pop() method
    p.sizzle();  // Not Legal! Popcorn does not have sizzle()
}
}
```

Compiling the preceding code gives us something like this:

```
Anon.java:19: cannot resolve symbol
symbol  : method sizzle  ()
location: class Popcorn
    p.sizzle();
               ^

```

which is the compiler's way of saying, "I can't find method `sizzle()` in class `Popcorn`," followed by, "Get a clue."

## Plain-Old Anonymous Inner Classes, Flavor Two

The only difference between flavor one and flavor two is that flavor one creates an anonymous *subclass* of the specified *class* type, whereas flavor two creates an anonymous *implementer* of the specified *interface* type. In the previous examples, we defined a new anonymous subclass of type `Popcorn` as follows:

```
Popcorn p = new Popcorn() {
```

But if `Popcorn` were an *interface* type instead of a *class* type, then the new anonymous class would be an *implementer* of the *interface* rather than a *subclass* of the *class*. Look at the following example:

```
interface Cookable {
    public void cook();
}

class Food {
    Cookable c = new Cookable() {
        public void cook() {
            System.out.println("anonymous cookable implementer");
        }
    };
}
```

The preceding code, like the Popcorn example, still creates an instance of an anonymous inner class, but this time, the new Just-In-Time class is an implementer of the `Cookable` interface. And note that this is the only time you will ever see the syntax:

```
new Cookable()
```

where `Cookable` is an *interface* rather than a non-abstract class type. Think about it: *You can't instantiate an interface*, yet that's what the code *looks* like it's doing. But, of course, it's not instantiating a `Cookable` object—it's creating an instance of a new anonymous implementer of `Cookable`. You can read this line:

```
Cookable c = new Cookable() {
```

as “Declare a reference variable of type `Cookable` that, obviously, will refer to an object from a class that implements the `Cookable` interface. But, oh yes, we don't yet *have* a class that implements `Cookable`, so we're going to make one right here, right now. We don't need a name for the class, but it will be a class that implements `Cookable`, and this curly brace starts the definition of the new implementing class.”

One more thing to keep in mind about anonymous interface implementers —*they can implement only one interface*. There simply isn't any mechanism to say that your anonymous inner class is going to implement multiple interfaces. In fact, an anonymous inner class can't even extend a class and implement an interface at the same time. The inner class has to choose either to be a subclass of a named class—and not directly implement any interfaces at all—or to implement a single interface. By directly, we mean actually using the keyword `implements` as part of the class declaration. If the anonymous inner class is a subclass of a class type, it automatically becomes an implementer of any interfaces implemented by the superclass.



***Don't be fooled by any attempts to instantiate an interface except in the case of an anonymous inner class. The following is not legal:***

```
Runnable r = new Runnable(); // can't instantiate interface
```

*whereas the following is legal, because it's instantiating an implementer of the Runnable interface (an anonymous implementation class):*

```
Runnable r = new Runnable() { // curly brace, not semicolon
    public void run() { }
};
```

## Argument-Defined Anonymous Inner Classes

If you understood what we've covered so far in this chapter, then this last part will be simple. If you *are* still a little fuzzy on anonymous classes, however, then you should reread the previous sections. If they're not completely clear, we'd like to take full responsibility for the confusion. But we'll be happy to share.

Okay, if you've made it to this sentence, then we're all going to assume you understood the preceding section, and now we're just going to add one new twist. Imagine the following scenario. You're typing along, creating the Perfect Class, when you write code calling a method on a Bar object and that method takes an object of type Foo (an interface).

```
class MyWonderfulClass {
    void go() {
        Bar b = new Bar();
        b.doStuff(ackWeDoNotHaveAFoo!); // Don't try to compile this at home
    }
}
interface Foo {
    void foof();
}
class Bar {
    void doStuff(Foo f) { }
}
```

No problemo, except that you don't *have* an object from a class that implements *Foo*, and you can't instantiate one, either, because *you don't even have a class that implements Foo*, let alone an instance of one. So you first need a class that implements *Foo*, and then you need an instance of that class to pass to the *Bar* class's *doStuff()* method. Savvy Java programmer that you are, you simply define an anonymous inner class *right inside the argument*. That's right, just where you least expect to find a class. And here's what it looks like:

```
1. public class MyWonderfulClass {  
2.     void go() {  
3.         Bar b = new Bar();  
4.         b.doStuff(new Foo() {  
5.             public void foof() {  
6.                 System.out.println("foofy");  
7.             } // end foof() method  
8.         }); // end inner class def, arg, and b.doStuff stmt.  
9.     } // end go()  
10. } // end class  
11.  
12. interface Foo {  
13.     void foof();  
14. }  
15. class Bar {  
16.     void doStuff(Foo f) {  
17.         f.foof();  
18.     };  
19. }
```

All the action starts on line 4. We're calling `doStuff()` on a `Bar` object, but the method takes an instance that IS-A `Foo`, where `Foo` is an interface. So we must make both an *implementation* class and an *instance* of that class, all right here in the argument to `doStuff()`. So that's what we do. We write

```
new Foo() {
```

to start the new class definition for the anonymous class that implements the `Foo` interface. `Foo` has a single method to implement, `foof()`, so on lines 5, 6, and 7,

we implement the `foof()` method. Then on line 8—wooh!—more strange syntax appears. The first curly brace closes off the new anonymous class definition. But don't forget that this all happened as part of a method argument, so the closing parenthesis, `)`, finishes off the method invocation, and then we must still end the statement that began on line 4, so we end with a semicolon. Study this syntax! You will see anonymous inner classes on the exam, and you'll have to be very, very picky about the way they're closed. If they're *argument local*, they end like this:

```
} ) ;
```

but if they're just plain-old anonymous classes, then they end like this:

```
} ;
```

Regardless, be careful. Any question from any part of the exam might involve anonymous inner classes as part of the code.

To run this code, simply create a new `MyWonderfulClass` and call its `go()` method:

```
MyWonderfulClass c = new MyWonderfulClass();  
  
c.go();
```

and you will see the output:

```
foofy
```

We'll come back to `MyWonderfulClass` at the very end of the chapter to see how we can write the anonymous inner class as a lambda expression. But before we do that, one more variation on inner classes.

## Static Nested Classes

We saved the easiest variation on inner classes for last, as a kind of treat!

You'll sometimes hear static nested classes referred to as *static inner classes* (and that's the way they are referred to in OCP Objective 2.3), but they really

aren't inner classes at all based on the standard definition of an inner class. Whereas an inner class (regardless of the flavor) enjoys that *special relationship* with the outer class (or rather, the *instances* of the two classes share a relationship), a static nested class does not. It is simply a non-inner (also called "top-level") class scoped within another. So with static classes, it's really more about name-space resolution than about an implicit relationship between the two classes.

A static nested class is simply a class that's a static member of the enclosing class:

```
class BigOuter {  
    static class Nested { }  
}
```

The class itself isn't really "static"; there's no such thing as a static class. The *static* modifier in this case says that the nested class is *a static member of the outer class*. That means it can be accessed, as with other static members, *without having an instance of the outer class*.

## Instantiating and Using Static Nested Classes

You use standard syntax to access a static nested class from its enclosing class. The syntax for instantiating a static nested class from a nonenclosing class is a little different from a normal inner class and looks like this:

```
class BigOuter {  
    static class Nest {void go() { System.out.println("hi"); } }  
}
```

```
class Broom {  
    static class B2 {void goB2() { System.out.println("hi 2"); } }  
    public static void main(String[] args) {  
        BigOuter.Nest n = new BigOuter.Nest(); // both class names  
        n.go();  
        B2 b2 = new B2(); // access the enclosed class  
        b2.goB2();  
    }  
}
```

which produces

```
hi  
hi 2
```

### e x a m watch

***Just as a static method does not have access to the instance variables and nonstatic methods of the class, a static nested class does not have access to the instance variables and nonstatic methods of the outer class. Look for static nested classes with code that behaves like a nonstatic (regular inner) class.***

## CERTIFICATION OBJECTIVE

### Lambda Expressions as Inner Classes (OCP Objective 2.6)

2.6 *Create and use Lambda expressions.*

Inner classes are often used for short, quick implementations of a class. Flavor two of anonymous inner classes showed you how to implement an interface with an anonymous class, either as a separate statement (as in the `Cookable` example) or as an argument (as in the `MyWonderfulClass` example).

When implementing an interface with an anonymous inner class, you'll often have opportunities to use lambda expressions to make your code more concise. In fact, some say lambdas are another way of writing anonymous inner classes. As you'll see, they certainly seem perfect for this job.

Let's take another look at `MyWonderfulClass` as a first example:

```

public class MyWonderfulClass {
    public static void main(String[] args) {
        MyWonderfulClass c = new MyWonderfulClass();
        c.go();
    }
    void go() {
        Bar b = new Bar();
        b.doStuff(new Foo() {
            public void foof() {
                System.out.println("foofy");
            } // end foof() method
       }); // end inner class def, arg, and bo.doStuff stmt
    } // end go()
} // end class
interface Foo {
    void foof();
}
class Bar {
    void doStuff(Foo f) {
        f.foof();
    };
}

```

`Foo` is an interface, and we need an instance of a class that implements that interface to pass to the `doStuff()` method of `Bar`, so we use an anonymous inner class. Notice one important thing about `Foo`: it's an interface with *one abstract method*. Does that sound familiar (from the previous chapter)? Yes, `Foo` is a *functional interface*.

That means we can replace the entire inner class with a lambda expression. Let's go through the same process we did in the previous chapter (with `GenreSort`) to see how we can convert the `Foo` anonymous inner class into a lambda expression.

1. First we get any parameters from the abstract method in `Foo`, `foof()`; those become the parameters for the lambda. In this case we don't have any parameters, so we write:

`( )`

2. Then, we add an arrow:

`( ) ->`

3. Then we write the body of the lambda expression. What should the body be? Exactly the same as the body of the `foof()` method in the `Foo` instance we're passing to `doStuff()`. That is:

`( ) -> System.out.println("foofy")`

We don't need to use any curly braces (or a return) because there's only one statement in the body of `foof()`, so we can just copy the expression as is. `foof()` is `void` (no return value), so in this case, the lambda expression will not automatically return a value (Java knows that `foof()` is `void` because it says so in `Foo`, so it's smart enough to know not to generate a return). Notice that we don't need a semicolon on the expression on the right either; however, as you'll see shortly, when we write a lambda expression as part of a statement, we still need to end the statement it's part of with a semicolon.

Now, let's see how to use this lambda expression:

```
1. public class MyWonderfulClass {  
2.     void go() {  
3.         Bar b = new Bar();  
4.         b.doStuff(() -> System.out.println("foofy")); // lambda magic!  
5.     }  
6. }  
7. interface Foo {  
8.     void foof();  
9. }  
10. class Bar {  
11.     void doStuff(Foo f) {  
12.         f.foof();  
13.     };  
14. }
```

Wow, that's a lot easier to read, isn't it? And definitely more concise. It might take you a while to get the hang of reading lambda expressions, but once you do, you'll find getting rid of the extra stuff that goes along with an inner class is a good way to make your code more concise and readable.

The important line to look at is line 4. Compare it to the code using the anonymous inner class. This lambda expression is “standing in” for the `foof()` function in the instance of the class implementing `Foo`.

But wait! There's no instance here. There's only a lambda expression that looks a lot more like a function than an object. And `doStuff()` is expecting an instance. So what gives?

Because `Foo` is a functional interface and has only one abstract method, Java knows that the function you're supplying as the lambda expression must be the function that implements that abstract method, `foof()`. It knows that the argument to `doStuff()` must be an instance of a class that implements the `Foo` interface, but the only really important part of that instance is the implementation of that one abstract method. So we shortcut by eliminating all the extra fluff of creating that class and instantiating the object and just provide

the method, in the form of a lambda expression.

We can make the instance object—the instance of a class that implements the Foo interface—a bit more explicit by creating it separately and then passing the instance to doStuff():

```
void go() {  
    Bar b = new Bar();  
    b.doStuff(() -> System.out.println("foofy"));  
    // more explicitly obvious version below  
    Foo f = () -> System.out.println("foofy 2"); // create the lambda  
    b.doStuff(f);                                // pass to doStuff  
}
```

You'll see lambda expressions written this way on the exam, so get lots of practice reading and writing them this way. The syntax of lambda expressions takes some getting used to because it hides a whole bunch of stuff going on behind the scenes. The trick is to remember that the type of a lambda expression is a functional interface. And when used to simplify inner classes, the important thing to know for the exam is that you can substitute a lambda expression for an anonymous inner class whenever that class is implementing a functional interface.

## Comparator Is a Functional Interface

Let's take one more look at the DVD example from the previous chapter to help inner classes and lambda expressions sink in just a bit more.

In that example we created a class `GenreSort` that implemented `comparator`, which we could use with the `Collections.sort()` method or 473with the `sort()` method of our `dvdlist` `ArrayList`, both of which take a comparator as an argument.

That `GenreSort` class looked like this:

```
class GenreSort implements Comparator<DVDInfo> {  
    public int compare(DVDInfo dvd1, DVDInfo dvd2) {  
        return dvd1.getGenre().compareTo(dvd2.getGenre());  
    }  
}
```

First, let's turn this into an anonymous inner class. Here's the revised go() method from the TestDVD class:

```
public void go() {  
    populateList();  
    // Now the GenreSort comparator is made using an  
    // anonymous inner class  
    Comparator<DVDInfo> genreSort = new Comparator<DVDInfo>() {  
  
        public int compare(DVDInfo dvd1, DVDInfo dvd2) {  
            return dvd1.getGenre().compareTo(dvd2.getGenre());  
        } };  
    dvclist.sort(genreSort);      // use the comparator to sort  
}
```

This is an example of flavor two of anonymous inner classes: that is, we're creating an instance of a class that implements the Comparator interface. We store this instance in the variable genreSort. Just like before, we pass that instance to the sort() method, which uses it to sort the items in the dvclist ArrayList by calling the compare() method we implemented in the comparator.

Take a careful look at the code to see how we translated the `GenreSort` class from an outer class into an anonymous inner class.

`Comparator` is a functional interface, meaning it has one abstract method that we must implement to make an instance of a class that implements the `Comparator` interface. So, we can scrap that inner class entirely and use a lambda expression instead:

```
dvdlist.sort( (dvd1, dvd2) ->
    dvd1.getGenre().compareTo(dvd2.getGenre()) );
```

This code is no different from what you saw in the previous chapter, only now instead of replacing the outer class `GenreSort`, we're replacing the anonymous inner class. It's exactly the same idea.

Note that we could also write the code like this:

```
Comparator<DVDInfo> genreSort = (dvd1, dvd2) ->
    dvd1.getGenre().compareTo(dvd2.getGenre());
dvdlist.sort(genreSort);
```

On the exam, you'll see lambdas used both ways: passed directly as arguments and assigned to variables. Assigning the lambda to a variable first makes the type of the lambda more explicit, but the type can always be inferred from the type signature of the method you're passing the lambda into.

For the exam, remember that you'll still see plenty of inner classes, because not all inner classes can be replaced by lambda expressions. Lambda expressions stand in for methods in classes that implement a functional interface. Don't be tricked by interfaces that might look functional but aren't. We'll cover all the rules that determine exactly what constitutes a functional interface in the next chapter.

## CERTIFICATION SUMMARY

---

Inner classes will show up throughout the exam, in any topic, and these are some of the exam's hardest questions. You should be comfortable with the sometimes bizarre syntax and know how to spot legal and illegal inner class definitions.

We looked first at “regular” inner classes, where one class is a member of another. You learned that coding an inner class means putting the class definition of the inner class inside the curly braces of the enclosing (outer) class, but outside of any method or other code block. You learned that an inner class *instance* shares a special relationship with a specific *instance* of the outer class and that this special relationship lets the inner class access all members of the outer class, including those marked `private`. You learned that to instantiate an inner class, you *must* have a reference to an instance of the outer class.

Next, we looked at method-local inner classes—classes defined *inside* a method. The code for a method-local inner class looks virtually the same as the code for any other class definition, except that you can't apply an access modifier the way you can with a regular inner class. You learned why method-local inner classes must use `final` or effectively final local variables declared within the method—the inner class instance may outlive the stack frame, so the local variable might vanish while the inner class object is still alive. You saw that to *use* the inner class you need to instantiate it and that the instantiation must come *after* the class declaration in the method.

We also explored the strangest inner class type of all—the *anonymous* inner class. You learned that they come in two forms: normal and argument-defined. Normal, ho-hum, anonymous inner classes are created as part of a variable assignment, whereas argument-defined inner classes are actually declared, defined, and automatically instantiated *all within the argument to a method!* We covered the way anonymous inner classes can be either a subclass of the named class type or an *implementer* of the named interface. Finally, we looked at how polymorphism applies to anonymous inner classes: You can invoke on the new instance only those methods defined in the named class or interface type. In other words, even if the anonymous inner class defines its own new method, no code from anywhere outside the inner class will be able to invoke that method.

As if we weren't already having enough fun for one day, we pushed on to static nested classes, which really aren't inner classes at all. Known as *static* nested classes, a nested class marked with the `static` modifier is quite similar to any other non-inner class, except that to access it, the code must have access to both the nested and enclosing class. We saw that because the class is *static*, no instance of the enclosing class is needed, and thus the static nested class *does not share a special relationship with any instance of the enclosing class*. Remember, static inner classes can't access instance methods or variables of the enclosing class.

And finally, just to seal the fate of inner classes entirely, we showed how you can replace an anonymous inner class that's implementing a functional interface with a lambda expression. Using lambda expressions as shorthand for anonymous inner classes usually makes your code a lot more concise because you no longer have to write out the instance of the class that's implementing an interface; instead, you just supply the method that's standing in for the instance. Get ready for a lot more on this topic in the next chapter, but before we get there, practice what you've learned in this chapter with the two-minute drill and self test.



## TWO-MINUTE DRILL

Here are some of the key points from this chapter. Most are related to OCP Objective 2.3.

### Regular Inner Classes (OCP Objective 2.3)

- A “regular” inner class is declared *inside* the curly braces of another class, but *outside* any method or other code block.
- An inner class is a full-fledged member of the enclosing (outer) class, so it can be marked with an access modifier as well as the abstract or final modifiers. (Never both abstract and final together—remember that abstract *must* be subclassed, whereas final *cannot* be subclassed.)
- An inner class instance shares a special relationship with an instance of the enclosing class. This relationship gives the inner class access to *all* of the outer class’s members, including those marked private.
- To instantiate an inner class, you must have a reference to an instance of the outer class.
- From code within the enclosing class, you can instantiate the inner class using only the name of the inner class, as follows:

```
MyInner mi = new MyInner();
```

- From code outside the enclosing class’s instance methods, you can instantiate the inner class only by using both the inner and outer class names and a reference to the outer class, as follows:

```
MyOuter mo = new MyOuter();
MyOuter.MyInner inner = mo.new MyInner();
```

- From code within the inner class, the keyword this holds a reference to the inner class instance. To reference the *outer* this (in other words, the instance of the outer class that this inner instance is tied to), precede the keyword this with the outer class name, as follows: MyOuter.this;

### Method-Local Inner Classes (OCP Objective 2.3)

- A method-local inner class is defined within a method of the enclosing class.

- ❑ For the inner class to be used, you must instantiate it, and that instantiation must happen within the same method, but *after* the class definition code.
- ❑ A method-local inner class cannot use variables declared within the method (including parameters) unless those variables are marked `final` or are effectively final.
- ❑ The only modifiers you can apply to a method-local inner class are `abstract` and `final`. (Never both at the same time, though.)

## Anonymous Inner Classes (OCP Objective 2.3)

- ❑ Anonymous inner classes have no name, and their type must be either a subclass of the named type or an implementer of the named interface.
- ❑ An anonymous inner class is always created as part of a statement; don't forget to close the statement after the class definition with a curly brace. This is a rare case in Java, a curly brace followed by a semicolon.
- ❑ Because of polymorphism, the only methods you can call on an anonymous inner class reference are those defined in the reference variable class (or interface), even though the anonymous class is really a subclass or implementer of the reference variable type.
- ❑ An anonymous inner class can extend one subclass *or* implement one interface. Unlike nonanonymous classes (inner or otherwise), an anonymous inner class cannot do both. In other words, it cannot both extend a class *and* implement an interface, nor can it implement more than one interface.
- ❑ An argument-defined inner class is declared, defined, and automatically instantiated as part of a method invocation. The key to remember is that the class is being defined within a method argument, so the syntax will end the class definition with a curly brace, followed by a closing parenthesis to end the method call, followed by a semicolon to end the statement: `} );`

## Static Nested Classes (OCP Objective 2.3)

- ❑ Static nested classes are inner classes marked with the `static` modifier.
- ❑ A static nested class is *not* an inner class; it's a top-level nested class.

- Because the nested class is `static`, it does not share any special relationship with an instance of the outer class. In fact, you don't need an instance of the outer class to instantiate a `static` nested class.
- For the purposes of the exam, instantiating a `static` nested class requires using both the outer and nested class names as follows:  
`BigOuter.Nested n = new BigOuter.Nested();`
- A `static` nested class cannot access non-`static` members of the outer class because it does not have any implicit reference to the outer instance (in other words, the nested class instance does not get an *outer* `this` reference).

## Replacing Inner Classes with Lambda Expressions (OCP Objective 2.6)

- Lambda expressions are a good way to write anonymous inner classes that implement functional interfaces.
- Instead of writing out an instance of the class using an anonymous inner class like this:

```
b.doStuff(new Foo() {
    public void foof() { System.out.println("foofy"); }
});
```

we can replace the anonymous inner class with a lambda expression, like this:

```
b.doStuff(() -> System.out.println("foofy"));
```

- You can replace anonymous inner classes with lambda expressions *only* if the inner class implements a functional interface. Be careful on the exam; make sure the interface is really functional.



The following questions will help you measure your understanding of the dynamic and life-altering material presented in this chapter. Read all of the choices carefully. Take your time. Breathe.

1. Which are true about a static nested class? (Choose all that apply.)
  - A. You must have a reference to an instance of the enclosing class in order to instantiate it
  - B. It does not have access to nonstatic members of the enclosing class
  - C. Its variables and methods must be static
  - D. If the outer class is named `MyOuter` and the nested class is named `MyInner`, it can be instantiated using `new MyOuter.MyInner();`
  - E. It must extend the enclosing class
2. Given:

```
class Boo {  
    Boo(String s) {}  
    Boo() {}  
}  
class Bar extends Boo {  
    Bar() {}  
    Bar(String s) {super(s);}  
    void zoo() {  
        // insert code here  
    }  
}
```

Which statements create an anonymous inner class from within class `Bar`? (Choose all that apply.)

- A. Boo f = new Boo(24) { };
  - B. Boo f = new Bar() { };
  - C. Boo f = new Boo() {String s; };
  - D. Bar f = new Boo(String s) { };
  - E. Boo f = new Boo.Bar(String s) { };
3. Which are true about a method-local inner class? (Choose all that apply.)
- A. It must be marked `final`
  - B. It can be marked `abstract`
  - C. It can be marked `public`
  - D. It can be marked `static`
  - E. It can access private members of the enclosing class
4. Given:

```
1. public class TestObj {  
2.     public static void main(String[] args) {  
3.         Object o = new Object() {  
4.             public boolean equals(Object obj) {  
5.                 return true;  
6.             }  
7.         }  
8.         System.out.println(o.equals("Fred"));  
9.     }  
10. }
```

What is the result?

- A. An exception occurs at runtime

- B. true
- C. Fred
- D. Compilation fails because of an error on line 3
- E. Compilation fails because of an error on line 4
- F. Compilation fails because of an error on line 8
- G. Compilation fails because of an error on a line other than 3, 4, or 8

5. Given:

```
1. public class HorseTest {  
2.     public static void main(String[] args) {  
3.         class Horse {  
4.             public String name;  
5.             public Horse(String s) {  
6.                 name = s;  
7.             }  
8.         }  
9.         Object obj = new Horse("Zippo");  
10.        System.out.println(obj.name);  
11.    }  
12. }
```

What is the result?

- A. An exception occurs at runtime at line 10
- B. zippo
- C. Compilation fails because of an error on line 3
- D. Compilation fails because of an error on line 9
- E. Compilation fails because of an error on line 10

6. Given:

```
public abstract class AbstractTest {
    public int getNum() {
        return 45;
    }
    public abstract class Bar {
        public int getNum() {
            return 38;
        }
    }
    public static void main(String[] args) {
        AbstractTest t = new AbstractTest() {
            public int getNum() {
                return 22;
            }
        };
        AbstractTest.Bar f = t.new Bar() {
            public int getNum() {
                return 57;
            }
        };
        System.out.println(f.getNum() + " " + t.getNum());
    }
}
```

What is the result?

- A. 57 22
- B. 45 38

- C. 45 57
- D. An exception occurs at runtime
- E. Compilation fails

7. Given:

```
3. public class Tour {  
4.     public static void main(String[] args) {  
5.         Cathedral c = new Cathedral();  
6.         // insert code here  
7.         s.go();  
8.     }  
9. }  
10. class Cathedral {  
11.     class Sanctum {  
12.         void go() { System.out.println("spooky"); }  
13.     }  
14. }
```

Which, inserted independently at line 6, compiles and produces the output “spooky”? (Choose all that apply.)

- A. Sanctum s = c.new Sanctum();
- B. c.Sanctum s = c.new Sanctum();
- C. c.Sanctum s = Cathedral.new Sanctum();
- D. Cathedral.Sanctum s = c.new Sanctum();
- E. Cathedral.Sanctum s = Cathedral.new Sanctum();

8. Given:

```
5. class A { void m() { System.out.println("outer"); } }
6.
7. public class TestInners {
8.     public static void main(String[] args) {
9.         new TestInners().go();
10.    }
11.    void go() {
12.        new A().m();
13.        class A { void m() { System.out.println("inner"); } }
14.    }
15.    class A { void m() { System.out.println("middle"); } }
16. }
```

What is the result?

- A. inner
- B. outer
- C. middle
- D. Compilation fails
- E. An exception is thrown at runtime

9. Given:

```
3. public class Car {  
4.     class Engine {  
5.         // insert code here  
6.     }  
7.     public static void main(String[] args) {  
8.         new Car().go();  
9.     }  
10.    void go() {  
11.        new Engine();  
12.    }  
13.    void drive() { System.out.println("hi"); }  
14. }
```

Which, inserted independently at line 5, produces the output "hi"? (Choose all that apply.)

- A. { Car.drive(); }
- B. { this.drive(); }
- C. { Car.this.drive(); }
- D. { this.Car.this.drive(); }
- E. Engine() { Car.drive(); }
- F. Engine() { this.drive(); }
- G. Engine() { Car.this.drive(); }

**10.** Given:

```
3. public class City {  
4.     class Manhattan {  
5.         void doStuff() throws Exception { System.out.print("x "); }  
6.     }  
7.     class TimesSquare extends Manhattan {  
8.         void doStuff() throws Exception { }  
9.     }  
10.    public static void main(String[] args) throws Exception {  
11.        new City().go();  
12.    }  
13.    void go() throws Exception { new TimesSquare().doStuff(); }  
14. }
```

What is the result?

- A. x
- B. x x
- C. No output is produced
- D. Compilation fails due to multiple errors
- E. Compilation fails due only to an error on line 4
- F. Compilation fails due only to an error on line 7
- G. Compilation fails due only to an error on line 10
- H. Compilation fails due only to an error on line 13

**11.** Given:

```
3. public class Navel {  
4.     private int size = 7;  
5.     private static int length = 3;  
6.     public static void main(String[] args) {  
7.         new Navel().go();  
8.     }  
9.     void go() {  
10.        int size = 5;  
11.        System.out.println(new Gazer().adder());  
12.    }  
13.    class Gazer {  
14.        int adder() { return size * length; }  
15.    }  
16. }
```

What is the result?

- A. 15
- B. 21
- C. An exception is thrown at runtime
- D. Compilation fails due to multiple errors
- E. Compilation fails due only to an error on line 4
- F. Compilation fails due only to an error on line 5

**12.** Given:

```
3. import java.util.*;
4. public class Pockets {
5.     public static void main(String[] args) {
6.         String[] sa = {"nickel", "button", "key", "lint"};
7.         Sorter s = new Sorter();
8.         for(String s2: sa) System.out.print(s2 + " ");
9.         Arrays.sort(sa,s);
10.        System.out.println();
11.        for(String s2: sa) System.out.print(s2 + " ");
12.    }
13.    class Sorter implements Comparator<String> {
14.        public int compare(String a, String b) {
15.            return b.compareTo(a);
16.        }
17.    }
18. }
```

What is the result?

- A. Compilation fails

- B. button key lint nickel  
                      nickel lint key button
- C. nickel button key lint  
                      button key lint nickel
- D. nickel button key lint  
                      nickel button key lint
- E. nickel button key lint  
                      nickel lint key button
- F. An exception is thrown at runtime

13. Given:

```
import java.util.*;
public class Pockets2 {
    public static void main(String[] args) {
        String[] sa = {"nickel", "button", "key", "lint"};
        for (String s2: sa) System.out.print(s2 + " ");
        Arrays.sort(sa, (a, b) -> a.compareTo(b));
        System.out.println();
        for (String s2: sa) System.out.print(s2 + " ");
    }
}
```

What is the result?

- A. Compilation fails

- B. button key lint nickel  
                      nickel lint key button
- C. nickel button key lint  
                     button key lint nickel
- D. nickel button key lint  
                     nickel button key lint
- E. nickel button key lint  
                     nickel lint key button
- F. An exception is thrown at runtime

## A SELF TEST ANSWERS

Note: Most of the questions in this chapter relate to OCP Objective 2.3. We've talked about the actual mapping of inner class ideas to the exam, so we will NOT be citing Objective numbers in the answers to the questions in this chapter, except for the last question, which relates to Objective 2.6.

1.  **B** and **D** are correct. **B** is correct because a static nested class is not tied to an instance of the enclosing class, and thus can't access the nonstatic members of the class (just as a static method can't access nonstatic members of a class). **D** uses the correct syntax for instantiating a static nested class.  
 **A** is incorrect because static nested classes do not need (and can't use) a reference to an instance of the enclosing class. **C** is incorrect because static nested classes can declare and define nonstatic members. **E** is wrong because...it just is. There's no rule that says an inner or nested class has to extend anything.
2.  **B** and **C** are correct. **B** is correct because anonymous inner classes are no different from any other class when it comes to polymorphism. That

means you are always allowed to declare a reference variable of the superclass type and have that reference variable refer to an instance of a subclass type, which, in this case, is an anonymous subclass of Bar. Since Bar is a subclass of Boo, it all works. C uses correct syntax for creating an instance of Boo.

☒ A is incorrect because it passes an int to the Boo constructor, and there is no matching constructor in the Boo class. D is incorrect because it violates the rules of polymorphism; you cannot refer to a superclass type using a reference variable declared as the subclass type. The superclass doesn't have everything the subclass has. E uses incorrect syntax.

3.  B and E are correct. B is correct because a method-local inner class can be abstract, although it means a subclass of the inner class must be created if the abstract class is to be used (so an abstract method-local inner class is probably not useful). E is correct because a method-local inner class works like any other inner class—it has a special relationship to an instance of the enclosing class, thus it can access all members of the enclosing class.

☒ A is incorrect because a method-local inner class does not have to be declared final (although it is legal to do so). C and D are incorrect because a method-local inner class cannot be made public (remember—local variables can't be public) or static.

4.  G is correct. This code would be legal if line 7 ended with a semicolon. Remember that line 3 is a statement that doesn't end until line 7, and a statement needs a closing semicolon!

☒ A, B, C, D, E, and F are incorrect based on the program logic just described. If the semicolon were added at line 7, then answer B would be correct—the program would print true, the return from the equals() method overridden by the anonymous subclass of Object.

5.  E is correct. If you use a reference variable of type Object, you can access only those members defined in class Object.

☒ A, B, C, and D are incorrect based on the program logic just described.

6.  A is correct. You can define an inner class as abstract, which means you can instantiate only concrete subclasses of the abstract inner class. The object referenced by the variable t is an instance of an anonymous subclass of AbstractTest, and the anonymous class overrides the getNum() method to return 22. The object referenced by variable f is an instance of an

anonymous subclass of Bar, and the anonymous Bar subclass also overrides the getNum() method to return 57. Remember that to create a Bar instance, we need an instance of the enclosing AbstractTest class to tie to the new Bar inner class instance. AbstractTest can't be instantiated because it's abstract, so we created an anonymous subclass (non-abstract) and then used the instance of that anonymous subclass to tie to the new Bar subclass instance.

**B, C, D, and E** are incorrect based on the program logic just described.

7.  **D** is correct. It is the only code that uses the correct inner class instantiation syntax.

**A, B, C, and E** are incorrect based on the above text.

8.  **C** is correct. The "inner" version of class A isn't used because its declaration comes after the instance of class A is created in the go() method.

**A, B, D, and E** are incorrect based on the above text.

9.  **C** and **G** are correct. **C** is the correct syntax to access an inner class's outer instance method from an initialization block, and **G** is the correct syntax to access it from a constructor.

**A, B, D, E, and F** are incorrect based on the above text.

10.  **C** is correct. The inner classes are valid, and all the methods (including main()), correctly throw an exception, given that doStuff() throws an exception. The doStuff() in class TimesSquare overrides class Manhattan's doStuff() and produces no output.

**A, B, D, E, F, G, and H** are incorrect based on the above text.

11.  **B** is correct. The inner class Gazer has access to Navel's private static and private instance variables.

**A, C, D, E, and F** are incorrect based on the above text.

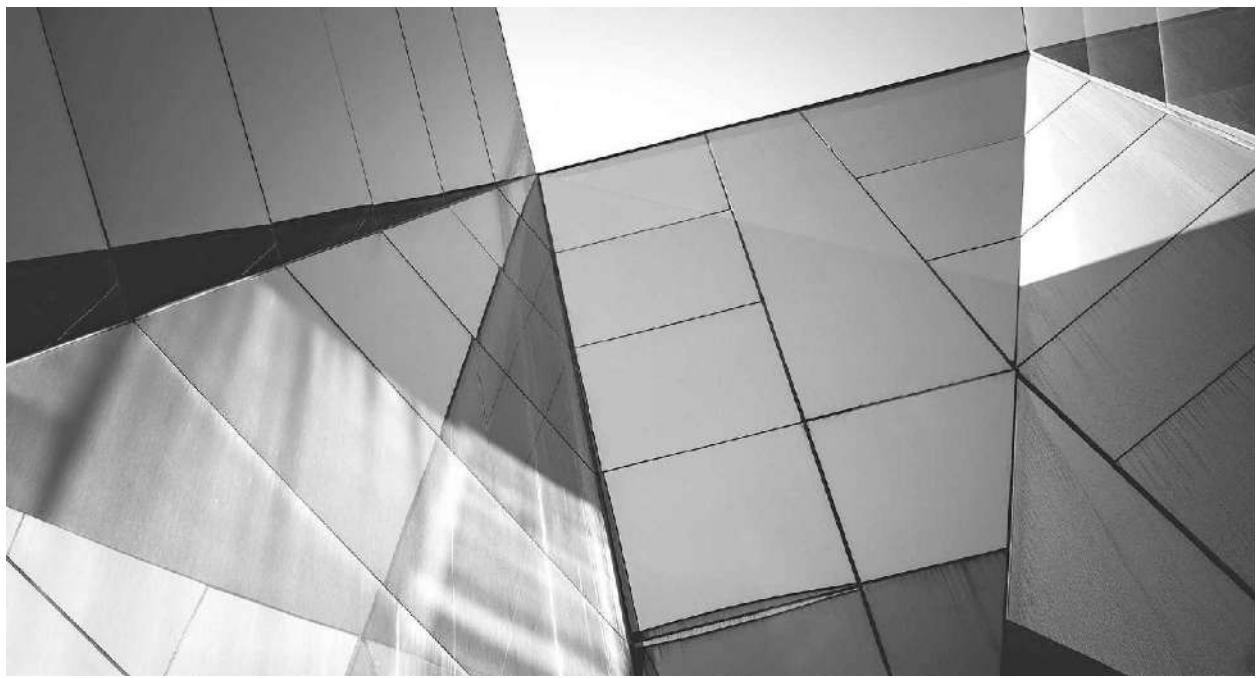
12.  **A** is correct. The inner class Sorter must be declared static to be called from the static method main(). If Sorter had been static, answer **E** would be correct.

**B, C, D, E, and F** are incorrect based on the above text.

13.  **C** is correct. We're using a lambda expression to stand in for the Comparator we pass to Arrays.sort(). There is no inner (or outer) class

we need to supply; the JDK knows we are supplying a lambda expression that implements the `compareTo()` method for `Comparator` because of the type signature of `Arrays.sort()`. In this case, we are sorting the list in ascending order.

- ☒ **A, B, D, E, and F** are incorrect based on the above text. (Objective 2.6)



# 8

## Lambda Expressions and Functional Interfaces

### CERTIFICATION OBJECTIVES

- Create and Use Lambda Expressions
  - Iterate Using forEach Methods of Streams and List
  - Use Method References with Streams
  - Use the Built-in Interfaces Included in the `java.util.function` Package such as `Predicate`, `Consumer`, `Function`, and `Supplier`
  - Develop Code That Uses Primitive Versions of Functional Interfaces
  - Develop Code That Uses Binary Versions of Functional Interfaces
  - Develop Code That Uses the `UnaryOperator` Interface
-  Two-Minute Drill

### Q&A Self Test

One of the big new language features added in Java 8 is the lambda expression. We've already talked a bit about lambdas, introducing the basic syntax and the idea of functional interfaces. You've seen how to use a lambda expression to replace an inner class (e.g., a `Comparator`) and you've also seen how to pass a lambda expression—really, just a block of code—to methods that expect Comparators, like `Collections.sort()`. Those earlier tastes of lambda expressions probably left you with more questions than answers, and this chapter is where we dive into all those details. As with inner classes, you'll need to know the syntax of lambda expressions and when you can use them like the back of your hand. You'll also need to get familiar with a

variety of functional interfaces so you can recognize them easily. Do that and you'll sail through the lambda expressions part of the exam.

## CERTIFICATION OBJECTIVE

### Lambda Expression Syntax (OCP Objective 2.6)

#### 2.6 *Create and use Lambda expressions.*

You already know (a little) how lambda expressions can make your code more concise. Concise code is good, but only if it makes sense. And, let's face it, lambda expression syntax can take some getting used to. Let's take another look at the syntax and talk about the varieties of lambda expression syntax you might expect to see in the real world and on the exam.

Imagine you've got a super simple interface, `DogQuerier`, with one abstract method, `test()` (remember that all interface methods are abstract unless they are declared default or static):

```
interface DogQuerier {  
    public boolean test(Dog d);  
}
```

We say that `DogQuerier` is a “functional interface”: it’s an interface with one abstract method.

Later, you use an inner class to define an instance of a class that implements this interface:

```
DogQuerier dq = new DogQuerier() {  
    public boolean test(Dog d) { return d.getAge() > 9; }  
};
```

Wait, scrap that—we can make the code much more concise by replacing that inner class with a lambda expression:

```
DogQuerier dq = (d) -> d.getAge() > 9;
```

You use that lambda expression just like you'd use the instance of the inner class:

```
System.out.println("Is Boi older than 9? " + dq.test(boi));
```

(assuming boi is a Dog with a getAge() method, of course).

Think of a lambda expression as a shorthand way of writing an instance of a class that implements a functional interface. It looks a lot like a method (in fact, some call lambda expressions “anonymous methods”), but it’s a bit more than that; it’s more like an instance with everything but the method stripped away. The important part of the instance is the method (the rest can be inferred from the interface definition), so the lambda expression is the syntax of the instance that’s been boiled down to the bare essentials.

To make the lambda expression, we copy the parameter of the test() method from the inner class, then write an arrow, and then copy the expression in the body of the test method, leaving out the return and the semicolon:

```
public boolean test(Dog d) { return d.getAge() > 9; }
```

$\begin{matrix} \swarrow & \uparrow & \searrow \\ \text{(d)} & \rightarrow & \text{d.getAge( ) > 9} \end{matrix}$

The type of the lambda expression is DogQuerier. That’s the same as the type of the interface and the same as the type of the instance being created by the inner class.

```
DogQuerier dq = (d) -> d.getAge() > 9;
```

And, of course, when we write the statement assigning the lambda to the instance variable, we end that statement with a semicolon. Now let’s look at

some variations. Because the `test()` method has only one parameter, it's perfectly legal to leave off the parentheses around the parameter and write the lambda like this instead:

```
DogQuerier dq = d -> d.getAge() > 9;
```

If you have more than one parameter, however, you *must* use the parentheses.

You might be wondering: how does the lambda "know" what `d`'s type is supposed to be? That can be inferred from the `DogQuerier` interface definition. However, there may be times when the type can't be inferred and you will need to write it in. And, in this example, you can supply the type if you want to, but if you do, you'll have to use the parentheses around the parameter:

```
DogQuerier dq = (Dog d) -> d.getAge() > 9;
```

What about the return value's type? That, too, can be inferred from the `DogQuerier` interface definition. And wait a sec, where'd that `return` go to anyway?

The rule is, if there is only one expression in the lambda, then the value of that expression gets returned by default, and you don't need a `return`. In fact, if you try to write

```
DogQuerier dq = d -> return d.getAge() > 9; // does not compile
```

you'll get a compile-time error:

Syntax error on token "->", { expected after this token.

If you want to write `return`, then you'll have to write the lambda like this:

```
DogQuerier dq = d -> { return d.getAge() > 9; };
```

In other words, if the body of your lambda is anything more than an expression—that is, a statement or multiple statements—you'll need to use the curly braces. Here's an example of a lambda expression with multiple statements:

```
DogQuerier dq = d -> {
    System.out.println("Testing " + d.getName());
    return d.getAge() > 9;
};
```

To summarize, we can write the original `DogQuerier` instance as a lambda expression in the following ways, all of which are equivalent:

```
DogQuerier dq = (d) -> d.getAge() > 9;
DogQuerier dq = d -> d.getAge() > 9;
DogQuerier dq = (Dog d) -> d.getAge() > 9;
DogQuerier dq = d -> { return d.getAge() > 9; };
```

Here's the full code so you can test the `DogQuerier` lambda expression:

```
// Our trusty Dog class
public class Dog {
    private String name;
    private int weight;
    private int age;
    public Dog(String name, int weight, int age) {
```

```
        this.name = name;
        this.weight = weight;
        this.age = age;
    }

    public String getName() { return this.name; }
    public int getWeight() { return this.weight; }
    public int getAge() { return this.age; }
    public String toString() { return this.name; }
}

// Our functional interface to test dogs
interface DogQuerier {
    public boolean test(Dog d);
}

public class TestDogs {
    public static void main(String[] args) {
        Dog boi = new Dog("boi", 30, 6);
        Dog clover = new Dog("clover", 35, 12);
        // We don't need this inner class anymore; replace with a lambda
        // DogQuerier dq = new DogQuerier() {
        //     public boolean test(Dog d) { return d.getAge() > 9; }
        // };
        DogQuerier dq = d -> d.getAge() > 9; // replaces the inner class
        System.out.println("Is Boi older than 9? " + dq.test(boi));
        System.out.println("Is Clover older than 9? " + dq.test(clover));
    }
}
```

The output is

```
Is Boi older than 9? false  
Is Clover older than 9? true
```

## e x a m Watch

*Here are a few examples of invalid lambda expression syntax to watch out for:*

```
Sheep s -> s.color.equals(color);
```

*Needs parentheses around the argument.*

```
(Sheep s) -> if (s.color.equals(color)) return true; else return false;
```

*Needs {} around body of lambda and an ending ;*

```
SheepQuerier testSheep = s, n -> s.age > n;
```

*Needs parentheses around the arguments.*

## Passing Lambda Expressions to Methods

Lambda expressions are easy to pass to methods. It's a bit like passing a block of code to a method. To demonstrate, let's add a class to our Dogs example:

```
class DogsPlay {  
    DogQuerier dogQuerier;  
    public DogsPlay(DogQuerier dogQuerier) {  
        this.dogQuerier = dogQuerier;  
    }  
    public boolean doQuery(Dog d) {  
        return dogQuerier.test(d);  
    }  
}
```

The constructor for the `DogsPlay` class takes a `DogQuerier` instance. We can pass the `dq` instance we created with a lambda expression to `DogsPlay` like this:

```
DogsPlay dp = new DogsPlay(dq);
```

Or we can pass a lambda expression directly:

```
DogsPlay dp = new DogsPlay(d -> d.getAge() > 9);
```

When we call `dp.doQuery()` and pass in a dog, the `test()` method of the `DogQuerier` gets called:

```
System.out.println("Is Boi older than 9? " + dp.doQuery(boi));
```

And we see the output:

```
Is Boi older than 9? false
```

The lambda expression we're passing to `DogsPlay` is simple; it has one `Dog` parameter and simply tests that dog's age and returns true or false.

## Accessing Variables from Lambda Expressions

What do you think happens if a lambda has a reference to another variable? While you can declare and use variables within the lambda expression, just like you would in the body of a method, the lambda is essentially just creating a

nested block, so you can't use the same variable name as you've used in the enclosing scope. So this is fine:

```
int numCats = 3;  
DogQuerier dqWithCats = d -> {  
    int numBalls = 1;          // completely new variable local to lambda  
    numBalls++;              // can modify numBalls  
    System.out.println("Number of balls: " + numBalls); // can access numBalls  
    System.out.println("Number of cats: " + numCats);    // can access numCats  
    return d.getAge() > 9;  
};
```

But this is not:

```
int numCats = 3;  
int numBalls = 1;          // now we have numBalls in enclosing scope  
DogQuerier dqWithCats = d -> {  
    int numBalls = 5;        // won't compile! Trying to redeclare numBalls  
    System.out.println("Number of balls: " + numBalls);  
    System.out.println("Number of cats: " + numCats);  
    return d.getAge() > 9;  
};
```

A lambda expression “captures” variables from the enclosing scope, so you can access those variables in the body of the lambda, but those variables must be final or effectively final. An effectively final variable is a variable or parameter whose value isn’t changed after it is initialized. Let’s see what happens if we try to modify the number of cats in the lambda expression:

```
int numCats = 3;
DogQuerier dqWithCats = d -> {
    int numBalls = 1;
    numBalls++;
    numCats++;           // Won't compile! Can't change numCats
    System.out.println("Number of balls: " + numBalls);
    System.out.println("Number of cats: " + numCats);
    return d.getAge() > 9;
};
```

If we try to change the value either in the lambda itself or elsewhere in the enclosing scope, we will get an error. We can *use* the value of numCats, but we can't *change* it. So this will work:

```
int numCats = 3;           // numCats is effectively final
DogQuerier dqWithCats = d -> {
    int numBalls = 1;
    numBalls++;
    System.out.println("Number of balls: " + numBalls);
    System.out.println("Number of cats: " + numCats); // Okay to use numCats
    return d.getAge() > 9;
};
```

The value of the variable numCats is captured by the lambda so it can be used later when we invoke the lambda (by calling its test( ) method). Let's see what happens when we pass a lambda with captured values to the DogsPlay constructor:

```
System.out.println("--- use DogsPlay ---");
DogsPlay dp = new DogsPlay(dqWithCats);
System.out.println("Is Clover older than 9? " + dp.doQuery(clover));
```

This works fine; the captured value for numCats is sent along with the lambda to DogsPlay and used when the lambda is invoked later when we call doQuery(). Here is the output:

```
--- use DogsPlay ---
Number of balls: 2
Number of cats: 3
Is Clover older than 9? True
```

## CERTIFICATION OBJECTIVE

### Functional Interfaces (OCP Objectives 3.5, 4.1, 4.2, 4.3, and 4.4)

3.5 *Iterate using forEach methods of Streams and List.*

4.1 *Use the built-in interfaces included in the java.util.function package such as Predicate, Consumer, Function, and Supplier.*

4.2 *Develop code that uses primitive versions of functional interfaces.*

4.3 *Develop code that uses binary versions of functional interfaces.*

4.4 *Develop code that uses the UnaryOperator interface.*

Lambda expressions work by standing in for instances of classes that implement interfaces with one abstract method, so there is no confusion about which method the lambda is defining. As we've said before, we call these interfaces with one and only one abstract method "functional interfaces." There's nothing particularly special about them, except their relationship to lambda expressions. We can explicitly identify a functional interface using the @FunctionalInterface annotation, like this:

```
@FunctionalInterface
interface DogQuerier {
    public boolean test(Dog d);
}
```

This annotation is not required but can be helpful when you want to ensure you don't inadvertently add methods to a functional interface that will then break other code. If you use the `DogQuerier` type to define a lambda expression and then later add another abstract method to this interface, without the `@FunctionalInterface` annotation, you'll get a compiler error, "The target type of this expression must be a functional interface." Yikes!

By using the `@FunctionalInterface`, the compiler will warn you that adding an extra method won't work and you can avoid the error. Seeing that annotation, you'll be reminded that you created a functional interface for a reason, so you'll know not to change it. You may not see `@FunctionalInterface` used on the exam, but you'll definitely be expected to identify functional interfaces with or without this annotation.

## Built-in Functional Interfaces

Functional interfaces turn out to be useful in all sorts of scenarios, some of which we'll get into here and some in the next chapter. You've already seen one useful functional interface: the `Comparator`. In addition, with Java 8 we get a big collection of functional interfaces in the `java.util.function` package, where you'll find `Predicate`, which you've also seen before, along with a variety of others. You will need to be familiar with them all.

Looking at the list, you might be wondering why you need all these different kinds of functional interfaces? Well, you probably won't ever need them all, but it's nice to have them available to use when you need a functional interface and don't want to define your own. As you'll see, there are many more uses for these interfaces than you might think, and having those interfaces already defined can save you time and code when you just want a quick lambda expression and don't want to bother with creating your own functional interface.

## What Makes an Interface Functional?

We already said that a functional interface is an interface with one and only one abstract method. Let's delve into this just a little more because it can get a bit tricky.

Take a look at the Java 8 API documentation (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html>) for `java.util.function.Predicate`, and you'll find it has five methods: `and()`, `isEqual()`, `negate()`, `or()`, and `test()`. Every method except `test()` is

declared as static or default, so it's pretty easy to determine that `test()` is the single abstract method in this functional interface. And, indeed, if you look under “Abstract Methods” for this interface, you’ll see one method there, `test()`. Clearly, `Predicate` is a functional interface.

Now, take a look at the documentation for `java.util.Comparator`. This has quite a few default and static methods, so click on “Abstract Methods” to narrow down your search for its one and only one abstract method. What do you see? Two methods: `compare()` and `equals()`. But we said this is a functional interface, so why are two abstract methods listed here? `equals()` is inherited from `Object`, and inherited public methods are not counted when you’re determining whether an interface is a functional interface. So even though `equals()` is abstract in `Comparator`, because it’s inherited, it doesn’t count. Tricky! Be ready to spot this on the exam.

To sum up, here is the rule for functional interfaces: A functional interface is an interface that has one abstract method. Default methods don’t count; static methods don’t count; and methods inherited from `Object` don’t count.

Oh, and just so you know, the single abstract method in a functional interface is called the “functional method.”

## Categories of Functional Interfaces

You’ll find 43 functional interfaces in `java.util.function`, and they all fall into one of four categories: suppliers, consumers, predicates, or functions. Each functional interface has a single abstract method (the functional method) and sometimes static and default methods.

The basic types of functional interfaces are `Supplier`, `Consumer`, `Predicate`, and `Function`.

**Suppliers** Suppliers supply results. `java.util.function.Supplier`’s functional method, `get()`, never takes an argument, and it always returns something.

**Consumers** Consumers consume values. `java.util.function.Consumer`’s functional method, `accept()`, always takes an argument and never returns anything.

**Predicates** Predicates test things (and do logical operations). `java.util.function.Predicate`’s functional method, `test()`, takes a value, does a logical test, and returns true or false.

**Functions** You can think of functions as the most generic of the functional interfaces. `java.util.function.Function`'s functional method, `apply()`, takes an argument and returns a value.

All of the functional interfaces in `java.util.function` are variations of suppliers, consumers, predicates, and functions. For example, `BooleanSupplier` is a supplier whose functional method supplies a boolean value. An `IntConsumer` is a consumer whose functional method takes an `int` value. And so on. Most of the variations are either to give you a way to provide more arguments (e.g., a `BiConsumer` that takes two arguments instead of one) or avoid autoboxing (e.g., `IntConsumer` and `BooleanSupplier`), which makes them more efficient. We'll give a lot of examples in the next few sections, so you get a sense of how these work.

When you first start working with functional interfaces, they can seem rather abstract. That's because, on their own, they don't really mean anything. What on earth is the point of having a `Function` interface with a method that takes a value and returns a value? That seems like the most generic thing in the world and rather pointless.

Seeing these interfaces in use and using them yourself will help make them seem less abstract and more useful. One good way to see functional interfaces in use is to look at the Oracle API documentation for one of the interfaces and click on "USE" in the top navigation bar. This shows you how that functional interface is used in the rest of the JDK.

## Working with Suppliers

Let's begin with a super simple supplier:

```
Supplier<Integer> answerSupplier = () -> 42;  
System.out.println("Answer to everything: " + answerSupplier.get());
```

And in the output we see

```
Answer to everything: 42
```

We've made a `Supplier` whose functional method, `get()` returns an `Integer` object, 42.

Looking at the definition of `java.util.function.Supplier`, we can see that it's defined like this:

```
@FunctionalInterface  
public interface Supplier<T>
```

And its functional method is `get()`, which returns a value of type  $\tau$ . So here  $\tau$  is a type parameter indicating the type of what's returned from the functional method. We like the method detail from the documentation page where it describes what the `get()` method returns: "A result." About as generic as you can get, right? This is why functional interfaces can feel so abstract when you first start working with them.

Just to cement how the lambda expression is being used for `answerSupplier` in the code above, let's take a look at how we might create a `Supplier` without using a lambda expression, so you can see one more time how we get to the lambda from, say, an inner class. Remember, `Supplier` is just an interface! So we can make an instance of a class implementing that interface the old-fashioned way, with an inner class, like this:

```
Supplier<Integer> answerSupplierInnerClass = new Supplier<Integer>() {  
    public Integer get() {  
        return 42;  
    }  
};
```

As you can see, using a lambda expression to stand in for this inner class is a whole lot shorter and easier. Hopefully by this point, you've got the hang of reading lambda expressions. From here on out, we'll primarily use lambda expressions when implementing functional interfaces. That said, remember you might see a full-blown implementation of a functional interface on the exam.

Okay, so a supplier that supplies 42 every time isn't that interesting; let's try another one:

```

Supplier<String> userSupplier = () -> {
    Map<String, String> env = System.getenv(); // get the system environment map
    return env.get("USER"); // get the value with the key
        // "USER" from the map and
        // return it (Note: on Windows,
        // this key is "USERNAME")
};

System.out.println("User is: " + userSupplier.get());

```

Here we've got a supplier with multiple statements, so we're using a slightly longer form of the lambda expression with curly braces to define a block of code and including a `return` statement to return the `String` that this supplier expects. In this case the `String` that's returned is the system username.

The key thing to note about both of these suppliers is they take no arguments and return an object. That's what suppliers do.

**IntSupplier** Looking at `java.util.function`, we can see that there are some variations on supplier, including `IntSupplier`, `DoubleSupplier`, and `LongSupplier`. As you can guess, these supply an `int`, a `double`, and a `long`, respectively. These are there primarily to avoid autoboxing in case you want a primitive, rather than an object, back from the supplier. The functional method names of each of these is different and is not `get()`, so you'll need to remember that for the exam. For example, the functional method of `IntSupplier` is `getAsInt()`, so we can use the following code to make a supplier that returns a new random `int` when you call the `getAsInt()` method:

```

Random random = new Random();
IntSupplier randomIntSupplier = () -> random.nextInt(50);
int myRandom = randomIntSupplier.getAsInt();
System.out.println("Random number: " + myRandom);

```

Notice that `IntSupplier` doesn't use a type parameter, because the functional method returns a primitive. This avoids the autoboxing to `Integer` that we get with `Supplier<Integer>` (as in our `answerSupplier` above), which saves just a

tiny bit of computation time (woo hoo!).

**What's the Point of Supplier?** At this stage, you might be asking yourself, What is the point of a supplier? After all we could just as easily write 42, or `random.nextInt(50)`, or put the block of code into a regular method, instead of creating a lambda expression and then calling its functional method.

Looking at how Suppliers are used in the JDK can give you a hint as to where and why they are useful. As an example we'll take a look at the `java.util.logging.Logger` class, which has been augmented with several methods that take `Supplier` as an argument. `Logger` is used to log messages for a system or application component, and you can set the logging level to determine what kinds of messages and how many messages to log. We can use the `log()` method to log a string if the log level is set, like this:

```
Logger logger = Logger.getLogger("Status Logger");
logger.setLevel(Level.SEVERE);

// Later...
String currentStatus = "Everything's okay";
logger.log(Level.INFO, currrentStatus);
```

In this example, we do *not* see the current status string logged because the log level is set to `SEVERE` and our call to the `log()` method says only log the message if the level is set to `INFO` or below (logging levels are ordered with `SEVERE` at the highest level, and `INFO` is a couple of levels below that). If we write this instead

```
String currentStatus = "Something's horribly wrong!";
logger.log(Level.SEVERE, currentStatus);
```

then we will see the message.

Now what if we need to do some expensive call to check the status of the system? Here we've hard-coded the status to a `String` but it's more likely that the status is actually determined by, say, making a network call to see if a system is up or down. If we go ahead and compute the status before sending a string to the `log()` method, then we've potentially wasted time checking the status if the log level is not set high enough that we'll actually log the status string. This is

where `Supplier` comes in handy.

A new `log()` method in `Logger` takes a level and a `Supplier<String>` rather than a `String`. Let's see how we might use this. We'll write some code that will determine the status of a system by sending a network request to a host to check to see if it's up and running. Imagine this is part of a bigger program that does something useful with the data from the host (in this case, `javaranch.com`); perhaps it alerts you every time someone posts a new message.

We want to log the status—that is, whether the host is up—when things are fine, but only if the logging level is set to `INFO` or below. We don't want to bother checking the status and logging anything if the logging level is set to `SEVERE`. In that case, we only want to log messages that indicate things are really bad!

And we want to log the status if things go wrong, pretty much no matter what the logging level is, so we'll log the status if the level is set to `SEVERE` or below (and since `SEVERE` is the highest level, then we'll *always* log the status if things go wrong).

Here's the code:

```
String host = "coderanch.com";
int port = 80;
// set up logging
Logger logger = Logger.getLogger("Status Logger");
logger.setLevel(Level.SEVERE); // line 5

// in case we need to check the status
Supplier<String> status = () -> {
    int timeout = 1000;
    try (Socket socket = new Socket()) {
        socket.connect(new InetSocketAddress(host, port), timeout);
        return "up";
    } catch (IOException e) {
```

```

        return "down";                                // Error; can't reach the system!
    }
};

try {
    logger.log(Level.INFO, status);               // only calls the get() method of the
                                                // status Supplier if level is INFO
                                                // or below
    // do stuff with coderanch.com
    // ...
} catch (Exception e) {
    logger.log(Level.SEVERE, status);           // calls the get() method of the status
                                                // Supplier if level is SEVERE or below
}

```

The logger's `log()` method takes the `status Supplier` and checks the log level. Because we've set the log level to `SEVERE` on line 5, in the `try` block, where we call the `log()` method with `Level.INFO`, the `log()` method won't bother calling the `get()` method of the `status Supplier`, and so we avoid making that expensive network call. And, of course, in the `catch` block, we're passing `Level.SEVERE` with the `Supplier`: the `log()` method will call the `status Supplier`'s `get()` method, get the `status` (a `String`—look at the type parameter on the `Supplier`), and log it.

Imagine if you were to always check the `status` to get the `String` to pass to `log()` instead. In that case, you'd be checking the `status` unnecessarily in the case where everything's fine and the logging level is set to `SEVERE`.

So using a `Supplier` here avoids that expensive operation when it's unnecessary. We're passing a block of code (the `Supplier`) that gets executed only if a certain condition applies. We don't have to check that condition ourselves; `Logger` does it for us and then can call the `get()` method on the `Supplier` only if it needs to. (You can test the `log` status yourself by adding a

```
throw new IOException();
```

in the `try` block if you want).

It's no different than if you created your own `Supplier` interface with a `get()` method that could be called whenever a value is needed, but having these built-in functional interfaces that other parts of the JDK can use in situations like this makes life just a bit easier for you. And using lambda expressions to eliminate one more step (actually instantiating a class that implements that interface) reduces your work, helps make your code more concise, and (once you're used to reading lambda expressions) easier to read.

## Working with Consumers

You can think of consumers as the opposite of suppliers (like matter and antimatter, it helps keeps the universe balanced). Consumers accept one or more arguments and don't return anything. So this lambda expression

```
Consumer<String> redOrBlue = pill -> {
    if (pill.equals("red")) {
        System.out.println("Down the rabbit hole");
    } else if (pill.equals("blue")) {
        System.out.println("Stay in lala land");
    }
};
```

implements the `java.util.function.Consumer` interface, which is defined like this:

```
@FunctionalInterface
public interface Consumer<T>
```

The `Consumer`'s functional method is `accept()`, which takes an object of type `T` and returns nothing, so we use the `redOrBlue` consumer like this:

```
redOrBlue.accept("red");
```

and see the output:

## Down the rabbit hole

As with suppliers, there are variations on consumers in the `java.util.function` package. `IntConsumer`, `DoubleConsumer`, and `LongConsumer` do what you'd expect: their `accept()` methods take one primitive argument and avoid the autoboxing you get with `Consumer`.

In addition to these, there's `ObjIntConsumer`, `ObjDoubleConsumer`, and `ObjLongConsumer`, whose `accept()` methods take an object (type `T`) and an `int`, a `double`, or a `long`.

And finally we have `BiConsumer`, which is similar, except that its `accept()` method takes two objects (types `T` and `U`, meaning the two objects don't have to be of the same type). So the `BiConsumer` interface looks like this:

```
@FunctionalInterface  
public interface BiConsumer<T, U>
```

with one function method, `accept()`:

```
void accept(T t, U u)
```

Here's a good use for a `BiConsumer`:

```
Map<String, String> env = System.getenv();  
BiConsumer<String, String> printEnv = (key, value) -> {  
    System.out.println(key + ": " + value);  
};  
printEnv.accept("USER", env.get("USER"));
```

The `printEnv` `BiConsumer` is a lambda expression with two arguments that we are using to display a key and value from a `Map`. To use the `printEnv` consumer, we call its `accept()` method, passing in two strings, and see the result displayed in the console.

**ForEach** Now is a good time to talk about the `forEach()` method that's been added to the Java 8 `Iterable` interface. Why? Because `forEach()` expects a

consumer. You'll end up using `forEach()` a lot, as it's a handy way to iterate through collections.

Here's how you use `forEach()` with a consumer to iterate through a `List` and display each item in the list:

```
List<String> dogNames = Arrays.asList("boi", "clover", "zooey");
Consumer<String> printName = name -> System.out.println(name);
dogNames.forEach(printName);    // pass the printName consumer to
                                // forEach()
```

We could, of course, combine the last two of these lines, like this:

```
dogNames.forEach(name -> System.out.println(name));
```

This consumer takes a `String` and just prints it.

The kind of consumer `forEach()` expects depends on the type of collection you're using. For example, when you iterate through `List`, you're accessing one object at a time, so the consumer you'll use is `Consumer` (that is, a consumer whose `accept()` method expects one argument, an object). And for `Map`, the consumer you'll use is a `BiConsumer` (that is, a consumer whose `accept()` method expects two arguments, both objects). Let's use the `BiConsumer` we created earlier, `printEnv`, with the `Map`'s `forEach()` method to display every key/value pair in the map:

```
Map<String, String> env = System.getenv();
BiConsumer<String, String> printEnv = (key, value) -> {
    System.out.println(key + ": " + value);
};

env.forEach(printEnv);    // the forEach() method of Map expects a BiConsumer
```

This displays every key/value pair in the `Map` that you get from `System.getenv()`.

**Side Effects from Within Lambdas** You already know that there are restrictions on modifying the value of variables from within lambda expressions. For instance, if you define a variable in the enclosing scope of a lambda

expression, you can't modify that variable from within the lambda:

```
String username;
BiConsumer<String, String> findUsername = (key, value) -> {
    if (key.equals("USER")) username = value; // compile error!
                                                // username must be
                                                // effectively final
};
env.forEach(findUsername);
```

This code will not compile because we're trying to change the value of `username` from within the lambda expression. Remember that variables declared outside a lambda expression must be final or effectively final to be used within a lambda expression.

However, we can cheat. Although we can't modify a variable from within a lambda expression, we *can* modify a field of an object. If we want to use `forEach()` to iterate through a collection of objects to, say, find a value, we can't return the value we find (because `forEach()` takes a consumer and the `accept()` method of a consumer is void), but we can change the field of an object from within the lambda to do effectively the same thing:

```

public class Consumers {
    public static void main(String[] args) {
        Map<String, String> env = System.getenv();
        User user = new User();
        BiConsumer<String, String> findUsername = (key, value) -> {
            if (key.equals("USER")) user.setUsername(value);
        };
        env.forEach(findUsername);
        System.out.println("Username from env: " + user.getUsername());
    }
}
class User {
    String username;
    public void setUsername(String username) {
        this.username = username;
    }
    public String getUsername() {
        return this.username;
    }
}

```

Note that this code will be less efficient than using a `for` loop iteration and breaking out of the iteration once the value is found. You might not ever need to do this, but now you know you can, just in case. In the next chapter on Streams, you'll learn a better way to extract and collect values that doesn't rely on side effects, as we're doing here.

**andThen...** andThen the murders began... Oh wait, no. This isn't a crime thriller; it's a Java book.

`andThen()` is actually a default method of the `Consumer` interface that you can use to chain consumers together. Let's imagine you have a `Dog` that can bark:

```
public class Dog {  
    private String name;  
    private int age;  
    private int weight;  
  
    public Dog(String name, int weight, int age) {  
        this.name = name;  
        this.weight = weight;  
        this.age = age;  
    }  
    // getters and setters here...  
    public String toString() { return this.name; }  
    public void bark() { System.out.println("Woof!"); }  
}
```

Now let's make some dogs and display them with `forEach()`:

```
public class ConsumerDogs {  
    public static void main(String[] args) {  
        List<Dog> dogs = new ArrayList<>();  
        Dog boi = new Dog("boi", 30, 6);  
        Dog clover = new Dog("clover", 35, 12);  
        Dog zooey = new Dog("zooey", 45, 8);  
        dogs.add(boi); dogs.add(clover); dogs.add(zooey);  
  
        Consumer<Dog> displayName = d -> System.out.print(d + " ");  
        dogs.forEach(displayName);      // line 10  
    }  
}
```

When you run this, you'll see

**boi clover zooey**

Now let's say we want to display the dog's name *andThen* we want to have the dog bark. Can we do it with consumers? Yes! Here's how:

```
dogs.forEach(displayName.andThen(d -> d.bark()));
```

Replace line 10 with this line and now you'll see

**boi Woof!**  
**clover Woof!**  
**zooey Woof!**

So how does this work? We're passing a “composed consumer” to the `forEach()` method of the `dogs` `ArrayList`. Let's step through this.

First, note that the `forEach()` method is calling the `accept()` method of the `Consumer` you're passing in behind the scenes. So for each dog `d` in the list, `forEach()` is essentially doing this:

## displayName.accept(d)

When the `andThen()` method of the `Consumer` is called, it says, okay, now use that same dog object, `d`, that we just used in the first consumer for the `accept()` method of the second consumer. Note that we've written the second consumer as an inline lambda (rather than as a separate declaration like we did for `displayName`)—but, of course, it works the same way. So the dog `d` whose name we just displayed is used in the second consumer, and we call that dog's `bark()` method, which simply displays `woof!` in the console. And so the result we see is the dog's name followed by `woof!` for each of the dogs in the list.

You might think that you could write both lambdas inline, like this:

```
dogs.forEach((d -> System.out.print(d + " ")).andThen(d -> d.bark()));
```

But you can't. You'll get a compile error. You can, however, use named consumers for both:

```
Consumer<Dog> displayName = d -> System.out.print(d + " ");
Consumer<Dog> doBark = d -> d.bark();
dogs.forEach(displayName.andThen(doBark));
```

Most (but not all!) of the consumers in `java.util.function` have an `andThen()` method, and notice that the type of the consumer you pass to the `andThen()` method must match the type of the consumer used as the first operation. So you can chain a `Consumer` with a `Consumer` and a `BiConsumer` with a `BiConsumer`, but not a `Consumer` with a `BiConsumer`.

## Working with Predicates

Remember earlier in this chapter we created a `DogQuerier` interface and used that interface to create an inner class and then a lambda expression.

Our interface looked like this:

```
interface DogQuerier {
    public boolean test(Dog d);
}
```

This interface is a functional interface, with a functional method `test()`, so

although we could create an instance using an inner class:

```
DogQuerier dq = new DogQuerier() {  
    public boolean test(Dog d) { return d.getAge() > 9; }  
};
```

we realized we could create an instance much more concisely using a lambda expression like this:

```
DogQuerier dq = d -> d.getAge() > 9; // replaces inner class!
```

Well, take a look at `java.util.function.Predicate`, and the interface might look familiar:

```
@FunctionalInterface  
public interface Predicate<T> {  
    boolean test(T t);  
    // default and static methods here.  
}
```

`Predicate` is a functional interface, with a functional method, `test()`, which returns a `boolean`, just like our `DogQuerier`. That means we can use the built-in `Predicate` interface in place of the `DogQuerier` interface (and get rid of the `DogQuerier` interface definition completely). Let's do that:

```
Dog boi = new Dog("boi", 30, 6);  
Dog clover = new Dog("clover", 35, 12);
```

```
Predicate<Dog> p = d -> d.getAge() > 9;  
System.out.println("Is Boi older than 9? " + p.test(boi));  
System.out.println("Is Clover older than 9? " + p.test(clover));
```

Boi is 6, and Clover is 12, so we see `false` and `true` for the results, just like we did before with `DogQuerier`.

Notice that one thing we must do to use `Predicate` in place of `DogQuerier` is add a type parameter to `Predicate`. Whereas we created `DogQuerier` to be specific to dogs, `Predicate` is a generic functional interface, so we have to provide a bit more information. The `T` argument defined in the `test()` method means we can pass any object to `Predicate`, so by adding a type parameter to `Predicate`, the predicate will know what type of argument to expect.

Let's expand the example just a bit so we can experiment more with predicates.

```

public class Dog {
    private String name;
    private int age;
    private int weight;

    public Dog(String name, int weight, int age) {
        this.name = name;
        this.weight = weight;
        this.age = age;
    }
    // getters and setters here
    // add a better description of the dog:
    public String toString() {
        return this.name + " is " + this.age + " years old and weighs " +
this.weight + " pounds";
    }
    public void bark() { System.out.println("Woof!"); }
}

public class TestDogPredicates {
    public static void main(String[] args) {
        ArrayList<Dog> dogs = new ArrayList<>();
        Dog boi = new Dog("boi", 30, 6);
        Dog clover = new Dog("clover", 35, 12);
        Dog aiko = new Dog("aiko", 50, 10);
        Dog zooley = new Dog("zooley", 45, 8);
        Dog charis = new Dog("charis", 120, 7);
        dogs.add(boi); dogs.add(clover); dogs.add(aiko);
        dogs.add(zooley); dogs.add(charis);

        System.out.println("--- All dogs ---");
        dogs.forEach(d -> System.out.println(d));

        System.out.println("--- Dogs younger than 9 ---");
        printDogIf(dogs, (d) -> d.getAge() < 9);

        System.out.println("--- Dogs 9 or older ---");
        printDogIf(dogs, (d) -> d.getAge() >= 9);
    }

    public static void printDogIf(ArrayList<Dog> dogs, Predicate<Dog> p) {
        for (Dog d : dogs) {
            if (p.test(d)) {
                System.out.println(d);
            }
        }
    }
}

```

What we've done here is create a method `printDogIf()` that takes a list of dogs and a `Predicate`, and tests each dog in the list against the predicate to see if the dog should be displayed. We can then use this method to display all the dogs younger than 9 with one predicate and display all the dogs 9 or older with another.

Running this code, we get the following output:

```
--- All dogs ---
boi is 6 years old and weighs 30 pounds
clover is 12 years old and weighs 35 pounds
aiko is 10 years old and weighs 50 pounds
zooey is 8 years old and weighs 45 pounds
charis is 7 years old and weighs 120 pounds
--- Dogs younger than 9 ---
boi is 6 years old and weighs 30 pounds
zooey is 8 years old and weighs 45 pounds
charis is 7 years old and weighs 120 pounds
--- Dogs 9 or older ---
clover is 12 years old and weighs 35 pounds
aiko is 10 years old and weighs 50 pounds
```

Now let's check out how `Predicates` are used in the JDK. One example is the `removeIf()` method of `ArrayList`, which takes a `Predicate` and removes an item from the `ArrayList` if the predicate's `test()` method returns true for that item. So, we can remove all dogs whose names begin with "c" (sorry Charis and Clover!) like this:

```
Predicate<Dog> findCs = d -> d.getName().startsWith("c");
dogs.removeIf(findCs);
System.out.println("---- After removing dogs whose names begin with c ---");
dogs.forEach(d -> System.out.println(d));
```

And see that, indeed, Charis and Clover have been removed from the output:

```
---- After removing dogs whose names begin with c ---  
boi is 6 years old and weighs 30 pounds  
aiko is 10 years old and weighs 50 pounds  
zooey is 8 years old and weighs 45 pounds
```

**Predicate's Default and Static Methods** If you look at the `Predicate` interface in the `java.util.function` package, you'll probably notice that `Predicate` has a few default methods and one static method. The default methods, `and()`, `or()`, and `negate()`, are there so you can chain predicates together, much like we did with consumers and the `andThen()` method. This can save you time creating new predicates that are logical combinations of predicates you already have. So, for instance, if we have a `Predicate` that tests to see if a dog's age is 6, we can easily test for a dog not being age 6 with the `negate()` method:

```
Predicate<Dog> age = d -> d.getAge() == 6;  
System.out.println("Is boi NOT 6? " + age.negate().test(boi));
```

And we get the result false because Boi is, indeed, 6 years old:

```
Is boi NOT 6? false
```

The `or()` and `and()` methods both take other predicates, so to chain them together you need two predicates. Let's create predicates to see if a dog's name is "boi" and the dog's age is 6 and a third predicate that chains the first two together with `and()`:

```
Predicate<Dog> name = d -> d.getName().equals("boi");  
Predicate<Dog> age = d -> d.getAge() == 6;  
Predicate<Dog> nameAndAge = d -> name.and(age).test(d);  
System.out.println("---- Test name and age of boi ----");  
System.out.println("Is boi named 'boi' and age 6? " + nameAndAge.test(boi));  
boi.setAge(7);  
System.out.println("Is boi named 'boi' and age 6? " + nameAndAge.test(boi));
```

First, we test to see if Boi is named “boi” and is age 6, then we set Boi’s age to 7, and test again. We get the output:

```
--- Test name and age of boi ---
Is boi named 'boi' and age 6? true
Is boi named 'boi' and age 6? false
```

We can simplify the nameAndAge Predicate even further by writing

```
Predicate<Dog> nameAndAge = name.and(age);
```

This works! Remember, what this does is create a new Predicate that is the composition of two Predicates, name and age. So the result of calling the and() method on the name predicate with the argument age is a new Predicate<Dog> that ands the result of calling name.test() on a dog and age.test() on that same dog.

The syntax for chaining can take a bit getting used to. Try writing a few of your own predicates, combining them with and(), or(), and negate(), to get the hang of it.

The static method in Predicate, isEqual(), just gives you a way to test if one object equals another, using the same test as equals() uses when comparing two objects (that is, are they the same object?).

```
Predicate<Dog> p = Predicate.isEqual(zooey);
System.out.println("Is aiko the same object as zooey? " + p.test(aiko));
System.out.println("Is zooey the same object as zooey? " + p.test(zooey));
```

One thing to note about isEqual() is this method is defined only on the predicates that take objects as arguments.

Along with Predicate, you’ll find BiPredicate, DoublePredicate, IntPredicate, and LongPredicate in java.util.function. You can probably guess what these do. Yep, BiPredicate’s test() method takes two arguments, whereas DoublePredicate, IntPredicate, and LongPredicate each take one argument of a primitive type (to avoid autoboxing).

Here’s a quick IntPredicate to demonstrate:

```
IntPredicate universeAnswer = i -> i == 42;  
System.out.println("Is the answer 42? " + universeAnswer.test(42));
```

`IntPredicate` is better than `Predicate<Integer>` for this example because the argument `i` doesn't have to be converted from `Integer` to `int` before it's tested. It's a similar idea with `DoublePredicate` and `LongPredicate`.

**BiPredicate** `BiPredicate` is just a variation on `Predicate` that allows you to pass in two objects for testing instead of one. Let's say we have an `ArrayList` of books and we want to create a set of predicates that will determine if we should buy a book based on its name, its price, or its name *and* price together. Here's how we might do that with `BiPredicate`:

```
List<Book> books = new ArrayList<>();  
// fill the books list with books here...  
BiPredicate<String, Double> javaBuy = (name, price) -> name.contains("Java");  
BiPredicate<String, Double> priceBuy = (name, price) -> price < 55.00;  
BiPredicate<String, Double> definitelyBuy = javaBuy.and(priceBuy);  
books.forEach(book -> {  
    if (definitelyBuy.test(book.getName(), book.getPrice())) {  
        System.out.println("You should definitely buy " + book.getName()  
            + "(" + book.getPrice() + ")");  
    }  
});
```

If we load up our books `ArrayList` with the following books:

```
"Your Brain is Better with Java", 58.99  
"OCP8 Java Certification Study Guide", 53.39  
"Is Java Coffee or Programming?", 39.86  
"While you were out Java happened", 12.99
```

then what we'll see in the output when we run this code is

You should definitely buy "OCP8 Java Certification Study Guide" (53.39)

You should definitely buy "Is Java Coffee or Programming?" (39.86)

You should definitely buy "While you were out Java happened" (12.99)

These are the books with Java in the title and a price less than \$55.00.

**Caveat Time** Of course, you don't *need* BiPredicate lambda expressions to write this code; you could just write the test (does the name contain "Java" and is the price less than 55.00) in the forEach lambda expression. Likewise, we haven't exactly needed many of the lambda expressions we've written in the simple examples throughout this chapter so far.

The main reason to use lambda expressions is if you'll end up using them in other ways too, and if having the code packaged up into lambdas will help make your code more concise and get you some code reuse (same reasons why you might use an inner class or even an external class). And, of course, if a method of a Java class, like the Logger log() example, requires a functional interface, like a Supplier, then that's a great reason to use a lambda expression (although, again, you don't *have* to—they are a convenience).

A big caveat here is that although we're showing lots of examples of building lambda expressions using the built-in functional interfaces so you get practice for the exam, once you're back in the real world, think about whether you really need a lambda expression before you write one.

## Working with Functions

We saved the most abstract of the functional interfaces from `java.util.function` for last. Aren't you lucky? Of course, by now, we hope you're feeling pretty solid about using functional interfaces.

The purpose of the `apply()` functional method in the `Function` interface is to take one value and turn it into another. The two values don't have to be the same type, so in the `Function` interface definition, we have two different type parameters, `T` (the type of the argument to `apply()`) and `R` (the type of the return value):

```
@FunctionalInterface  
public interface Function<T, R> {  
    R apply(T t);  
}
```

We can create an instance of `Function` to turn an `Integer` into a `String`:

```
Function<Integer, String> answer = a -> {  
    if (a == 42) return "forty-two";  
    else return "No answer for you!";  
};  
System.out.println(answer.apply(42));  
System.out.println(answer.apply(64));
```

If you run this code, you'll see the output:

```
forty-two  
No answer for you!
```

A `BiFunction` is similar except the `apply()` method takes two arguments and returns a value:

```
BiFunction<String, String, String> firstLast =  
    (first, last) -> first + " " + last;  
System.out.println("First and Last name: " + firstLast.apply("Joe", "Smith"));
```

In this example, the `BiFunction` `apply()` method takes two `Strings` and returns a `String`, but you could pass two arguments of different types and return a value of a third type. What we see, of course, is

```
First and Last name: Joe Smith
```

**Functions in the JDK** Let's use both a `Function` and a `BiFunction` in an example using the `Map` methods `computeIfAbsent()` and `replaceAll()`. These

are two examples from the JDK where you'll find `Function` and `BiFunction` used.

```
public class Functions {  
    public static void main(String[] args) {  
        Map<String, String> aprilWinner = new TreeMap<>();  
        aprilWinner.put("April 2017", "Bob");  
        aprilWinner.put("April 2016", "Annette");  
        aprilWinner.put("April 2015", "Lamar");  
  
        System.out.println("--- List, before checking April 2014 ---");  
        aprilWinner.forEach((k, v) -> System.out.println(k + ": " + v));  
  
        // no key for April 2014, so John Doe gets added to the map  
        aprilWinner.computeIfAbsent("April 2014", (k) -> "John Doe");  
  
        // key April 2014 now has a value, so Jane won't be added  
        aprilWinner.computeIfAbsent("April 2014", (k) -> "Jane Doe");  
  
        System.out.println("--- List, after checking April 2014 ---");  
        aprilWinner.forEach((k, v) -> System.out.println(k + ": " + v));  
  
        // use a BiFunction to replace all values in the map with  
        // uppercase values  
        aprilWinner.replaceAll((key, oldValue) -> oldValue.toUpperCase());  
        System.out.println("--- List, after replacing values with  
        uppercase ---");
```

```
    aprilWinner.forEach((k, v) -> System.out.println(k + ": " + v));  
}  
}
```

The output is as follows:

```
--- List, before checking April 2014 ---  
April 2015: Lamar  
April 2016: Annette  
April 2017: Bob  
--- List, after checking April 2014 ---  
April 2014: John Doe  
April 2015: Lamar  
April 2016: Annette  
April 2017: Bob  
--- List, after replacing values with uppercase ---  
April 2014: JOHN DOE  
April 2015: LAMAR  
April 2016: ANNETTE  
April 2017: BOB
```

We first use `computeIfAbsent()` to add a key and value to our Map of April winners if that key/value pair doesn't yet exist in the Map.

`computeIfAbsent()` takes a key and a Function. The Function provides a value to store in the Map for the key if a value for that key doesn't yet exist. The argument in the Function lambda expression is the key, so you could create a value based on the key, but in this example, we're keeping it super simple and just returning a String.

Then we use the `replaceAll()` method to replace every value in the Map with the uppercase version of the old value. So where we have stored Bob, we'll now

be storing BOB.

`replaceAll()` takes a `BiFunction`. The lambda expression we pass to `replaceAll()` has two arguments, a key, and the current value in the `Map`, and returns a new value to store in the `Map` for that key. In this example, we're just returning the previous value in uppercase.

**More Functions** `Function` has a couple of default methods and a static method in addition to its functional method, `apply()`: `andThen()`, `compose()`, and `identity()`. `andThen()` is similar to the `Consumer`'s `andThen()` method, applying Functions in sequence. `compose()` is the same except it applies the Functions in reverse order.

And the static method in `Function` is `identity()`, which just returns its input argument:

```
Function<Integer, Integer> id = Function.identity();
System.out.println(id.apply(42));
```

Answer: 42. (Of course.)

What on earth is an identity function used for? Imagine a scenario where you have defined a method that takes a `Function` as an argument that changes a value in a data structure. But in some cases, you don't want that value to change. In those cases, pass the `identity` `Function` as an easy "do nothing" operation.

Along with `Function` and `BiFunction`, as you might expect, you'll also find `DoubleFunction` (`apply()` takes a `double` as an argument and returns an object), `IntFunction` (`apply()` takes an `int` as an argument and returns an object), and `LongFunction` (`apply()` takes a `long` as an argument and returns an object).

Ah, but wait, there's more! We also have `DoubleToIntFunction`, `DoubleToLongFunction`, `IntToDoubleFunction`, `IntToLongFunction`, `LongToDoubleFunction`, `LongToIntFunction`, `ToDoubleFunction`, `ToIntFunction`, `ToLongFunction`, `ToDoubleBiFunction`, `ToIntBiFunction`, and last but definitely not least, `ToLongBiFunction`.

Oh, my goodness—the variations the JDK authors thought of when creating the functional interfaces. (It's a wonder they didn't think of 100 more.) We are pretty sure you can make a good guess at what these do. The main trick with these is that the functional method is not `apply()`; it's a slight variation on `apply()`, so just keep that in the back of your mind.

For instance, how about `IntToDoubleFunction`? The functional method is `applyAsDouble()`, and yes, it takes an `int` and returns a `double`, avoiding all that

inefficient autoboxing. How about `ToIntFunction`? The `applyAsInt()` method takes an object and returns an `int`.

Okay, we are sure you get the hang of it at this point, and we'll leave you to the Java docs to find out more if you're interested. For the exam, focus on `Function` and `BiFunction`, but be aware that these other variations exist and remember that there are variations in the functional method names that go with them.

## Working with Operators

Finally (yes, really!), we have the operator variations on the functional interfaces. All the operators are, in fact, slightly modified versions of other functional interfaces. Let's pick the one operator you should be familiar with for the exam, `UnaryOperator`, to look at, and you can explore the rest on your own.

`UnaryOperator` extends the `Function` interface, so its functional method is also `apply()`. However, unlike `Function`, it requires that the type of the argument to `apply()` be the same as the type of the return value, so `UnaryOperator` is defined like this:

```
@FunctionalInterface  
public interface UnaryOperator<T>  
extends Function<T, T>
```

The `T, T` in the `Function` type parameters is what tips you off that the type of the argument and return value must be the same. That's why you only have to specify one type parameter for `UnaryOperator`:

```
UnaryOperator<Double> log2 = v -> Math.log(v) / Math.log(2);  
System.out.println(log2.apply(8.0));
```

In this example, we're defining a `log2` `UnaryOperator` that computes the log base 2 of a value. Log base 2 of 8.0 is 3.0 because  $2 * 2 * 2$  (3 times) is 8.

Notice that you could, of course, use `Function<Double, Double>` instead... that's essentially the same thing. `UnaryOperator` saves a little typing and makes it a bit clearer that you're defining an operator that takes a value that is the same type as the return value...but that's about it. It's just a slightly restricted version of `Function`. And the same applies to the other operators in the package.

Time to pat yourself on the back, take a break, and eat some cookies, because

you made it through all of the functional interfaces in `java.util.function`. As long as you have a good sense of how to use them to create lambda expressions and how to use the core interfaces that we've covered here, you'll be in solid shape.

## CERTIFICATION OBJECTIVE

### Method References (OCP Objective 3.8)

#### 3.8 Use method references with Streams.

You already know that a lambda expression is a shorthand way of writing an instance of a class that implements a functional interface. Believe it or not, there are a few circumstances when you can make your code even *more* concise by writing a shorthand for the lambda expression (yes, it's a shorthand for the shorthand).

Sometimes, the only thing a lambda expression does is call another method, for instance:

```
List<String> trees = Arrays.asList("fir", "cedar", "pine");
trees.forEach(t -> System.out.println(t));
```

Here we're using a lambda expression to take a tree name, `t`, and pass it to `System.out.println()`. This code is already pretty short; can we shorten it even more? Yes! Apparently, the Java 8 authors like finding ways to avoid typing, so they invented the “method reference”:

```
List<String> trees = Arrays.asList("fir", "cedar", "pine");
trees.forEach(System.out::println);
```

This method reference is a shorthand way of writing the lambda expression. But wait, where did the argument `t` go and don't we need that? We know that `forEach()` takes a `Consumer`, and we know what it's consuming is tree names, which are `Strings`. And we know that `System.out.println()` takes a `String`. A lot can be inferred from this shorthand for the lambda expression. Here what's inferred is that we want to call the `println()` method of `System.out`, passing in

the `String` object that we have at hand via the `forEach()`. Of course, keep in mind, we can't do anything fancy with how we print the tree name because we're not specifying anything but the method to call on the tree name argument that the consumer (the method reference) is getting behind the scenes.

## Kinds of Method References

The `System.out::println` method reference is an example of a “method reference,” meaning we’re calling a method, `println()`, of `System.out`. You can create method references for your own methods, too:

```
public class MethodRefs {  
    public static void main(String[] args) {  
        List<String> trees = Arrays.asList("fir", "cedar", "pine");  
  
        trees.forEach(t -> System.out.println(t)); // print with lambda  
        trees.forEach(System.out::println); // print with a  
                                         // method reference  
  
        trees.forEach(MethodRefs::printTreeStatic); // print with our own  
                                         // static method reference  
    }  
  
    public static void printTreeStatic(String t) {  
        System.out.println("Tree name: " + t);  
    }  
}
```

For this code, we’ll see this output:

```
fir
cedar
pine
fir
cedar
pine
Tree name: fir
Tree name: cedar
Tree name: pine
```

First, the tree names are printed using a lambda expression; then the tree names are printed using an *instance method reference* to `System.out.println()` and finally a longer string using a *static method reference* to our own static method, `printTreeStatic()`. A “static method reference” is a method reference to a static method. The method reference to `System.out.println()` is an “instance method reference”—that is, a reference to the `println()` instance method of `System.out`.

Writing `::` instead of `.` in the method reference takes a little getting used to; you’ll find yourself typing a `.` instead of a `::` and then going back to fix it (at least we do!). What the method reference does here is make a `Consumer` that calls the method on the implicit argument that’s getting passed to that lambda behind the scenes.

In addition to instance and static method references, there are a couple of other types of method references: method references for arbitrary objects and constructor method references. We’ll see some more examples of method references in the next chapter when we talk about streams.

## Write Your Own Functional Interface

Guess what, you already wrote your own functional interface. Remember `DogQuerier`? Yep, that’s the one:

```
@FunctionalInterface  
interface DogQuerier {  
    public boolean test(Dog d);  
}
```

Of course, `DogQuerier` is all about dogs. You then saw that `DogQuerier` is really just a dog version of `Predicate`, which works on any type of object.

You might want to write a functional interface that works on any type of object, too. Imagine, for instance, that you want an interface with a `test()` method that takes three objects and returns a boolean—a `TriPredicate` if you will. There is no `TriPredicate` in `java.util.function` (only `Predicate` and `BiPredicate`), so how about writing your own?

```
@FunctionalInterface  
interface TriPredicate<T, U, V> {  
    boolean test(T t, U u, V v);  
}
```

We're specifying an interface with three type parameters: `T`, `U`, and `V`. The `test()` method takes three types of objects to test, and they can all be of differing types or all the same. Now let's write a lambda expression of this type and test it out:

```

public void triPredicate() {
    TriPredicate<String, Integer, Integer> theTest =
        (s, n, w) -> {
            if (s.equals("There is no spoon") && n > 2 && w < n) {
                return true;
            } else {
                return false;
            }
        };
    System.out.println("Pass the test? " +
        theTest.test("Follow the White Rabbit", 2, 3));
    System.out.println("Pass the test? " +
        theTest.test("There is no spoon", 101, 3));
}

```

We see this output:

```

Pass the test? false
Pass the test? true

```

The trick to writing your own generic functional interfaces is having a good handle on generics. As long as you understand what a functional interface is and how to use generics to specify parameter and return types, you're good to go.

## Functional Interface Overview

As we've said, the `java.util.function` package has 43 different functional interfaces. You need to make sure you are familiar with the core interfaces: `Supplier`, `Consumer`, `Predicate`, and `Function`. In addition, you should understand the variations on these core interfaces—the primitive interfaces designed to avoid autoboxing, and the `Bi-`versions that allow you to pass two parameters to the functional methods rather than one. You don't need to

memorize all 43 interfaces, but you need to understand the patterns of the variations and their functional method names. We've listed the core interfaces, plus several variations, and their functional methods in [Table 8-1](#). In addition to the functional methods, you should be familiar with the default and static methods in the interfaces, such as `andThen()`, `and()`, `negate()`, `or()`, and so on. We don't list those in [Table 8-1](#), so look at the online documentation for details beyond what we've covered in this chapter.

**TABLE 8-1** Functional Interfaces Covered on the Exam

Interface	Functional Method	Description
<code>Supplier&lt;T&gt;</code>	<code>T get()</code>	Suppliers supply results. The Supplier's functional method <code>get()</code> takes no arguments and supplies a generic object result ( <code>T</code> ).
<code>Consumer&lt;T&gt;</code>	<code>void accept(T t)</code>	Consumers consume values. The Consumer's functional method <code>accept()</code> takes a generic object argument ( <code>T</code> ) and returns no value.
<code>Predicate&lt;T&gt;</code>	<code>boolean test(T t)</code>	Predicates test things (and do logical operations). The Predicate's functional method <code>test()</code> takes a generic object argument ( <code>T</code> ) and returns a boolean.
<code>Function&lt;T,R&gt;</code>	<code>R apply(T t)</code>	Functions are generic functional interfaces. The functional method <code>apply()</code> takes a generic object argument ( <code>T</code> ) and returns a generic object ( <code>R</code> ).
<code>IntSupplier</code>	<code>int getAsInt()</code>	Supplies int values (to avoid autoboxing). Note the difference in the functional method name ( <code>getAsInt()</code> rather than <code>get()</code> , as for <code>Supplier</code> ). This pattern is used across the various primitive versions of the functional interfaces.
<code>IntConsumer</code>	<code>void accept(int value)</code>	Consumes int values.
<code>IntPredicate</code>	<code>boolean test(int value)</code>	Tests int values.
<code>IntFunction&lt;R&gt;</code>	<code>R apply(int value)</code>	The IntFunction's functional method <code>apply()</code> takes an int argument and returns a generic object result ( <code>R</code> ).
<code>BiConsumer&lt;T,U&gt;</code>	<code>void accept(T t, U u)</code>	Consumes two values.
<code>BiPredicate&lt;T,U&gt;</code>	<code>boolean test(T t, U u)</code>	The BiPredicate's functional method <code>test()</code> takes two generic arguments ( <code>T, U</code> ).
<code>BiFunction&lt;T,U,R&gt;</code>	<code>R apply(T t, U u)</code>	The BiFunction's functional method <code>apply()</code> takes two generic arguments ( <code>T, U</code> ) and returns a generic object result ( <code>R</code> ).
<code>UnaryOperator&lt;T&gt;</code>	<code>T apply(T t)</code>	The UnaryOperator's functional method <code>apply()</code> takes one generic operator ( <code>T</code> ) and returns a value of the same type ( <code>T</code> ).

Note: You should review all the functional interfaces in `java.util.function` for variations on the functional interfaces covered in [Table 8-1](#) and for details of the default and static methods of interfaces.



*You know that functional interfaces are just interfaces with one abstract method—the functional method. The interfaces in [Table 8-1](#) have been added to the JDK to make it easier for you to write code, but as you know, there's nothing particularly special about functional interfaces beyond their use in the JDK (as we saw with `Logger`).*

*There are a few functional interfaces already in the JDK, not listed in `java.util.function`. You've already seen one of these—`Comparator`—with its functional method, `compare()`. Another is `Comparable`, which is implemented by various types (like `String`, `LocalDateTime`, and so on) for sorting.*

*A third example is `Runnable`, which has a functional method, `run()`. So where you might have implemented a `Runnable` as an inner class before:*

```
Runnable r = new Runnable() {
    @Override
    public void run() {
        System.out.println("Do this");
    }
};
```

*you can now use a lambda expression instead:*

```
Runnable r = () -> System.out.println("Do this");
```

## CERTIFICATION SUMMARY

---

---

We began this chapter by looking at what lambda expressions are, how to write lambda expressions, and when to write them. Lambdas are syntax shorthands for writing a class to implement a functional interface and then instantiating that class. With lambdas, you focus on the method that's implementing the abstract method of the interface (remember, there's only one, because the functional interfaces have only one abstract method) and eliminate the other syntax, so you end up with more concise code.

The lambda syntax can take a little getting used to, but there aren't too many rules to remember: you use an `->` symbol with parameters on the left and a method body on the right. One trick is that if a lambda expression is just one expression, you can even eliminate the `{` and `}` that usually go with a method body, like this:

`( ) -> 42`

We talked about how you can think of lambdas as chunks of code that you pass around. We can pass an object to a method, and we can pass a lambda expression to a method. Remembering that a lambda represents an instance of a class that implements an interface can help you see how this works.

When lambdas refer to variables from their enclosing scope, those variables are “captured” and so you can refer to them when applying a lambda later. The important key here, however, is that the variables must be final or effectively final.

Then we delved into functional interfaces. A functional interface is simply an interface that has one abstract method—the functional method. Lambda expressions are related to functional interfaces because the type of a lambda expression is always a functional interface. Why? Because the method the lambda expression defines is the one and only functional method in that interface, so there is no confusion in the lambda expression syntax about which method you're writing.

The tricky part of functional interfaces is learning about the new interfaces defined in `java.util.function`. Forty-three new interfaces are there to help you shortcut the process of writing functional interfaces and also to provide a way for other JDK methods to accept objects of a functional interface type. The core functional interfaces are `Consumer`, `Supplier`, `Predicate`, and `Function`, and all the other functional interfaces in `java.util.function` are variations on these. We talked about a few examples in which new JDK methods, like the `Logger`'s `log()` method, can now take objects of one of the types in this package.

As we covered the various functional interfaces, we looked at the functional methods defined for each one, like the `Predicate`'s `test()` method and the `Supplier`'s `get()` method, and some of the static and default methods included in some of the functional interfaces, too, like the `Consumer`'s `andThen()` method.

We talked about operators, which are just slight variations on the other functional interfaces and have a restriction that the parameter type is the same as the return value type. So, if you define a `UnaryOperator` whose functional method takes an `Integer`, that operator method must return an `Integer` too.

A lambda expression is a syntax shorthand to make your code more concise, and a method reference is a shorthand that can make your code even more concise in certain situations, such as when you're just passing on an argument to another method. A common example for this is replacing a lambda expression that simply prints its argument:

```
Consumer<String> c = (s) -> System.out.println(s);
```

with a method reference:

```
Consumer<String> c = System.out::println;
```

Java can infer that you want to pass on the argument to the `Consumer` (which must have an argument; it's a `consumer!`) to the `System.out.println()` method. Again, this syntax takes some getting used to. You'll see it again in the next chapter and get more practice with it.



## TWO-MINUTE DRILL

Here are some of the key points from this chapter.

## Create and Use Lambda Expressions (OCP Objective 2.6)

- A lambda expression is a shorthand syntax for an instance of a class that implements a functional interface.
- Use `->` to define a lambda expression with the arguments on the left and the body on the right of the arrow.

- Typically, we leave off the types of a lambda's arguments because they can be inferred from the functional interface definition.
- If you have multiple parameters for a lambda, then you must surround them with parentheses.
- If you have no parameters for a lambda, then you must use empty parentheses, () .
- If you specify the type of a parameter of a lambda, you must use parentheses.
- If the body of a lambda expression has multiple statements, you must use curly braces.
- Lambda expressions are often used in place of an inner class.
- If the body of a lambda expression simply evaluates an expression and returns a value, you can leave off the return keyword.
- The type of the return value of a lambda expression is inferred from the functional interface definition.
- You can pass lambda expressions to methods, either by name or by writing them inline.
- Lambda expressions capture variables from the enclosing scope if they are used within the body of the lambda.
- All captured variables in a lambda expression must be final or effectively final.

## **Iterate Using forEach Methods of List (OCP Objective 3.5)**

- The forEach( ) method in collection types, like List, takes a Consumer and allows you to easily iterate through the collection. The Consumer's accept( ) method argument is the current object in the collection you are iterating over.

## **Use the Built-in Interfaces Included in the java.util.function Package such as Predicate, Consumer, Function, and Supplier (OCP Objective 4.1)**

- A functional interface is an interface with one abstract method.
- The single abstract method in a functional interface is called the functional method.
- Functional interfaces can include any number of default and static methods in addition to the functional method.
- Functional interfaces can redefine public methods from `Object`.
- Use `@FunctionalInterface` to annotate functional interfaces.
- If you add a second abstract method to a functional interface annotated with `@FunctionalInterface`, you will get a compiler error: “Invalid ‘`@FunctionalInterface`’ annotation; [interface name] is not a functional interface.“

## Core Functional Interfaces (OCP Objectives 4.1, 4.2, 4.3, 4.4)

- The JDK provides several built-in functional interfaces in `java.util.function`.
- All of these functional interfaces fall into one of four categories: suppliers, consumers, predicates, or functions.
- The basic functional interfaces from this package are `Supplier`, `Consumer`, `Predicate`, and `Function`.
  - The functional method of `Supplier` is `get()`. It returns a value.
  - The functional method of `Consumer` is `accept()`. It takes an argument and returns no value.
  - The functional method of `Predicate` is `test()`. It takes an argument and returns a boolean.
  - The functional method of `Function` is `apply()`. It takes an argument and returns a value.

## Using Functional Interfaces (OCP Objectives 4.1, 4.2, 4.3, 4.4)

- You can compose consumers with the `andThen()` default method.
- Use the same type of consumer when composing two consumers together.

- Perform logical tests with predicates using the default predicate methods `and()`, `or()`, and `negate()`.
- A `Predicate`'s static `isEqual()` method returns a `Predicate` that tests to see if two objects are equal. Note that this method is only available in the `Predicate` interface, not the predicate variations in `java.util.function`.
- Compose Functions together with the methods `andThen()` and `compose()`.
- Check carefully to see which functional interfaces support which default and static methods.
- The `forEach()` method in collection types, like `List`, takes a `Consumer` and allows you to easily iterate through the collection. The `Consumer`'s `accept()` method argument is the current object in the collection you are iterating over.
- The `replaceAll()` method in collection types, like `List`, takes a `UnaryOperator` and allows you to replace items in the `List` with different values, ones that could be based on the current values or completely new values.
- The built-in functional interfaces are conveniences so you don't have to create your own.
- The built-in functional interfaces are used in several ways in the Java 8 JDK, including with Streams (see [Chapter 9](#)).
- Creating your own functional interfaces is no different from creating any interface except you must make sure the interface has only one abstract method.
- Brush up on generics ([Chapter 6](#)) to make sure you know how to create a functional interface with generic types.

## Develop Code That Uses Primitive Versions of Functional Interfaces (OCP Objective 4.2)

- Some variations of functional interfaces in the `java.util.package` are meant to handle primitive values to avoid autoboxing.
- `IntSupplier`'s functional method, `getAsInt()`, takes no arguments and returns an `int`. This is to avoid autoboxing the result as `Integer`, in case you need a primitive.

- Likewise, DoubleSupplier's functional method, `getAsDouble()`, takes no arguments and returns a double.
- IntConsumer's functional method, `accept()`, takes an int and does not return any value.
- IntPredicate's functional method, `test()`, takes an int and returns a boolean.
- IntFunction's functional method, `apply()`, takes an int and returns an object value.
- The functional method name in functional interfaces is not always the same (e.g., `IntSupplier` uses `getAsInt()` rather than `get()`, and `IntToLongFunction` uses `applyAsLong()` rather than `apply()`), so note the patterns of naming conventions for functional methods.

## Develop Code That Uses Binary Versions of Functional Interfaces (OCP Objective 4.3)

- Some variations of functional interfaces in the `java.util` package are meant to allow multiple arguments.
- BiConsumer's functional method, `accept()`, takes two arguments and returns no value.
- BiPredicate's functional method, `test()`, takes two arguments and returns a boolean.
- BiFunction's functional method, `apply()`, takes two arguments and returns a value.
- The `forEach()` method in the `Map` collection type takes a `BiConsumer`, and the arguments of the `accept()` method are the key and value of the current `Map` entry.

## Develop Code That Uses the UnaryOperator Interface (OCP Objective 4.4)

- The operator functional interfaces in `java.util.function` are variations on the function interfaces (such as `Function`).
- Whereas `Function` takes a value of a type and returns a value of perhaps a different type, the `UnaryOperator` takes a value of a type and returns a

value of that same type.

## SELF TEST

The following questions will help you measure your understanding of the material in this chapter. If you don't get them all, go back and review and try again.

1. Which of these are functional interfaces? (Choose all that apply.)

A.

```
interface Question {  
    default int answer() {  
        return 42;  
    }  
}
```

B.

```
interface Tree {  
    void grow();  
}
```

C.

```
interface Book {  
    static void read() {  
        System.out.println("Turn the page...");  
    }  
}
```

D.

```
interface Flower {  
    boolean equals(Object f);  
    default void bloom() {  
        System.out.println("Petals are opening");  
    }  
    String pick();  
}
```

2. Given the following interface:

```
@FunctionalInterface  
interface FtoC {  
    double convert(double f);  
}
```

Which of the following expressions are legal? (Choose all that apply.)

- A. FtoC converter = (f) -> (f - 32.0) \* 5/9;
  - B. FtoC converter = f -> (f - 32.0) \* 5/9;
  - C. FtoC converter = f -> return ((f - 32.0) \* 5/9);
  - D. double converter = f -> (f - 32.0) \* 5/9;
  - E. FtoC converter = f -> { return (f - 32.0) \* 5/9; };
3. Which of the following compiles correctly?
- A. Predicate<String> p = (s) -> System.out.println(s);
  - B. Consumer<String> c = (s) -> System.out.println(s);
  - C. Supplier<String> s = (s) -> System.out.println(s);
  - D. Function<String> f = (s) -> System.out.println(s);
4. Given the code fragment:

```
System.out.format("Total = %.2f", computeTax(10.00, (p) -> p * 0.05));
```

Which method would you use for `computeTax()` so the code fragment prints `Total = 10.50`?

A.

```
double computeTax(double price, Function<Double> op) {  
    return op.apply(price) + price;  
}
```

B.

```
double computeTax(double price, UnaryOperator<Double> op) {  
    return op.apply(price) + price;  
}
```

C.

```
double computeTax(double price, double op) {  
    return op.apply(price) + price;  
}
```

D.

```
Function<Double, Double> computeTax(double price, UnaryOperator<Double> op) {  
    return op.apply(price) + price;  
}
```

5. Given:

```
class Reading {  
    int year;  
    int month;  
    int day;  
    double value;  
  
    Reading(int year, int month, int day, double value) {  
        this.year = year; this.month = month; this.day = day; this.value = value;  
    }  
}
```

and the code fragment:

```
List<Reading> readings = Arrays.asList(  
    new Reading(2017, 1, 1, 405.91),  
    new Reading(2017, 1, 8, 405.98),  
    new Reading(2017, 1, 15, 406.14),  
    new Reading(2017, 1, 22, 406.48),  
    new Reading(2017, 1, 29, 406.20),  
    new Reading(2017, 2, 5, 406.03));
```

Which code fragment will sort the readings in ascending order by value and print the value of each reading?

A.

```
readings.sort((r1, r2) -> r1.value < r2.value ? -1 : 1);  
readings.forEach(System.out.println(r.value));
```

B.

```
readings.sort((r1, r2) -> r1.value < r2.value ? 1 : -1);  
readings.forEach(System.out::println(r.value));
```

C.

```
readings.sort((r1, r2) -> r1.value < r2.value ? -1 : 1);  
readings.forEach(System.out::println);
```

D.

```
readings.sort((r1, r2) -> r1.value < r2.value ? -1 : 1);  
readings.forEach(r -> System.out.println(r.value));
```

6. Given the code fragments:

```
class Human {  
    public Integer age;  
    public String name;  
  
    public Human(Integer age, String name) {  
        this.age = age;  
        this.name = name;  
    }  
    public Integer getAge() { return this.age; }  
}
```

and

```
Supplier<Human> human = () -> new Human(34, "Joe");  
Human joe = human.XXXX;
```

Which code fragment inserted at xxxx will cause a new Human object to be stored in the variable joe?

- A. push()
- B. get()
- C. apply()
- D. test()
- E. accept()

7. Given:

```
class Human {  
    public Integer age;  
    public String name;  
  
    public Human(Integer age, String name) {  
        this.age = age;  
        this.name = name;  
    }  
    public Integer getAge() { return this.age; }  
}
```

and the code fragment:

```
Human jenny = new Human(18, "Jenny");  
Human jeff = new Human(17, "Jeff");  
Human jill = new Human(21, "Jill");  
List<Human> people = new ArrayList<>(Arrays.asList(jenny, jeff, jill));  
// L1  
people.forEach(printAdults);
```

Which code fragment inserted at line // L1 will print the names of only adults (those humans whose age is older than 17)?

A.

```
Predicate<Human> printAdults = p -> { if (p.getAge() >= 18) {  
    System.out.println(p.name);  
} };
```

B.

```
Predicate<Human> adult = p -> p.getAge() >= 18;  
Consumer<Human> printAdults = p -> { if (p.getAge(adult.test()) >= 18) {  
    System.out.println(p.name);  
} };
```

C.

```
Predicate<Human> adult = p -> p.getAge() >= 18;
Consumer<Human> printAdults = p -> { if (adult.test(p)) {
    System.out.println(p.name);
}};


```

D.

```
Consumer printAdults(Human p) {
    if (p.getAge() >= 18) {
        System.out.println(p.name);
    }
}


```

8. Given:

```
List<String> birds =
    Arrays.asList("eagle", "seagull", "albatross", "buzzard", "goose");
int longest = 0;
birds.forEach(b -> { // L3
    if (b.length() > longest) {
        longest = b.length(); // L5
    }
});
System.out.println("Longest bird name is length: " + longest);


```

What is the result?

- A. "Longest bird name is length: 9"
- B. Compilation fails because of an error on line L5
- C. Compilation fails because of an error on line L3
- D. A runtime exception occurs on line L5

9. Given the following code fragment:

```
Supplier<List<Double>> readingsSupplier =  
    Arrays.asList(405.91, 405.98, 406.14, 406.48, 406.20, 406.03);  
    for (Double r : readingsSupplier.get()) { System.out.print(r + " "); }
```

What is the result?

- A. 405.91
- B. 405.91 405.98 406.14 406.48 406.2 406.03
- C. An exception is thrown at runtime
- D. Compilation fails

10. Given the code fragment:

```
BiFunction<Integer, String, String> foo = (n, s) -> {  
    String newString = "";  
    for (int i = 0; i < n; i++) {  
        newString = s + " " + newString;  
    }  
    return newString;  
};  
Function<String, String> bar = (s) -> s + "bar";  
System.out.println(foo.andThen(bar).apply(3, "foo"));
```

What is the result?

- A. foo foo foo bar bar bar
- B. foo foo foo bar

- C. foo foo foo
- D. Compilation fails

**11.** Given the code fragment:

```
List<String> trees = Arrays.asList("FIR", "CEDAR", "PINE");
// L1
trees.replaceAll(convert);
trees.forEach(t -> System.out.print(t + " "));
```

Which fragment(s), inserted independently at // L1, produce the output?  
(Choose all that apply.)

fir cedar pine

- A. UnaryOperator<String> convert = (t) -> t.toLowerCase();
- B. UnaryOperator<String, String> convert = (t) -> t.toLowerCase();
- C. Function<String, String> convert = (t) -> t.toLowerCase();
- D. Supplier<String> convert = (t) -> t.toLowerCase();

**12.** Given the code fragment:

```
Map<String, String> todos = new HashMap<String, String>();
todos.put("monday", "wash dog");
todos.put("tuesday", "weed yard");
// L1
todos.forEach(printTodo);
```

Which fragment(s), inserted independently at // L1, produce the to-do items, both key and value, in the Map? (Choose all that apply.)

A.

```
Function<String, String> printTodo =  
    (String k, String v) -> System.out.println("On " + k + " do: " + v);
```

B.

```
Consumer<String, String> printTodo =  
    (String k, String v) -> System.out.println("On " + k + " do: " + v);
```

C.

```
Consumer<Map> printTodo =  
    (Map m) -> System.out.println("On " + m.keySet() + "do: " + m.values());
```

D.

```
BiConsumer<String, String> printTodo =  
    (String k, String v) -> System.out.println("On " + k + " do: " + v);
```

**13.** Given the code fragments:

```
class TodoList {  
    public void checkTodoDay(Map<String, String> todos, Predicate<String> isDay) {  
        todos.forEach((d, t) -> {  
            if (isDay.test(d)) {  
                System.out.println("You really should do this today! " + t);  
            }  
        });  
    }  
}
```

and

```
Map<String, String> todos = new HashMap<String, String>();  
todos.put("monday", "wash dog");  
todos.put("tuesday", "weed yard");  
TodoList todoList = new TodoList();  
// L1
```

Which fragment(s), inserted independently at // L1, display the output consisting of the to-do item for Tuesday? (Choose all that apply.)

A.

```
todoList.checkTodoDay(todos,  
    (k) -> if (k.equals("tuesday")) return true; else return false; );
```

B.

```
todoList.checkTodoDay(todos, (k, v) -> {  
    if (k.equals("tuesday")) {  
        System.out.println("You really should do this today! " + v);  
    }  
});
```

C.

```
todoList.checkTodoDay(todos, (k) -> k.equals("tuesday"));
```

D.

```
todoList.checkTodoDay(todos, (k) -> k.test("tuesday"));
```

**14.** Given the code fragment:

```
DoubleSupplier d = () -> 42.0;  
// L1
```

Which fragment(s), inserted independently at // L1, produce the output  
(Choose all that apply.)

The answer is: 42.0

A.

```
System.out.println("The answer is: " + d.get());
```

B.

```
double answer = d.getAsDouble();
System.out.println("The answer is: " + answer);
```

C.

```
System.out.println("The answer is: " + d.getAsDouble());
```

D.

```
double answer = d.get();
System.out.println("The answer is: " + answer);
```

## A SELF TEST ANSWERS

1.  **B** and **D** are correct. For **B**, we just have one abstract method, so this is a functional interface. Although **D** includes the `equals()` method (implying that classes implementing this interface must implement an `equals()` method), this method doesn't count because it's inherited from `Object`. The default method `bloom()` is default and so doesn't count; the method `pick()` is the functional method.

- A** is incorrect because the interface has no functional method (one abstract method). A default method is not a functional method. Likewise, **C** is incorrect because the interface has no functional method (one abstract method). A static method is not a functional method. (OCP Objective 4.1)
2.  **A, B, and E** are correct variations on writing lambda expressions.
- C** and **D** are incorrect. **C** is invalid syntax for lambda expressions, with a return statement that's not enclosed in { }. **D** is incorrect because it has the wrong type for the lambda expression. (OCP Objective 2.6)
3.  **B** is correct. The lambda expression is a consumer, so only the type Consumer is correct for this lambda expression—the functional method takes an argument and returns nothing.
- A, C, and D** are incorrect based on the above. (OCP Objective 4.1)
4.  **B** is correct. computeTax() is a method that takes two arguments, a double and a UnaryOperator, and returns a double value. We know that the lambda expression is a UnaryOperator because the functional method takes a Double and returns a Double (with autoboxing and autounboxing).
- A, C, and D** are incorrect. **A** could almost work because a UnaryOperator is a type of Function; however, a correct declaration of the Function would specify the type of both the argument and the return value, because unlike UnaryOperator, they are not necessarily the same type. (OCP Objective 4.4)
5.  **D** is correct. The sort() method of the List requires a Comparator, which can be expressed as a lambda expression implementing the compare() functional method. This method must return a -1 or 1 (for items that are not equal) depending on the ordering; because we want ascending order, we test if object 1 is less than object 2 and return -1 if so; otherwise, 1 is returned to get ascending order. Because we are comparing the values of the reading objects, we use the value field of each reading object to make the comparison. In the forEach, we supply a Consumer, whose functional method takes a reading object and prints the value field of that reading. **D** properly supplies a Consumer to forEach that prints just the values.
- A, B, and C** are incorrect. **A** is incorrect because we don't supply a Consumer for the forEach. **B** is incorrect because we are using invalid syntax for the method reference. Here we can't use a method reference

because we are printing the reading values, not the whole reading object, and so must specify more details about the argument to the consumer's functional method than allowed with a method reference, which is why **C** is incorrect. **C** prints the entire reading object, rather than just the value. (OCP Objectives 3.3, 3.5, and 4.1)

6.  **B** is correct. `human` is a `Supplier`, so its functional method is `get()`. Calling `get()` returns a new `Human` with the name `Joe` and the age `34`.  
 **A, C, D, and E** are incorrect based on the above. (OCP Objective 4.1)
7.  **C** is correct. We know `printAdults` must be a `Consumer` that prints adults in the `people` `ArrayList`. The functional method of the `Consumer` will take a `Human` object. Then we must test that object to see if the `Human` is 18 or older, which we can do by defining a `Predicate` whose functional method returns true if the `Human`'s age is 18 or older. We use the `Predicate` by calling the function method, `test()`. If the `Human` passes the test, we print the `Human`'s name.  
 **A, B, and D** are incorrect. **A** is incorrect because we haven't defined the `printAdults` as a `Consumer`. **B** is incorrect because, although we are defining the `printAdults` `Consumer` in the body of the `Consumer`'s functional method, we are using the `Predicate` wrong. **D** can initially be tempting, but if you look carefully, the syntax is incorrect. Either we must implement the `Consumer` interface with an inner class or with a lambda expression. (OCP Objective 4.1)
8.  **B** is correct. The code does not compile because the variable `longest` from the lambda's enclosing scope must be final or effectively final, and we are trying to change the value of `longest` within the lambda.  
 **A, C, and D** are incorrect for the above reasons. (OCP Objective 2.6)
9.  **D** is correct. The code does not compile because `readingsSupplier` is not being assigned a `Supplier`; rather it is being assigned a `List`.  
 **A, B, and C** are incorrect for the above reasons. (OCP Objective 4.1)
10.  **B** is correct. In the last line of the code, we are first calling the `apply()` method of the `foo BiFunction` and then calling the `apply()` method of the `bar Function`. Looking at the first line of code, we see that `foo`'s `apply()` method is implemented by the lambda expression and takes two arguments, an `integer n` and a `String`. The method concatenates the `String n` times (with a space between) and then returns it. So by calling `apply()` with the

arguments 3 and "foo", the String "foo foo foo" is returned. Then we call the apply() method of bar. The andThen() method of the foo BiFunction passes the value returned from the first BiFunction to the Function whose apply() method is called, so "foo foo foo" gets passed to the apply() method of bar, which is implemented with a lambda expression and takes a String, concatenates "bar" to that String, and returns it, resulting in "foo foo foo bar". (Note, too, this is an example of how you can combine a BiFunction with a Function using andThen()!)

☒ **A**, **C**, and **D** are incorrect for the above reasons. (OCP Objectives 4.1 and 4.3)

- 11.**  **A** is correct. The List replaceAll() method takes a UnaryOperator. In this case, the UnaryOperator's functional method takes a String and returns the lowercase value of that String. Because it's a UnaryOperator, we need only specify one type parameter.

☒ **B**, **C**, and **D** are incorrect. **B** looks like it might be correct if you forget that UnaryOperator uses only one type parameter. **C** is tempting because a UnaryOperator is a type of Function, but replaceAll() specifies a UnaryOperator as an argument. **D** can't work because a Supplier's functional method does not take an argument. (OCP Objective 4.4)

- 12.**  **D** is correct. To use forEach() with a Map, we need a BiConsumer, whose functional method takes two arguments, both Strings.

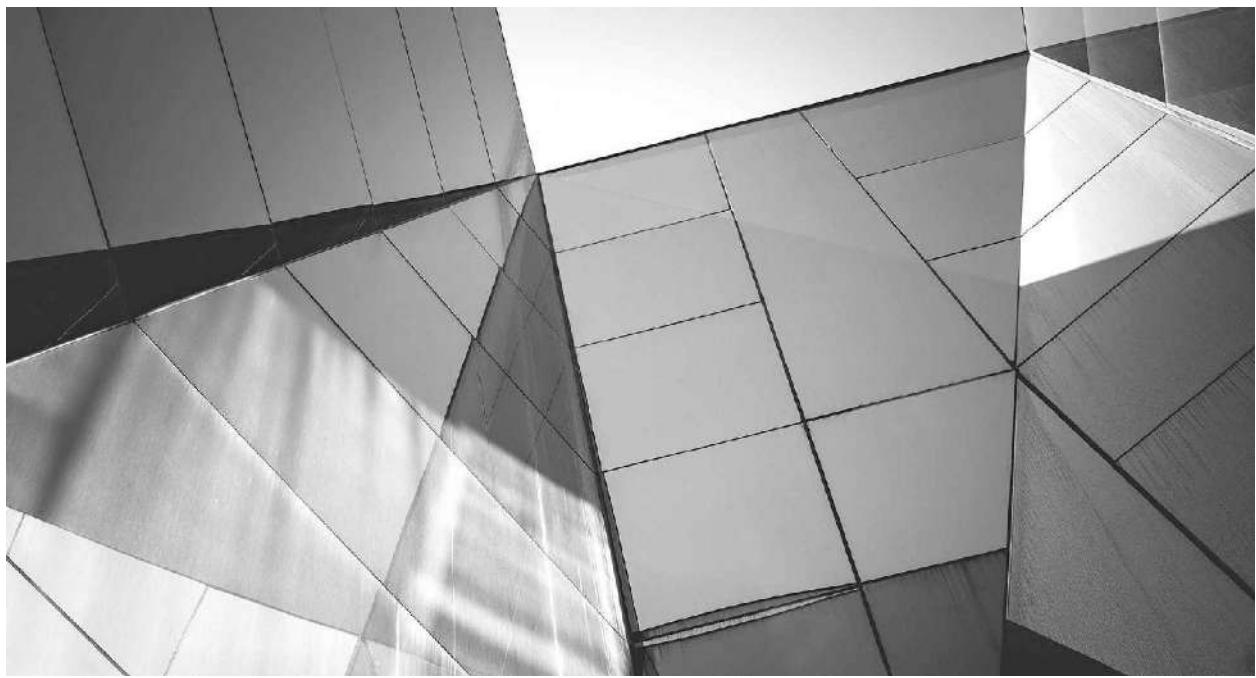
☒ **A**, **B**, and **C** are incorrect for the above reasons. (OCP Objective 4.3)

- 13.**  **C** is correct. To print the to-do item for Tuesday, we need to call the to-do list's checkTodoDay() method, passing the list of to-dos and a Predicate. We can see in the body of the checkTodoDay() method that we call the Predicate's test() method on the map key, which is the day, a String, and if the test passes, we display the value in the map. So the Predicate should test to see if the day is equal to "tuesday" and return true if it is.

☒ **A**, **B**, and **D** are incorrect. **A** could work, but the syntax is incorrect (we should use { } for statements in the body of the lambda) and will cause a compile error. **B** is incorrect because the Predicate's test() method doesn't take two values; it takes only one, the key (day). **D** is incorrect because the test() method is not valid for a String; we need equals() instead. (OCP Objective 4.1)

**14.**  **B** and **C** are correct. To get a double value from a DoubleSupplier, you must use the `getAsDouble()` functional method.

**A** and **D** are incorrect. **A** and **D** are incorrectly using `get()` instead of `getAsDouble()`. (OCP Objective 4.2)



# 9

## Streams

### CERTIFICATION OBJECTIVES

- Collections Streams and Filters
- Iterate Using forEach Methods of Streams and List
- Describe Stream Interface and Stream Pipeline
- Filter a Collection by Using Lambda Expressions
- Use Method References with Streams
- Develop Code to Extract Data from an Object Using peek() and map() Methods Including Primitive Versions of the map() Method
- Search for Data by Using Search Methods of the Stream Classes Including findFirst, findAny, anyMatch, allMatch, noneMatch
- Develop Code That Uses the Optional Class
- Develop Code That Uses Stream Data Methods and Calculation Methods
- Sort a Collection Using the Stream API
- Save Results to a Collection Using the Collect Method and Group/Partition Data Using the Collectors Class
- Use flatMap() Methods in the Stream API
- Use the Stream API with NIO.2
- Use parallel Streams Including Reduction, Decomposition, Merging Processes, Pipelines and Performance



Two-Minute Drill

**Q&A** Self Test

In the previous chapter we looked at how we can use lambda expressions to represent instances of classes that implement functional interfaces and explored

a few places in the JDK where functional interfaces are used, like with the `Logger`'s `log()` method and with the new `Iterable.forEach()` method.

## I

Well, get ready for more. In this chapter about streams, we'll use the functional interfaces you just learned about. The `java.util.stream` package's `Stream` interface has over 30 methods, and about three-quarters of those work with functional interfaces in some way.

Like lambdas and functional interfaces, streams are another new addition in Java 8. The syntax of streams will be new to you and will take some getting used to. As you go through the chapter, take time to practice writing examples and testing the code to get the hang of it. The concept of streams is something quite new, too; at first, you might think streams are just another way of organizing data, like a collection, but they are actually about processing data efficiently, sometimes in ways you might initially find unintuitive given how you're used to programming in Java.

Because streams are related to collections (you can make a stream from a collection type to process the data in the collection) and to lambdas (we'll be using lambdas frequently to operate on streams); on the exam, you'll often see streams questions mixed in with questions that use collections and lambdas. The certification objectives related to streams are also mixed in with collections, functional interfaces, and lambdas, as you'll see throughout this chapter.

## CERTIFICATION OBJECTIVE

### What Is a Stream? (OCP Objective 3.4)

#### 3.4 Collections Streams and Filters.

A *stream* is a sequence of elements (you can think of these elements as data) that can be processed with operations. That's plenty vague as a definition, so to get a better handle on what a stream is, let's make one:

```
Integer[] myNums = { 1, 2, 3 };
Stream<Integer> myStream = Arrays.stream(myNums);
```

Here, `myNums` is an array of three `Integers`. We're using the `Arrays.stream()` method to create a stream from that array. The resulting stream, `myStream`, is a

stream of Integers. What does the stream look like?

```
System.out.println(myStream);
```

This results in the output:

```
java.util.stream.ReferencePipeline$Head@14ae5a5
```

Okay, so far this is clear as mud, right?!

Initially, you might think that a stream is a bit like a data structure, like a `List` or an array. After all, we're making a stream out of an array of three integers:

```
Integer[] myNums = { 1, 2, 3 };
Stream<Integer> myStream = Arrays.stream(myNums);
```

But streams are not a data structure to organize data like a `List` is or an array is; rather, they are a way to process data that you can think of as flowing through the stream, much like water flows through a stream in the real world. An *array* is a way of describing how data is organized and gives you flexible ways to access that data. Now, think of a stream as a way to *operate* on data that's flowing from an array through the stream. We say that the array is the stream's source (like a spring is the source of water for a real stream).

So far, `myStream` is just a description of where we're sourcing data. In this example:

```
Integer[] myNums = { 1, 2, 3 };
Stream<Integer> myStream = Arrays.stream(myNums);
```

we're saying "the source of the data for `myStream` is `myNums`, an array of three integers, 1, 2, 3." That's why when we try to display the stream, we don't see any data; we just see a cryptic description of the object that's describing how to get at the data.

Okay, so let's add an operation and do something with the stream of elements:

```
Integer[] myNums = { 1, 2, 3 }; // create an array
Stream<Integer> myStream = Arrays.stream(myNums); // stream the array
long numElements = myStream.count(); // get the number of elements in the stream
System.out.println("Number of elements in the stream: " + numElements);
```

The result is

Number of elements in the stream: 3

Here, we've added an operation `count()` to `myStream`. The `count()` method simply returns the count of elements in the stream as a long value. Getting a number from the stream, the count of elements, means that stream is done. That is, we can't perform any more operations on the stream because the stream's been turned into one number by the `count()` operation. We say that the `count()` operation is a “terminal operation” because the stream ends there: no more data flows through the stream after the `count()` is done.

The real power of streams comes from the “intermediate operations” you can perform between the source and the end of the stream. For instance, you could filter for even numbers (intermediate operation one), multiply each of those even numbers by 2 (intermediate operation two), and then display the results in the console (terminal operation). By specifying multiple operations on a stream, you are essentially defining a set of things you want to do in a particular order to the data in the source. You could write it all out using a `for` loop and multiple lines of code (possibly even using one or more temporary or new data structures), but with streams, you can be more concise.

You can string together as many of these intermediate operations as you like, but for now, we'll begin with just one:

```
Integer[] myNums = { 1, 2, 3 };
Stream<Integer> myStream = Arrays.stream(myNums);
long numElements =
    myStream
        .filter((i) -> i > 1) // add an intermediate operation to filter the stream
        .count();               // terminal operation, counts the elements in a
                           stream
System.out.println("Number of elements > 1: " + numElements);
```

The result we get with this code is

Number of elements > 1: 2

Now, rather than just counting the elements of the stream, we're first filtering the stream, looking for elements whose value is greater than 1. Notice that the `filter()` method takes a `Predicate`, and recall from the previous chapter that a `Predicate` is an interface with one abstract method, `test()`, that takes a value and returns a boolean. Here, we're representing the `Predicate` with a lambda expression; the value we take as an argument is one element from the stream of data; and we return true if the value of the element is greater than 1. You can read the code

`myStream.filter( (i) -> i > 1 ).count()`

like this: “As the elements of `myStream` flow by, keep only those greater than 1, and count how many elements there are.” The `filter()` method is calling the `test()` method of the `Predicate` you pass to `filter()` behind the scenes, and if the value passes the test, that value gets passed on to `count()`.

Notice that the `filter()` method of `myStream` produces a stream. It's a slightly modified stream consisting only of elements whose values are greater than 1. We're now calling the `count()` method on that filtered stream, rather than on the original `myStream`. Intermediate operations always produce another stream—that's how we can chain multiple operations together to manipulate the data as it flows by in the stream. As long as we keep doing intermediate operations, we keep the stream of data flowing; it ends only when we perform a terminal operation like `count()`. That turns the stream into one thing—say, a

number—and ends the stream. We'll look at streams with multiple intermediate operations shortly.

Now, let's try filtering for elements  $> 2$  and count the results:

```
Integer[] myNums = { 1, 2, 3 };
Stream<Integer> myStream = Arrays.stream(myNums);
long numElements = myStream.filter((i) -> i > 1).count();
System.out.println("Number of elements > 1: " + numElements);
numElements = myStream.filter((i) -> i > 2).count(); // filter by > 2 instead
System.out.println("Number of elements > 2: " + numElements);
```

Run that code and you'll get an exception:

```
Exception in thread "main" java.lang.IllegalStateException: stream has
already been operated upon or closed
```

Hmm. What does it mean the “stream has already been operated upon or closed”?

Streams can be used only once. To turn again to our analogy: Imagine you're standing on the bank of the stream. Once the water in the stream has flowed by you, you can't see it again. That water is gone, and you can't get it back.

No problem, we can just create the stream again. In Java, streams are lightweight objects, so you can create multiple streams if you need to:

```

Integer[] myNums = { 1, 2, 3 };
Stream<Integer> myStream = Arrays.stream(myNums);
long numElements = myStream.filter(i -> i > 1).count();
System.out.println("Number of elements > 1: " + numElements);
numElements =
    Arrays.stream(myNums)
        .filter((i) -> i > 2) // filter by > 2 on a whole new stream
        .count();
System.out.println("Number of elements > 2: " + numElements);

```

We can't reuse `myStream`, so here, we've created a new stream from the `myNums` array. We're filtering that stream using a Predicate that tests for an element greater than 2 and then counting the elements that pass that test. We do all three things: create the stream, filter the stream, and count the elements in one line of code.

In our example, we have only 1 element greater than 2, so we see this as the result:

```

Number of elements > 1: 2    // result of the 1st filter by > 1
Number of elements > 2: 1    // result of the 2nd filter by > 2

```

In Java, a stream is an object that gets its data from a source, but it doesn't store any data itself. The data flowing through the stream can be operated on, multiple times if we want, with intermediate operations, like `filter()`. The stream ends when we use a terminal operation, like `count()`, and once we've used a stream, we can't reuse it.

## CERTIFICATION OBJECTIVE

### **How to Create a Stream (OCP Objectives 3.5 and 9.3)**

3.5 *Iterate using forEach methods of Streams and List.*

9.3 *Use Stream API with NIO.2.*

There are a variety of ways you can create a stream. Given that streams are for data processing, you'll probably find you most often create streams from collections, arrays, and files. The collection, array, or file you use to create the stream is the "source" of the stream and provides the data that will flow through the stream. Remember that the stream itself does not contain any data; it operates on the data that is contained in the source, as that data flows through the stream operations.

## Create a Stream from a Collection

Let's step through some examples of how to create a stream. First, here's how you can create a stream from a basic `List`, one of the collection types:

```
List<Double> tempsInPhoenix = Arrays.asList(123.6, 118.0, 113.0, 112.5,  
    115.8, 117.0, 110.2, 110.1, 106.0, 106.4);  
System.out.println("Number of days over 110 in 10 day period: " +  
    tempsInPhoenix  
    .stream()          // stream the List of Doubles  
    .filter(t -> t > 110.0) // filter the stream  
    .count());          // count the Doubles that pass the filter test
```

This code produces the output:

```
Number of days over 110 in 10 day period: 8
```

What we're doing in this code is first creating a `List` of `Doubles`; then we're using that `List` as a source for a stream that filters values, in this case temperatures greater than 110.0, and counts them. Notice that to create the stream, we're calling the `stream()` method of the `tempsInPhoenix` `List`. This method is a default method of the `Collection` interface and so is inherited by all classes that implement `Collection`. The `stream()` method's signature is

```
default Stream<E> stream()
```

You can see that `stream()` returns an object of type `Stream`, with a generic object type parameter, meaning the stream is a stream of any object type. If we were to split the line where we create the stream in the code above into two and

store the stream in a variable, we'd write the code like this:

```
Stream<Double> tempStream = tempsInPhoenix.stream();
System.out.println("Number of days over 110 in 10 day period: " +
tempStream.filter(t -> t > 110.0).count());
```

using `Double` as the type parameter for the stream whose source is a `List` of `Doubles`.

Don't forget that a `Map` (`HashMap`, `TreeMap`, etc.) is not a collection inheriting from `Collection`. If you want to stream a `Map`, you must first use the `entrySet()` method to turn the `Map` into a `Set`, which *is* a `Collection` type:

```
Map<String, Integer> myMap = new HashMap<String, Integer>();
myMap.put("Boi", 6); myMap.put("Zooey", 3); myMap.put("Charis", 8);
System.out.println("Number of items in the map with value > 4: " +
myMap
    .entrySet()          // get a Set of Map.Entry objects
    .stream()            // stream the Set
    .filter(d -> d.getValue() > 4)      // filter the Map.Entry objects
    .count());           // count the objects
```

Here, we create a `HashMap` and add three items to the `Map`, each with a `String` and `Integer` value—a dog's name and age. Then we stream the dog data from the `Map` and count the number of dogs older than 4.

If we try to stream the `Map` directly, we'll get a compile-time error. Instead, we first call `entrySet()` on the `Map` and then call `stream()` on the resulting `Set`. We then call the `filter()` operation on this stream and get a `Map.Entry` object as the argument to the `Predicate` lambda we pass to the filter. We filter to get only dogs older than 4 and count the results (there are two).

## Build a Stream with `Stream.of()`

`Stream.of()` is quite flexible; it works with any object values, so you can create a `Stream` of `Strings`, `Integers`, `Doubles`, etc. The method signature is

```
static <T> Stream<T> of(T... values)
```

meaning you can supply the `of()` method with any number of arguments, and you get back an ordered stream of those values.

We can use `Stream.of()` to create a stream from our array of Integers, `myNums`, like this:

```
Integer[] myNums = { 1, 2, 3 };
Stream<Integer> myStream = Stream.of(myNums);
```

Then we use `myStream` just like we did before to filter and count the items.

We can make that code even shorter by skipping declaring the array altogether and supply Integer values directly, like this:

```
Stream<Integer> myStream = Stream.of(1, 2, 3);
```

The source of the `Stream` here is a little fuzzy; you aren't actually storing the data values in a data structure first, like you are if you're streaming the `myNums` array. The source is there; it's just hidden behind the scenes.

## Create a Stream from an Array

You've already seen how to use `Arrays.stream()` to stream an array; earlier we created an array of Integers and streamed it. Here's another example of streaming an array, this time, an array of Strings:

```
String[] dogs = { "Boi", "Zooey", "Charis" };           // make an array
Stream<String> dogStream = Arrays.stream(dogs);        // stream it
System.out.println("Number of dogs in array: " + dogStream.count()); // count it
```

And we see 3 as the result for the number of dogs in the array. (Of course, this code is completely contrived as you'd never stream an array to count the number of items, but you get the point, we hope).

Another way to create a stream from an array is to use the `Stream.of()` method. For our example with the Strings of dog names, we could rewrite the line to create the stream like this:

```
Stream<String> dogStream = Stream.of(dogs);
```

## Create a Stream from a File

Using a stream to process data in a file is easy. You know you can use `Files` to read data from a file; the static `lines()` method of `Files` returns a `Stream`, so we can stream the data using the file as the source. Here is the signature of the `Files.lines()` method:

```
public static Stream<String> lines(Path path) throws IOException
```

So to create a stream from a file you write:

```
Stream<String> stream = Files.lines(Paths.get(filename));
```

This sets up the stream, but how do you process the data from the stream? You can use the `Stream's forEach()` method:

```
stream.forEach(line -> ...do something with the line of data from the file...)
```

The `forEach()` method on a stream works much like the `forEach()` method on a collection. It takes a `Consumer`, which we can represent with a lambda expression, and processes each line from the file in the body of the lambda. The `File.lines()` method provides one line at a time from the file as each data element in the stream, which makes processing data from the file easy.

Here's an example to bring this together. We have a file, "dvdinfo.txt," containing the name, genre, and star of a movie, one per line:

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

We want to read the lines of the file, creating a new `DVDInfo` object for each movie entry:

```
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;  
    }  
    // getters and setters here  
}
```

Here's how we can do that using the file as a source for a stream, and using the `forEach()` method to process each line from the file:

```
public class DVDs {
    public static void main(String[] args) {
        List<DVDInfo> dvds = loadDVDs("dvdinfo.txt"); // load the DVDs from a file
        dvds.forEach(System.out::println); // just print the DVDs
    }
    public static List<DVDInfo> loadDVDs(String filename) {
        List<DVDInfo> dvds = new ArrayList<DVDInfo>();
        // stream a file, line by line
        try (Stream<String> stream = Files.lines(Paths.get(filename))) {
            stream.forEach(line -> { // use forEach to display each line
                String[] dvdItems = line.split("/");
                DVDInfo dvd = new DVDInfo(dvdItems[0], dvdItems[1], dvdItems[2]);
                dvds.add(dvd); // for now; there's a better way
            });
        } catch (IOException e) {
            System.out.println("Error reading DVDs");
            e.printStackTrace();
        }
        return dvds;
    }
}
```

We see the output from printing the DVDs:

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

When processing data from a file using a stream with `forEach()` like we do here, remember that within the lambda expression we pass to `forEach()`, all variables must be final or effectively final, so we can't modify a variable directly (e.g., by changing its value to a value from the file). However, also remember that we *can* add to or modify the fields of an object from within the lambda expression, so that's how we can add each DVD to the `List` of `dvds`. The `dvds` `List` is effectively final (we don't try to create a new `List` object and assign it to the `dvds` property within the lambda), but the contents of the `List` can still be modified by adding new DVDs to it. As a result, once the stream terminates (i.e., the last line from the file is read), we have all the DVDs from the file stored in the `dvds` `List`. We must store the DVDs somewhere if we want to use them after the stream is complete because the stream itself doesn't hold any data! However, there's a better way to store the DVDs in the `dvds` `List`, which we'll see later on in the chapter, so put a bookmark here and we'll return to this example later. For more `Files` methods that create streams, check the Online Appendix.

The `forEach()` method of `Stream` is another example of a terminal operation (along with `count()`, which we saw earlier). It does not produce another stream; rather it takes each item flowing by in the stream and consumes it (with a `Consumer`). It doesn't produce anything (i.e., `forEach()` is `void`), so whatever you want to do with the data you're processing with `forEach()` must happen in the `Consumer` you pass to it.

## Primitive Value Streams

As you might expect, there are also primitive streams designed to avoid autoboxing, for `doubles`, `ints`, and `longs`. These are `DoubleStream`, `IntStream`, and `LongStream`, respectively. So, you can create a `DoubleStream` like this:

```
DoubleStream s3 = DoubleStream.of(406.13, 406.42, 407.18, 409.01);
```

Notice there's no type parameter on the stream because this is a stream of double values and that's specified by the type itself, DoubleStream.



***Keep in mind the difference between a Stream<Double> and DoubleStream. The first is a stream of Double objects; the second is a stream of double values. If you create a List<Double> and then stream it:***

```
List<Double> co2Monthly = Arrays.asList(406.13, 406.42, 407.18, 409.01);  
Stream<Double> s1 = co2Monthly.stream();
```

***the stream is type Stream<Double>. Don't get fooled on the exam with this small distinction!***

## Summary of Methods to Create Streams

As you've just seen, there are a variety of ways to make streams—from collections, arrays, files, and values. [Table 9-1](#) summarizes the methods you should be familiar with for the exam. And keep a close eye on those types; it can be easy to slip up and think you're creating a DoubleStream when you're creating a Stream<Double>! Note: This is not an exhaustive list of all the ways to create streams, so see each interface/class for more details and options.

**TABLE 9-1** Methods to Create Streams

Interface/Class	Creates a...	With method...
Collection	Stream<E>	stream()
Arrays	Stream<T>	stream(T[] array) stream(T[] array, int startInclusive, int endExclusive)
Arrays	IntStream, DoubleStream, LongStream	stream(int[] array), stream(double[] array), stream(long[] array) (and versions with start/end like Stream)
Files	Stream<String>	lines(Path path), lines(Path path, Charset cs)
Stream	Stream<T>	of(T... values), of(T t)
DoubleStream	DoubleStream	of(double... values), of(double t)
IntStream	IntStream	of(int... values), of(int t)
LongStream	LongStream	of(long... values), of(long t)

## Why Streams?

You might be wondering why streams were added to Java. So far the examples you've seen are relatively simple, and you could easily accomplish the same thing using iteration over a `Collection`. The code might be a bit more concise, but it doesn't seem to really do anything fantastically different.

The main reason to use streams is when you start doing multiple intermediate operations. So far, we've been performing only one intermediate operation: a filter, using a variety of different `Predicates` to filter the data we get from the stream before we count it. However, when we use multiple intermediate operations, we start seeing the benefits of streams.

First, here's a quick example:

```
List<String> names =  
    Arrays.asList("Boi", "Charis", "Zooey", "Bokeh", "Clover", "Aiko");  
names.stream()                                // Create the stream  
    .filter( s -> s.startsWith("B")  
            || s.startsWith("C"))           // Filter by first letter  
    .filter(s -> s.length() > 3)           // Filter by length  
    .forEach(System.out::println);          // print
```

Here, we've got a list of names. Let's say we want to see the names that begin with “B” or “C” and have a length > 3.

First, we stream the names from the source `List`, `names`. Then, we filter on the starting letter “B” or “C”. The result is that only names that start with “B” or “C” get passed through to the next filter. The next filter checks the length of the string and only passes through strings whose length is greater than 3. Both of these filter operations are “intermediate operations” because the stream continues; the stream is slightly modified (some data elements are discarded if they don't pass the test in the filter), but any data elements left in the stream continue flowing through the stream.

Finally we have a “terminal operation” that ends the stream: the `forEach()` method, which just prints the names. And we see the output:

Charis  
Bokeh  
Clover

You're probably saying to yourself, what's the big deal? We could do the same thing using iteration, right?

The big deal is that streams use something called a “pipeline,” which can, in some circumstances, dramatically improve the efficiency of data processing. To understand how the stream pipeline works, let’s break the above code down a bit more and see what happens in each step.

## CERTIFICATION OBJECTIVE

### The Stream Pipeline (OCP Objective 3.6)

3.6 *Describe Stream interface and Stream pipeline.*

A stream pipeline consists of three parts: a source, zero or more intermediate operations, and a terminal operation. The *source* describes where the data is coming from; the *intermediate operations* operate on the stream and produce another, perhaps modified, stream; and the *terminal operation* ends the stream and typically produces a value or some output.

So the stream pipeline in our code above defines the names `List` as the source for the stream, contains two intermediate operations that filter the data, and then terminates with `forEach()`, which prints the data:

```
List<String> names =  
    Arrays.asList("Boi", "Charis", "Zooey", "Bokeh", "Clover", "Aiko");  
    names.stream() // the source  
        .filter( s -> s.startsWith("B") ) // intermediate op  
        || s.startsWith("C"))  
        .filter(s -> s.length() > 3) // intermediate op  
        .forEach(System.out::println); // terminal op
```

One analogy often used at this point is an assembly line. When you stream the names `List`, the first data element of the list is streamed to the first filter operation. At that point the second filter and the `forEach()` are just waiting for a data element. Once the first filter is complete, that data element is possibly discarded, if it doesn't begin with a "B" or "C", in which case the second filter and the `forEach()` never see it. This makes the subsequent stream more efficient.

If the data element is not discarded, then it's passed along to the second filter, which starts working on that element. In the meantime, the second data element is streamed to the first filter. Just like the assembly line, we now have the first filter working on the second data element, while the second filter is still working on the first one.

If the second filter's `Predicate` test is passed on the first data element, that element is passed along to the `forEach()`, which starts working on it (by printing it out). In the meantime, the second data element is passed to the second filter, assuming it passes the `Predicate` test in the first filter. Then the third data element is streamed to the first filter.

As you can see, the assembly-line analogy works pretty well for the pipeline. Like an assembly line, we can get some efficiencies working with streams for two reasons: First, we can do multiple operations on data in one pass, and Java is optimized so it keeps minimal intermediate state during the operations part of the pipeline. Second, in some circumstances, we can parallelize streams to take advantage of the underlying architecture of the system and do parallel computations very easily. Not all streams can be parallelized, but some can. We'll talk more about parallel streams later in "A Taste of Parallel Streams."

There's one other thing you should know about streams: they are lazy! And, despite what you might think, lazy streams can be more efficient.

## Streams Are Lazy

Look again at the code:

```
names.stream()                      // the source
    .filter( s -> s.startsWith("B") ) // intermediate op
    || s.startsWith("C"))
    .filter(s -> s.length() > 3)     // intermediate op
    .forEach(System.out::println);    // terminal op
```

What do you think happens if we write the following instead:

```
names.stream()                      // the source
    .filter( s -> s.startsWith("B") ) // intermediate op
    || s.startsWith("C"))
    .filter(s -> s.length() > 3);   // intermediate op
```

That is, we've written the pipeline without the `forEach()` terminal operation.

You know that what you get back from the second `filter()` is another stream. But has anything actually happened yet? Is any data flowing through the stream?

The answer is no. Nothing has happened. Going back to the assembly-line analogy, the worker at station 1 (filter one) is still sitting idle, as is the worker at station 2 (filter two). That's because no terminal operation has been executed yet. We've got everything set up, but nothing to kick-start the data processing.

Not until the `forEach()` is executed does anything happen. As soon as the terminal operation is executed, then the assembly line kicks into gear, and the data starts flowing from the source through the stream and into the operations.

It's important to note here that our analogies of the water stream and the assembly line break down when you remember that streams don't hold any data. Even though we talk about data elements as "flowing through the stream," the data is never "in" the stream (unlike water that really is in a real-life stream and items that really are in an assembly line). The operations you define on a stream are how you specify ways to manipulate the data that's in the source in a particular order. In that sense, of course, streams are lazy because no data is flowing, even though it helps us to think about streams that way.

This laziness makes streams more efficient because the JDK can perform optimizations to combine the operations efficiently, to operate on the data in a single pass, and to reduce operations on data whenever possible. If it's not

necessary to run an operation on a piece of data (e.g., because we've already found the data element we're looking for, or because we've eliminated a data element in a prior intermediate operation, or because we've limited the number of data elements we want to operate on), then we can avoid even getting the data element from the source.

For most, if not all, of the simple examples we'll be working with in this book in preparation for the exam, the stream pipeline will not offer any tremendous advantage beyond thinking about data processing in a new way, and perhaps more concise code. However, once you get in the real world and start working with large amounts of data, you may realize the benefits of streams from an efficiency standpoint, especially if you can work with streams that can be parallelized (which we'll get to later, as we said before).

## CERTIFICATION OBJECTIVE

### Operating on Streams (OCP Objectives 3.7 and 5.1)

3.7 *Filter a collection by using lambda expressions.*

5.1 *Develop code to extract data from an object using peek() and map() methods including primitive versions of the map() method.*

You may have heard the expression “map-filter-reduce”: a general abstraction to describe functions that operate on a sequence of elements. It perfectly describes what we are often doing with streams: we might “map” an input to a slightly different output, then “filter” that output by some criteria, and finally “reduce” to a single value or to a printed output. When you map-filter-reduce, you are simply specifying a sequence of operations to be performed on the data in the stream’s source that leads to a result. As we said earlier, instead of using a stream, you could create a `for` loop over the source data structure yourself, apply each of the operations in turn, accumulate the results into a new data structure, and you’d accomplish the same thing as a stream. The advantage of a stream is twofold: you can (often) write a sequence of operations to perform on the stream more concisely, and you can (sometimes) take advantage of some optimizations the JDK does under the covers to perform those operations more efficiently.

With streams, we take the map-filter-reduce abstraction and translate it into operations using the `map()`, `filter()`, and `reduce()` methods (and their variations) on a stream.

You’ve already seen the Java Stream’s `filter()` method. As we said earlier,

`filter()` takes a `Predicate` and tests each element in the stream pipeline, producing a stream of elements that pass the test.

The `Stream`'s `map()` method is another intermediate operation; however, unlike `filter()`, `map()` doesn't winnow down the elements of a stream; rather, `map()` transforms elements of a stream. For instance, `map()` might compute the square of a number, mapping a stream of numbers to a stream of their squares. Or `map()` might get the age of a `Person` so that the next step (say, a `filter`) can find ages greater than 21.

The `map()` method takes a `Function`. Remember from the previous chapter that the purpose of a `Function` is to transform a value. The `Function`'s `apply()` method takes one value and produces another value (not necessarily of the same type), so the `Function` you pass to `map()` will "map" one value to another. The `Function`'s functional method `apply()` that you pass into `map()` gets called by `map()` behind the scenes, and the value returned from `apply()` gets passed on to the next operation in the stream pipeline.

Reduce is both a general method, `reduce()`, as well as a specific method like `count()` and others you'll see shortly. All reductions are also terminal operations. Reduction operations are designed to combine multiple inputs into one summary result, which could be a single number, like the reduction operation `count()` produces, or a collection of values, which we'll see examples of later.

The `Stream`'s `reduce()` method is a general method for reducing a stream to a value; the basic version of `reduce()` takes a `BiFunction` (a `Function` whose `apply()` method takes two arguments and produces one value) and applies it to pairs of elements from the stream, returning a value. You can think of `reduce()` as a way to "accumulate" a result, such as summing the values of a stream. Reductions like `count()`, `sum()`, and `average()` are defined as methods of streams; if you want a custom reduce function you can define it yourself using `reduce()`.

(Notice how all those functional interfaces from the previous chapter are showing up here?)

As we work with map-filter-reduce, keep in mind that the stream pipeline specifies a sequence of operations to perform on the data in a source data structure, like an array. There is one stream of data that gets modified and winnowed (the *map-filter* part) as it passes through the pipeline, and eventually, the data elements in the stream are reduced (the *reduce* part) to a value (when the stream is no longer a stream again, but rather a single value or another data structure).

Let's take a look at a map-filter-reduce set of operations on a stream. We'll define a `List` of `Integers`, take the square of each `Integer`, test to see whether the square is greater than 20, and if it is, we'll add 1 to a count, `result`. Before we do, we'll look at how we might do this with Java 7, so we can compare it with how we compute this using streams.

Here's the Java 7 way:

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6);
long result = 0;
for (Integer n : nums) {
    int square = n * n;
    if (square > 20) {
        result = result + 1;
        System.out.println("Square of " + n + " is: " + square);
    }
}
System.out.println("Result: " + result);
```

With the output:

```
Square of 5 is: 25
Square of 6 is: 36
Result: 2
```

We're simply creating a `List`, and using a `for` loop to iterate over the list, compute the square, print out the square if it is greater than 20, and print the number of squares  $> 20$ .

And here's the Java 8 way, with streams:

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6);
long result = nums.stream()
    .map(n -> n * n)           // map values in stream to squares
                                // (map intermediate op)
    .filter(n -> n > 20)        // keep only squares > 20
                                // (filter intermediate op)
    .count()                    // count the squares > 20 (reduction op)

System.out.println("Result (stream): " + result);
```

With the output:

**Result (stream) : 2**

Here, we're streaming the `List` of `Integers`, first mapping the number from the stream to its square, then filtering the squares so that we keep only those squares greater than 20, and then reducing using the `count()` operation to count the squares > 20.

Which code do you like better? You're very familiar with the `for` loop iteration; the stream pipeline with a map-filter-reduce sequence is likely new to you. They both accomplish the same thing. Using streams like this has the potential to be a bit more efficient (not so much with this particular example because it's so simple) and is more concise.

Notice that we can't print the number with its square using the streams way. Why? Because by the time we get to the `count()` reduction operation, the only elements we have access to in the stream are the squares of the numbers. The original numbers are stored only in the original `List`. And don't be tempted to try to somehow keep track of the index of the square; that won't work and doesn't fit the patterns of how we use stream pipelines.

We can create a “sort of” solution to the problem like this, however:

```
long result = nums.stream()
    .peek(n -> System.out.print("Number is: " + n + ", "))
    .map(n -> n * n)
    .filter(n -> n > 20)
    .peek(n -> System.out.print("Square is: " + n + ", "))
    .count();
```

And see the output:

```
Number is: 1, Number is: 2, Number is: 3, Number is: 4, Number is: 5,
Square is: 25, Number is: 6, Square is: 36, Result (stream): 2
```

The method `peek()` is an intermediate operation that allows you to “peek” into the stream as the elements flow by. It takes a `Consumer` and produces the same exact stream as it’s called on, so it doesn’t change the values or filter them in any way. Here we’re using it to peek at the numbers in the original stream before we map those numbers to squares and then filter them. We see all the numbers in the original stream, rather than just the numbers that have squares greater than 20.

You might feel a bit limited by this, but remember that streams are designed for data processing, so typically we’re looking for the result of a sequence of operations applied to a big set of data, and, other than when we’re debugging, we’re not going to be looking at the data as it flows by.

A more typical example of a map-filter-reduce operation might be to stream a data set of readings, map the stream to get a particular value from a reading, filter that stream to eliminate any outliers, and then find the average reading of that stream. We’ll tackle this example next, and in the process, you’ll see a new way to map values and learn about the concept of “optional” values.

## CERTIFICATION OBJECTIVE

### Map-Filter-Reduce with `average()` and Optionals (OCP Objectives 5.3 and 5.4)

5.3 *Develop code that uses the Optional class.*

#### *5.4 Develop code that uses Stream data methods and calculation methods.*

Imagine you have a piece of equipment that's taking a reading once per week, and you want to find the average of the readings you've measured so far this year. You know that the readings should probably be between 406 and 407, so you decide to throw away readings that are less than 406 or greater than 407 because those may be errors.

You start by creating a class for the readings:

```
class Reading {  
    int year;  
    int month;  
    int day;  
    double value;  
  
    Reading(int year, int month, int day, double value) {  
        this.year = year; this.month = month;  
        this.day = day; this.value = value;  
    }  
}
```

Each reading gets stored with its value, along with the year, month, and day, in a `Reading` object, and the `Reading` objects are stored in a `List`:

```
List<Reading> readings = Arrays.asList(  
    new Reading(2017, 1, 1, 405.91),  
    new Reading(2017, 1, 8, 405.98),  
    new Reading(2017, 1, 15, 406.14),  
    new Reading(2017, 1, 22, 406.48),  
    new Reading(2017, 1, 29, 406.20),  
    new Reading(2017, 2, 5, 407.12),  
    new Reading(2017, 2, 12, 406.03));
```

Now let's use a stream to find the average of the readings that are between 406 and 407. The first thing we'll do is stream the readings:

```
readings.stream()
```

For this computation, we're interested only in the value portion of the readings because we're trying to get the average. We can map each reading to its value, essentially converting the reading to a single double value. But we can't use `map()` because `map()` takes a `Function` whose `apply()` method takes an object and returns an object. Here's the method signature of `map()`:

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper)
```

As we said earlier, `map()` takes a `Function` (called `mapper` in the signature above). `map()` applies that `Function` (by calling the `apply()` method) on each value in the stream. Recall that the `Function`'s `apply()` method takes an object and returns an object. So if we call `map()` on a `Stream` of objects, like we want to do here with our stream of `Reading` objects, what we get back is also a stream of objects. But `Reading.value` is a `double`, not a `Double`, so what we really want to get back is a stream of `doubles`.

Is there a solution? Yes, of course! The `Stream.mapToDouble()` method takes a `ToDoubleFunction`, whose `applyAsDouble()` method takes an object and returns a `double`:

```
double applyAsDouble(T value)
```

This is exactly what we need, so the next step in the stream pipeline is

```
mapToDouble():

    readings.stream().mapToDouble(r -> r.value)
```

We use a lambda expression for the `ToDoubleFunction`, pass in a `Reading` object, `r`, to the `applyAsDouble()` method (which gets called by `mapToDouble()` behind the scenes), and get back the double value, `r.value`.

Keeping track of the type in these situations is tricky! Notice: what we get back from the `mapToDouble()` method of the `Stream` that we created with `readings.stream()` is a `DoubleStream`:

```
DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)
```

So by calling `mapToDouble()`, we've converted the initial `Stream` of `Reading` objects (`Stream<Reading>`) into a stream of doubles, and that stream has the type `DoubleStream`.

The capitalization on these types can be very confusing. Don't mix up a `DoubleStream` with a `Stream<Double>` here, or forget that `ToDoubleFunction`'s `applyAsDouble()` method returns a `double`. It's easy to get mixed up on this issue and think you're working with `Doubles` when you're actually working with `doubles` (or vice versa), and in this case, it matters a lot, as you'll see shortly.

After all that, where are we?

```
readings.stream().mapToDouble(r -> r.value)
```

We've mapped the stream of `Reading` objects into a stream of doubles, and next we want to filter the stream so we keep only the values greater than or equal to 406.0 and less than 407.0:

```
readings.stream()

    .mapToDouble(r -> r.value)

    .filter(v -> v >= 406.0 && v < 407.0)
```

We've changed the variable we're using in the lambda expression in `filter()` from `r` (for `Reading`) to `v` (for `value`) as a reminder that the stream we're filtering is a `DoubleStream`: that is, a stream of the double values from the `List` of `Readings`.

Note again that at this point in the computation, nothing has actually

happened. The stream is lazy! No computation is going on until we get to the reduce part of the map-filter-reduce: the terminal operation. So far we have a source (the `List` of `Readings`), a stream, and two intermediate operations.

The reduction operation we are going to use is the `average()` method. But if you look for `average()` in the `Stream` interface, you will not find it. That's because `average()` is a method of the `DoubleStream` interface (and its primitive cousins, `IntStream` and `LongStream`). That makes sense because `average()` is designed to work with numbers, not objects. And it's convenient that we mapped the stream of `Readings` into a stream of doubles (we *do* plan ahead for these examples).

Here is the type signature for the `DoubleStream`'s `average()` method:

```
OptionalDouble average()
```

You probably expected `average()` to return a `double`, right? And what on earth is this `OptionalDouble` thing?

An `OptionalDouble` is one of several types of optionals that are new in Java 8, which represent values that may or may not be there. Strange! Why would the average of our readings not be there?

Well, think about it this way. What if the source is empty? That is, what if our `List` of `Readings` is empty? Or what if, when we filter the stream, the filter eliminates all the values from the stream, so the result of the filter is an empty stream? We don't know until we get to `average()` whether there's anything actually *in* the stream (remember, the stream is lazy). And, as you know, the average value of a set of values is computed by adding up all the values and then dividing by the number of values. If the number of values is 0, then you'd be dividing by 0, which, as we all know, is a *very bad thing to do*.

Whenever we compute the average of some collection of values, we always have to check to make sure there's at least one of them so we don't get ourselves into this situation. But if we forget to check to see whether our source is empty before we stream it, or, perhaps more likely, we're in the middle of a stream pipeline and we've created an empty stream by filtering out all the values, presto: we're in a situation where we're trying to compute the average of an empty stream and we're going to get ourselves into trouble. Optionals help prevent that trouble.

It doesn't make sense to return `null` in this situation because we're expecting a primitive `double`. And it doesn't make sense to return `0.0` because that could be a valid result for the average. Instead, we get back an `OptionalDouble`, which

is a double value that might or might not be there. To finish off this line of code, we terminate the stream pipeline with a call to `average()` and store the result in an `OptionalDouble` variable, `avg`:

```
OptionalDouble avg =  
    readings.stream()  
        .mapToDouble(r -> r.value)  
        .filter(v -> v >= 406.0 && v < 407.00)  
        .average();
```

If you try to print the value of `avg`, like this:

```
System.out.println("Average of 406 readings: " + avg);
```

what you see is

```
Average of 406 readings: OptionalDouble[406.2125]
```

The correct result—that is, the average of the readings—is 406.2125, but we’re seeing that result wrapped an `OptionalDouble`. How do you get a double value out of an `OptionalDouble`? You use the `getAsDouble()` method:

```
System.out.println("Average of 406 readings: " + avg.getAsDouble());
```

And now you’ll see the result:

```
Average of 406 readings: 406.2125
```

The problem with this code, however, is if the `OptionalDouble` is empty, then you’ll get a `NoSuchElementException`. Just because you get an `OptionalDouble` back from `average()` doesn’t mean you completely abdicate responsibility for checking to make sure there’s a value there before you try to use it!

If you do try to call `getAsDouble()` on an empty `OptionalDouble`, you’ll see

```
Exception in thread "main" java.util.NoSuchElementException: No value present
```

meaning you’re trying to get a double that isn’t there. A better way to write this code is to first check to make sure there is a value *present* in the

`OptionalDouble` with the `isPresent()` method:

```
if (avg.isPresent()) {  
    System.out.println("Average of 406 readings: " + avg.getAsDouble());  
} else {  
    System.out.println("Empty optional!");  
}
```

If there's a value, we print it; otherwise, we print a message saying the optional must have been empty. Now if you run the code on an empty `List`, or a `List` with no values in the 406–407 range, you won't get a runtime error and you'll see the message:

`Empty optional!`

Whew! We finally have our average value, 406.2125, for our `List` of `Readings` using a stream pipeline. That was a lot to take in. We introduced map-filter-reduce, talked about how map works in more detail and how map and filter differ, and introduced optionals, which we'll be returning to in more detail.

## e x a m watch

*There's ample room for confusion with `DoubleStream`, `Stream<Double>`, `getAsDouble()`, etc. Adding to the confusion is that sometimes a reduction will return an `OptionalDouble` and sometimes an `Optional<Double>` (and, likewise, for `long/Long` and `int/Integer`).*

*This is a good time to pay close attention to types, type parameters, and method signatures. There will be times on the exam when you'll have to select the correct type as part of your answer, and knowing the difference between `DoubleStream` and `Stream<Double>` or `OptionalDouble` and `Optional<Double>` will be the key to choosing the correct answer.*

## Reduce

What we did with this example code is take a source, a `List` of `Readings`, stream it, map the readings (mapping a `Reading` to a `double`), filter the values (keeping only values between 406 and 407), and then reduce (by taking the average). Let's return to the reduce part of map-filter-reduce.

You've seen a couple of reductions so far: `count()` and `average()`. The `count()` method is defined for both `Stream` and the primitive versions of `Stream`: `IntStream`, `LongStream`, and `DoubleStream`, while `average()` is defined only for the primitive streams. Other handy methods defined on the primitive streams include `min()`, `max()`, and `sum()`, which result in the minimum value in a stream, the maximum value in a stream, and the sum of the values in a stream.

Like `average()`, `min()` and `max()` also return optional values. If you are looking for the `max()` reading in our `List` of `Readings`, you'd write:

```
OptionalDouble max =  
    readings.stream()  
        .mapToDouble(r -> r.value)  
        .max();  
  
if (max.isPresent()) {  
    System.out.println("Max of all readings: " + max.getAsDouble());  
} else {  
    System.out.println("Empty optional!");  
}
```

Looking at the type signature of `sum()`, however, you will probably notice that `sum()` does not return an optional value. Rather, `DoubleStream.sum()` returns a `double`. Why do `average()`, `max()`, and `min()` return optionals, but `sum()` does not?

As with `average()`, it makes no sense to take the `max()` or `min()` of an empty stream. Therefore, an optional value is returned, indicating that a value may not exist.

Taking the `sum()` of an empty list makes a little more sense if you assume the sum of an empty stream is 0.0. And, in fact, if you try to `sum()` an empty `DoubleStream`, that's exactly what you'll get:

```
List<Reading> readings2 = Arrays.asList(); // empty list for testing sum  
double sum = readings2.stream().mapToDouble(r -> r.value).sum();  
System.out.println("Sum of all readings: " + sum);
```

produces the output:

```
Sum of all readings: 0.0
```

By default, the sum of an empty stream is 0. Let's take a look at how you can write your own reduction methods with `reduce()` and have an opportunity to provide a default value in case you have an empty stream.

## Using `reduce()`

The methods `count()`, `average()`, `min()`, `max()`, and `sum()` are all reduction methods already defined on streams. You can also define your own reductions using the `reduce()` method. We'll rewrite the code to sum the values in the (original, nonempty) `readings` list using `reduce()` rather than `sum()`.

What is `sum()` doing to “reduce” a `DoubleStream` of `double` values into one value? It's taking values as they flow from the stream and adding them up, so that's what we need to do in `reduce()` if we want to create our own sum reduction method.

If we look at the type signature of `DoubleStream.reduce()`, we see

```
OptionalDouble reduce(DoubleBinaryOperator op)
```

In the previous chapter, we looked only at `UnaryOperators`, but hopefully, if you studied the Java 8 Functional APIs a bit further, you ran across `BinaryOperators`, which are like `UnaryOperators` except their functional methods take two values as arguments, rather than one. And recall that operators are a special case of `Function` in that the arguments and the return value of the functional method must be the same type.

A `DoubleBinaryOperator` is an operator whose functional method, `applyAsDouble()`, takes two `double` values and returns a `double`.

To write our own method to sum all the values in the stream using `reduce()`, we pass a `DoubleBinaryOperator` to `reduce()`, which adds the two arguments together and returns the sum:

```

OptionalDouble sum =
    readings.stream()
        .mapToDouble(r -> r.value)
        .reduce((v1, v2) -> v1 + v2);
if (sum.isPresent()) {
    System.out.println("Sum of all readings: " + sum.getAsDouble());
}

```

And we see the output:

Sum of all readings: 2843.860000000006

The `reduce()` method sums all the values in the stream, two at a time, to get a total sum. In other words, it computes the final result by “accumulating” the values coming in from the stream.

You might wonder, well, how does this work when we are streaming elements one at a time? If you’re thinking of the assembly-line analogy, you might be thinking of `reduce()` as the last station on the assembly line and imagining `reduce()` getting one element at a time.

Again, think of reductions as *accumulators*: they accumulate values from the stream so they can compute one value. It’s as if the final station in the assembly line is putting all the values from the stream into one big box in order to do the terminal operation on them.



*Don’t forget: this analogy of the assembly line isn’t quite correct. We think of elements flowing through a stream one at a time, like items on an assembly line, but in reality, it doesn’t quite work that way. There is no data in a stream; we’re simply defining operations on the stream’s source (a data structure) that happen in a particular sequence and then get a result by terminating with a reduction. Although it’s helpful to think of it like a stream of water or an assembly line as a way to understand streams, how the JDK handles that computation internally is probably quite*

**different: with optimizations and even parallelization, the reality is not like the analogy.**

Take another look at the code above and notice that when we switched from `sum()` to `reduce()`, we had to go back to using `OptionalDouble` for the result. That's because `reduce()` is a general reduction function that takes any `DoubleBinaryOperator`. We know that the `applyAsDouble()` method must produce a number, but in the situation where the stream is empty, that `applyAsDouble()` method will never get called, so we need a way to say there might not be a result at all. Again, that's what the `OptionalDouble` says.

There's another `reduce()` method in `DoubleStream` that takes an *identity* along with an accumulator function and returns a double:

```
double reduce(double identity, DoubleBinaryOperator op)
```

The `identity` argument serves two roles: it provides an initial value and a value to return if the stream is empty.

If you provide the `identity` value, you're providing an initial value for the result of applying the accumulator function, `op`. The sum is computed by adding values from the stream to this initial value, and when the stream is empty, the `identity` provides a default result for the sum. For a method that produces a sum, it makes sense for that `identity` value to be 0.0. Because we have a value to return when the stream is empty, we no longer have to use an `OptionalDouble` because we'll always get a value back:

```
double sum = readings.stream()
    .mapToDouble(r -> r.value)
    .reduce(0.0, (v1, v2) -> v1 + v2);           // provide an identity
                                                // value
System.out.println("Sum of all readings: " + sum); // print 0.0 if stream
                                                // is empty
```

## Associative Accumulations

We just showed you how to use `reduce()` to build your own function that sums up the values in a stream. So you might be thinking, hey I could try the same

thing for average. You might even try writing the following code:

```
OptionalDouble avgWithReduce =  
    readings.stream()                                // stream the readings  
        .mapToDouble(r -> r.value)                  // map to double values  
        .filter(v -> v >= 406.0 && v < 407.00)    // filter 406 values  
        .reduce((v1, v2) -> (v1 + v2) / 2);         // take the average  
  
if (avgWithReduce.isPresent()) {  
    System.out.println("Average of 406 readings: " +  
        avgWithReduce.getAsDouble());  
} else {  
    System.out.println("Empty optional!");  
}
```

What you might see is

Average of 406 readings: 406.31125

Not only is that answer different from the one we got before (which was 406.2125), but you also might get a different answer from ours. What is going on here?

Let's take a look at how we replaced the call to `average()` with a call to `reduce()`:

```
reduce( (v1, v2) -> (v1 + v2) / 2)
```

As we did with sum, we're passing a `DoubleBinaryOperator` to reduce, whose functional method takes two values. We're summing those two values and dividing by two. The idea is that `reduce()` operates on two values from the stream at a time, so we'll get an average for two values and then average that result with the next value that comes in from the stream. It sounds like it should work, but it clearly doesn't. Something's up.

The problem is that `average()`—unlike `sum()`, `max()`, and `min()`—is not *associative*. If you stretch your mind back to high-school algebra, you might (or

might not) recall that an operator is associative if the following is true:

$$(A \text{ operator } B) \text{ operator } C = A \text{ operator } (B \text{ operator } C)$$

Addition is associative, thus,

$$(1 + 2) + 3 = 1 + (2 + 3)$$

The parentheses indicate which operators to apply first. So, on the left, we compute  $1 + 2$ , get 3, and then add 3, to get 6. On the right, we compute  $2 + 3$ , get 5, and then add 1, to get 6 again. We get the same result; therefore, addition (i.e., `sum()`) is associative.

Our problem with the code above that replaces the call to `average()` with a `reduce()`:

```
reduce( (v1, v2) -> (v1 + v2) / 2)
```

is that `average` is *not* an associative operator. That might seem odd; it feels like it should be associative, but it's really not.

The average of 1, 2, and 3 is 2, right?

$$1 + 2 + 3 = 6. 6 / 3 = 2$$

Yep.

Okay, now try this:

The average of 1 and 2 = 1.5

The average of 1.5 and 3 = 2.25

And this:

The average of 2 and 3 = 2.5

The average of 2.5 and 1 = 1.75

Average is clearly *not* associative!

Reduction operations *must* be associative in order to work correctly with streams. That's why you can't define your own average function with `reduce()`; `average` is not associative. So to take the average of a stream, you must use the `average()` method.

What about `min()` and `max()`? They are both associative reductions. Try

writing your own versions using `reduce()`. Convince yourself that these operators really are associative and then try writing the code to find the min and max of a stream of values.

## map-filter-reduce Methods

With map, as with all things streams, there are a variety of options depending on the types you are working with. Review [Table 9-2](#) to get a sense of the patterns of names used with streams and primitive streams. (Note: This is not an exhaustive list of all the ways to map, filter, and reduce, so see each interface for more details and options.)

**TABLE 9-2** Methods to Map, Filter, and Reduce

Interface	Method	Returns...
<b>Stream</b>	map(Function<? super T, ? extends R> mapper)	Stream<R>
<b>Stream</b>	mapToDouble( .ToDoubleFunction<? Super T> mapper),  mapToInt( ToIntFunction<? Super T> mapper),  mapToLong( ToLongFunction<? Super T> mapper)	DoubleStream,  IntStream,  LongStream
<b>DoubleStream</b>	map( DoubleUnaryOperator mapper), mapToInt( DoubleToIntFunction mapper),  mapToLong( DoubleToLongFunction mapper),  mapToObj( DoubleFunction<? Extends U> mapper)	DoubleStream,  IntStream,  LongStream,  Stream<U>



Interface	Method	Returns...
<code>IntStream</code>	Like <code>DoubleStream</code> , but for ints	
<code>LongStream</code>	Like <code>DoubleStream</code> , but for longs	
<code>Stream</code>	<code>filter(Predicate&lt;? super T&gt; predicate)</code>	<code>Stream&lt;T&gt;</code>
<code>DoubleStream</code>	<code>filter(DoublePredicate predicate)</code>	<code>DoubleStream</code>
<code>IntStream</code>	<code>filter(IntPredicate predicate)</code>	<code>IntStream</code>
<code>LongStream</code>	<code>filter(LongPredicate predicate)</code>	<code>LongStream</code>
<code>Stream</code>	<code>reduce(BinaryOperator&lt;T&gt; accumulator), reduce(T identity, BinaryOperator&lt;T&gt; Accumulator)</code>	<code>Optional&lt;T&gt;, T</code>
<code>DoubleStream</code>	<code>reduce(DoubleBinaryOperator op), reduce(double identity, DoubleBinaryOperator op)</code>	<code>OptionalDouble, double</code>
<code>IntStream</code>	<code>reduce(IntBinaryOperator op), reduce(int identity, IntBinaryOperator op)</code>	<code>OptionalInt, int</code>
<code>LongStream</code>	<code>reduce(LongBinaryOperator op), reduce(long identity, LongBinaryOperator op)</code>	<code>OptionalLong, long</code>
<code>Stream</code>	<code>count()</code>	<code>long</code>
<code>DoubleStream,</code> <code>IntStream,</code> <code>LongStream</code>	<code>count(), sum()</code>	<code>long, double, int, long</code>
<code>DoubleStream,</code> <code>IntStream,</code> <code>LongStream</code>	<code>average()</code>	<code>OptionalDouble</code>
<code>DoubleStream,</code> <code>IntStream,</code> <code>LongStream</code>	<code>max(), min()</code>	<code>OptionalDouble, OptionalInt, OptionalLong</code>

The key idea to remember with map-filter-reduce is the purpose of each: `map()` maps values to modify the type or create a new value from the existing value, but without changing the number of elements in the stream you’re working with; `filter()` potentially winnows the values you’re working with, depending on the result of the Predicate test; and `reduce()` (and its equivalents) changes the stream of values to one value (or a collection of values) as a terminal operation.

## CERTIFICATION OBJECTIVE

### Optionals (OCP Objective 5.3)

5.3 *Develop code that uses the Optional class.*

We’ve explored optionals a bit, and now it’s time to dig in a bit more as optionals are an important part of working with streams and you’ll encounter optionals on the exam. You’ll need to make sure you know which stream operations return optionals and the correct type for those optionals. [Table 9-3](#) summarizes the methods that result in optionals that we cover in this chapter.

**TABLE 9-3** Stream Methods That Return Optionals

Interface	Method	Returns...
Stream	findAny()	Optional<T>
Stream	findFirst()	Optional<T>
Stream	max(Comparator<? super T> comparator)	Optional<T>
Stream	min(Comparator<? super T> comparator)	Optional<T>
Stream	reduce(BinaryOperator<T> accumulator)	Optional<T>
DoubleStream	average()	OptionalDouble
DoubleStream	findAny(), findFirst()	OptionalDouble
DoubleStream	max(), min()	OptionalDouble
DoubleStream	reduce(DoubleBinaryOperator op)	OptionalDouble
IntStream, LongStream	Similar methods to DoubleStream	OptionalInt, OptionalLong

What is an optional? It's a container that may or may not contain a value. You can think of an optional as a wrapper around a value and that wrapper provides various methods to determine whether a value is there and, if it is, to get that value.

Optionals show up a lot with streams, because, as you've seen, if a stream is empty at any part of the pipeline, then there's a chance that no value will be returned by the terminal operation of a stream pipeline. Optionals provide a way

for the JDK to handle that situation without having to throw a runtime exception.

The types of optionals include `Optional<T>` (for objects) and three primitive optionals, `OptionalDouble`, `OptionalInt`, and `OptionalLong`.

You've seen how we used `OptionalDouble` with the `average()` operation; now let's take a look at how we might use `Optional<Double>` for comparison:

```
Stream<Double> doublesStream =  
    Stream.of(1.0, 2.0, 3.0, 4.0);           // stream of doubles  
Optional<Double> aNum = doublesStream.findFirst(); // first the first double  
if (aNum.isPresent()) {                         // check to see if aNum  
    // has a value  
    System.out.println("First number from the doubles stream: " + aNum.get());  
} else {  
    System.out.println("Doubles stream is empty");  
}
```

You should see the output:

```
First number from the doubles stream: 1.0
```

We're creating a stream of `Double` values, using `Stream.of()` (notice that the doubles are being autoboxed into `Doubles` to create the `Stream<Double>`), and then we're using the stream terminal operation method, `findFirst()`, to find the first element of the stream.

We'll get into `findFirst()` in more detail later, but what it does, as you might guess, is find the first element of the stream. Typically, you'll use `findFirst()` after filtering the stream, but here, we just want to return the first element of the original stream.

That element is a `Double` object, so the type we use for the return value from `findFirst()` is `Optional<Double>`. Note that `OptionalDouble` will not work here! Why? Because `findFirst()` is operating on a stream of `Doubles`, not a stream of doubles. Again, pay very close attention to the types here.

Just like before, now that we have an optional value, we need to first test to see whether it's present, with `isPresent()`, before trying to get that `Double` value from the `optional`. Because we're working with a `Double` object rather

than a primitive type, the method we use to get the value is `get()`.



***Recall that the method for `OptionalDouble` is `getAsDouble()`, and note the patterns for the method names for optionals and primitive optionals:***

- ***Get a value from an `Optional<T>`: `get()`***
- ***Get a value from an `OptionalDouble`: `getAsDouble()`***
- ***Get a value from an `OptionalInt`: `getAsInt()`***
- ***Get a value from an `OptionalLong`: `getAsLong()`***

Another way to test to see whether an optional contains a value and, if it does, to then get that value, is the `ifPresent()` method. This method takes a Consumer and tests to see whether the optional value is present; if it is, then the unwrapped value is passed to the consumer:

```
Stream<Double> doublesStream = Stream.of(1.0, 2.0, 3.0, 4.0);
Optional<Double> aNum = doublesStream.findFirst();
aNum.ifPresent(n ->
    System.out.println("First number from the doubles stream: " + n));
```

Note here that the type of `n` is `Double`, not `Optional<Double>`, so to print `n`, we just write `n`, not `n.get()`. You'll see the output:

First number from the doubles stream: 1.0

If your `doublesStream` is empty, then you will not see any output at all.

You can create your own optionals, too. For instance, here's how to create an `Optional<Dog>` (using our previous class for `Dog`):

```
Dog boi = new Dog("boi", 30, 6); // create a dog named "boi" with weight 30,  
                                // age 6  
Optional<Dog> optionalBoi = Optional.of(boi);  
optionalBoi.ifPresent(System.out::println);
```

This will give you the output:

boi is 6 years old and weighs 30 pounds

What happens if boi happens to be null? Try this:

```
boi = null;                                // boi is null!  
Optional<Dog> optionalBoi = Optional.of(boi); // potential problem here  
optionalBoi.ifPresent(System.out::println);
```

Run it and you'll see the error:

Exception in thread "main" java.lang.NullPointerException

If you are in a situation in which you might be creating an optional from a null object, then you have to take an extra precaution and use the `Optional.ofNullable()` method rather than `Optional.of()`. The `ofNullable()` method creates the optional if the object you pass in is not null; otherwise, it creates an empty optional:

```
boi = null;  
Optional<Dog> optionalBoi = Optional.ofNullable(boi); // check for null  
optionalBoi.ifPresent(System.out::println);  
if (!optionalBoi.isPresent()) System.out.println("Boi must be null");
```

You should see the output:

Boi must be null

You can also create empty optionals directly. Assume we have a `List` of `Dog` objects. We can create an empty `Optional<Dog>` and then assign it a value later

by streaming the List of Dogs, using `findFirst()` to find the first Dog:

```
Optional<Dog> emptyDog = Optional.empty(); // make an empty Dog optional
if (!emptyDog.isPresent()) {
    System.out.println("Empty dog must be empty");
}
emptyDog = dogs.stream().findFirst();      // find the first dog in
                                              // the list, assign it to emptyDog
emptyDog.ifPresent(d -> System.out.println("Empty dog is no longer empty"));
```

Run this code and you'll see

```
Empty dog must be empty
Empty dog is no longer empty
```

Another way to handle an empty optional is the `orElse()` method. With this method you get the value in the optional, or, if that optional is empty, you get the value you specify with the `orElse()` method:

```
Optional<Dog> emptyDog = Optional.empty();
Dog aDog = emptyDog.orElse(new Dog("Default Dog", 50, 10));
System.out.println("A Dog: " + aDog);
```

You will get the output:

```
A Dog: Default Dog is 10 years old and weighs 50 pounds
```

Try changing `emptyDog` to `optionalBoi`:

```
Dog boi = new Dog("boi", 30, 6);          // create a dog named "boi"
                                              // with weight 30, age 6
Optional<Dog> optionalBoi = Optional.of(boi); // not an empty optional
```

```

Dog aDog =                               // get boi, or if optionalBoi
                                         // is empty, get Default Dog.
                                         // optionalBoi.orElse(new Dog("Default Dog", 50, 10));
                                         System.out.println("A Dog: " + aDog);

```

And now you should see

A Dog: boi is 6 years old and weighs 30 pounds

because optionalBoi does, indeed, contain a Dog, boi.

[Table 9-4](#) summarizes the Optional methods.

**TABLE 9-4** Methods of the Optional Class

Class	Method	Returns...
Optional	empty()	Optional<T>
Optional	get()	T
Optional	ifPresent(Consumer<? super T> consumer)	void
Optional	isPresent()	boolean
Optional	of()	Optional<T>
Optional	ofNullable(T value)	Optional<T>
Optional	orElse(T other)	T
OptionalDouble, OptionalInt, OptionalLong	Each has similar methods to the above methods that return primitive optionals or primitives for each type	double, int, long

## CERTIFICATION OBJECTIVE

### Searching and Sorting with Streams (OCP Objectives 5.2 and 5.5)

5.2 *Search for data by using search methods of the Stream classes including findFirst, findAny, anyMatch, allMatch, noneMatch.*

5.5 *Sort a collection using Stream API.*

The `Stream` interface provides several methods for searching for elements in streams: `allMatch`, `anyMatch`, `noneMatch`, `findFirst`, and `findAny`. All of these operations are terminal operations; that is, they all return a single value, not a stream.

In addition, all of these operations are *short-circuiting* operations: that is, as soon as the result is determined, then the operation stops. So, for instance, if you are using `allMatch()` to determine whether each element passes a matching test, as soon as an element that doesn't pass the test is found, the operation stops, and the boolean result, `false`, is returned. All these operations can also be parallelized, so all can operate on streams efficiently.

### Searching to See Whether an Element Exists

The methods `allMatch()`, `anyMatch()`, and `noneMatch()` all take a `Predicate` to do a matching test and return a boolean. Let's go back to our trusty dogs, and try these methods on a stream of dogs. Here's the `Dog` class:

```
class Dog {  
    private String name;  
    private int age;  
    private int weight;  
  
    public Dog(String name, int weight, int age) {  
        this.name = name; this.weight = weight; this.age = age;  
    }  
    // getters and setters here  
    public String toString() {  
        return this.name + " is " + this.age + " years old and weighs "  
            + this.weight + " pounds";  
    }  
}
```

And here's a `List` of dogs:

```
List<Dog> dogs = new ArrayList<>();  
Dog boi = new Dog("boi", 30, 6);  
Dog clover = new Dog("clover", 35, 12);  
Dog aiko = new Dog("aiko", 50, 10);  
Dog zooey = new Dog("zooey", 45, 8);  
Dog charis = new Dog("charis", 120, 7);  
dogs.add(boi); dogs.add(clover); dogs.add(aiko);  
dogs.add(zooey); dogs.add(charis);
```

Now that we've got a `List` of dogs, we can stream it and perform stream pipeline operations on the dogs.

First, we'll look for dogs whose weight is greater than 50 pounds and whose

names start with "c" using `anyMatch()`. The `anyMatch()` method will stop searching as soon as it's found at least one dog whose name starts with "c":

```
boolean cNames =  
    dogs.stream() // stream the dogs  
        // keep dogs whose weight > 50  
        .filter(d -> d.getWeight() > 50)  
            // do any dog names start with c?  
            .anyMatch(d -> d.getName().startsWith("c"));  
System.out.println(  
    "Are there any dogs > 50 pounds whose name starts with 'c'? " + cNames);
```

You should see the output:

Are there any dogs > 50 pounds whose name starts with 'c'? true

The dog that matched must be "charis" because he weighs over 50 pounds.

Now let's use `allMatch()` to find whether all the dogs in the list have an age greater than 5. Here, we map the stream of dogs to a stream of their ages first and then use `allMatch()` to check each dog to make sure each has an age greater than 5795. `allMatch()` will stop and return false as soon as it finds any dog that fails this test.

```
boolean isOlder =  
    dogs.stream()  
        .mapToInt(d -> d.getAge()) // map from Dog to the Dog's age (integer)  
        .allMatch(a -> a > 5); // do all dogs have an age > 5?  
System.out.println("Are all the dogs age older than 5? " + isOlder);
```

All our dogs are older than 5, so you should see

Are all the dogs age older than 5? true

Finally, we'll use `noneMatch()` to make sure that none of the dogs in the

stream are named "red". First, we map the stream of dogs to a stream of the dog names, and then we use `noneMatch()` to check the name:

```
boolean notRed =  
    dogs.stream()  
        .map(d -> d.getName())           // map from Dog to Dog's name (String)  
        .noneMatch(n -> n.equals("red")); // are any of the dogs named "red"?  
    System.out.println("None of the dogs are red: " + notRed);
```

Since none of our dogs have the name "red," you should see

```
None of the dogs are red: true
```

## Searching to Find and Return an Object

All of these matching methods determine whether a match exists or not. If you want to actually get back a result of a match, then you can use `findFirst()` or `findAny()`. Neither of these methods takes an argument, so you need to filter first to narrow down the elements you might be searching for. Because `filter()` could potentially filter out all the elements of the stream, leaving you with an empty stream, you can guess that `findFirst()` and `findAny()` both return optionals in case they are called on empty streams; in which case, there is no valid result and the optional will be empty.

Let's use `findAny()` to find any `Dog` in our list of dogs that weighs more than 50 pounds and whose name begins with "c". You already know we have one of these dogs (we had a true result when we did the same test earlier with `anyMatch()`), but this time we want to actually get a `Dog` object back and use it.

```
Optional<Dog> c50 =  
    dogs.stream()                                // stream the dogs  
        .filter(d -> d.getWeight() > 50)          // keep dogs with weight > 50  
        .filter(d -> d.getName().startsWith("c")) // keep dogs with name "c"  
        .findAny();                                // pick any dog from the  
                                                // stream  
                                                // and return it  
  
c50.ifPresent(System.out::println);
```

The output is

```
charis is 7 years old and weighs 120 pounds
```

In this example, only one dog passed the tests in both filters; only “charis” weighs more than 50 pounds and has a name beginning with “c”. So `findAny()` returns the one dog left in the stream when we call that operation.

But what if there’s more than one dog in the stream when we call `findAny()`? For instance, if we look for any dog whose age is older than 5, all the dogs will pass that filter test, so all the dogs will still be in the stream when we call `findAny()`:

```
Optional<Dog> d5 =  
    dogs.stream()  
        .filter(d -> d.getAge() > 5)  
        .findAny();  
  
d5.ifPresent(System.out::println);
```

You’ll probably see the output:

```
boi is 6 years old and weighs 30 pounds
```

Of course, boi happens to be the first dog in the stream, but technically `findAny()` could return *any* of the dogs in the stream. There’s no guarantee it will be the first one (particularly when you parallelize the stream, which we’ll

get to later on).



***Don't forget that all these matching and finding methods are short-circuiting. That means that as soon as `findAny()` finds a dog that matches, everything stops.***

***Let's add a `peek()` at just the right spot to see what's happening in the stream when we run this code. Give this a try:***

```
Optional<Dog> d5 =  
    dogs.stream()  
        .filter(d -> d.getAge() > 5).peek(System.out::println)  
        .findAny();  
  
d5.ifPresent(System.out::println);
```

***Remember that `peek()` allows you to “peek” into the stream. It takes a consumer and returns exactly the same stream as you call it on, so it makes no changes to anything. It's simply a window to what's flowing by on the stream.***

***Run this and you'll probably see***

```
boi is 6 years old and weighs 30 pounds // from the peek  
boi is 6 years old and weighs 30 pounds // from the final print
```

***So if you're asked on the exam what output you'll see with a `peek()`, make sure you understand how this works. You might be tempted to choose an answer that shows all the dogs printed with the `peek` and then `boi` for the final print, but that wouldn't be correct.***

## Sorting

We often want to sort things stored in data structures, so it makes sense we might want to sort elements flowing through a stream pipeline, too. The Stream

interface includes a `sorted()` method that you can use to sort a stream of elements by natural order or by providing a `Comparator` to determine the order.

Sorting by natural order is easy: just add the `sorted()` operator into the stream pipeline, like this:

```
Stream.of("Jerry", "George", "Kramer", "Elaine")
    .sorted()
    .forEach(System.out::println);
```

If you run this code, you'll see the output:

```
Elaine
George
Jerry
Kramer
```

What if you want to sort more complex objects, like `Duck` objects? Here's a `Duck`:

```
class Duck implements Comparable<Duck> {
    String name;
    String color;
    int age;

    public Duck(String name, String color, int age) {
        this.name = name; this.color = color; this.age = age;
    }
    // getters and setters here...
    public String toString() {
        return (getName() + " is " + getColor() + " and is "
               + getAge() + " years old.");
    }
    @Override                      // describe how to sort Ducks
    public int compareTo(Duck duck) {
        return this.getName().compareTo(duck.getName());
    }
}
```

We've gone ahead and made the Duck a Comparable and implemented the compareTo() method that sorts by name so we can sort these Ducks:

```
List<Duck> ducks = Arrays.asList(      // create a List of Ducks
    new Duck("Jerry", "yellow", 3),
    new Duck("George", "brown", 4),
    new Duck("Kramer", "mottled", 6),
    new Duck("Elaine", "white", 2)
);
ducks.stream()
    .sorted()                      // sort ducks by name
    .forEach(System.out::println);   // print them
```

Run this code and you should see

```
Elaine is white and is 2 years old.
George is brown and is 4 years old.
Jerry is yellow and is 3 years old.
Kramer is mottled and is 6 years old.
```

The `compareTo()` method we provided in the `Duck` class sorts the ducks by their names.

What if you want to change the sort when you stream the ducks? You can use `sorted()` with a `Comparator`:

```
ducks.stream()
    .sorted((d1, d2) -> d1.getAge() - d2.getAge()) // sort ducks by age
    .forEach(System.out::println);
```

Here, you're providing a `Comparator` to `sorted()` that it will use to sort the Ducks (instead of the `compareTo()` method in the `Duck` class). The result is

Elaine is white and is 2 years old.  
Jerry is yellow and is 3 years old.  
George is brown and is 4 years old.  
Kramer is mottled and is 6 years old.

Now the ducks are sorted by age.

Comparators are handy because they can be defined separately from the class, like we did to sort the ducks by age rather than by name (which is the default sort order, defined by the Comparable).

Above, we passed the Comparator directly to the sorted() method using a lambda expression. If we wanted to, we could write out the Comparator separately, assign it to a variable, and then pass that variable to sorted(), like this:

```
Comparator<Duck> byAgeLambda = (d1, d2) -> d1.getAge() - d2.getAge();  
ducks.stream()  
    .sorted(byAgeLambda)           // pass the Comparator to sorted  
    .forEach(System.out::println);
```

You've seen us use comparators like this before (in the previous chapter). Comparator also has some handy static methods that you're likely to see when defining comparators for use with streams: comparing(), reversed(), and thenComparing().

Let's write three Comparators to sort ducks by their color, by their name, and by their age, using the Comparator.comparing() method. The comparing() method takes a Function whose functional method expects a property to sort by, like Duck.age, as an argument and returns a Comparator that compares objects by that property.

```
Comparator<Duck> byColor = Comparator.comparing(Duck::getColor);  
Comparator<Duck> byName = Comparator.comparing(Duck::getName);  
Comparator<Duck> byAge = Comparator.comparing(Duck::getAge);
```

Here, we're using a method reference to specify each Function that comparing() will use to get the properties of the Duck we want to compare by

for each comparator. Now that we have these comparators, we can use them to sort ducks in various combinations.

We can sort by age:

```
ducks.stream().sorted(byAge).forEach(System.out::println);
```

Or sort by age reversed:

```
ducks.stream().sorted(byAge.reversed()).forEach(System.out::println);
```

Note that `reversed()` is a method of the `Comparator`, that returns a new `Comparator` that has the reverse ordering.

We can also sort by one property and then by another property, using the `thenComparing()` method. `thenComparing()` returns a `Comparator` that compares first by the `Comparator` you call it on and then by the `Comparator` you pass in. To sort by name, and then by age, we'll write:

```
byName.thenComparing(byAge)
```

Before we do that, let's add a few more ducks:

```
List<Duck> ducks = Arrays.asList(  
    new Duck("Jerry", "yellow", 3),  
    new Duck("George", "brown", 4),  
    new Duck("Kramer", "mottled", 6),  
    new Duck("Elaine", "white", 2),  
    new Duck("Jerry", "mottled", 10),  
    new Duck("George", "white", 12),  
    new Duck("Kramer", "brown", 11),  
    new Duck("Elaine", "brown", 13)  
);
```

We have our original four ducks, as well as four new ducks with the same names but different colors and all much older. (Think of these as the “bizarro” ducks.)

Now let's sort the ducks, first by name and then by age:

```
ducks.stream()
    .sorted(byName.thenComparing(byAge))
    .foreach(System.out::println);
```

The result is:

```
Elaine is white and is 2 years old.
Elaine is brown and is 13 years old.
George is brown and is 4 years old.
George is white and is 12 years old.
Jerry is yellow and is 3 years old.
Jerry is mottled and is 10 years old.
Kramer is mottled and is 6 years old.
Kramer is brown and is 11 years old.
```

And, of course, we could sort by color and then by age:

```
ducks.stream()
    .sorted(byColor.thenComparing(byAge))
    .foreach(System.out::println);
```

Or a variety of other combinations.

Finally, another method of `Stream` that comes in handy when sorting streams is `distinct()`. This method returns a stream with distinct elements, so if an element is repeated in the stream, you'll end up with only one of them.

Let's say, for instance, that you want to see how many different colors your ducks are. You only need to see the color once, even if you have multiple ducks with the same color.

Here's a revised List of Ducks:

```
List<Duck> ducks = Arrays.asList(  
    new Duck("Jerry", "yellow", 3),  
    new Duck("George", "brown", 4),  
    new Duck("Kramer", "mottled", 6),  
    new Duck("Elaine", "white", 2),  
    new Duck("Huey", "mottled", 2),  
    new Duck("Louie", "white", 4),  
    new Duck("Dewey", "brown", 6)  
);
```

You can see that we have 1 yellow, 2 brown, 2 mottled, and 2 white ducks.

To get the list of distinct colors of ducks, we'll first map each Duck to its color, then use `distinct()` to make sure we get a stream of distinct color strings, and then print those:

```
ducks.stream()  
    .map(d -> d.getColor())          // get the duck colors  
    .distinct()                      // make sure there are no repeats!  
    .forEach(System.out::println);   // print the colors
```

And we see

```
yellow  
brown  
mottled  
white
```



*We slipped some method references into this section: `Duck::getColor`, `Duck::getName`, `Duck::getAge`, and `System.out::println`. Recall from*

*the previous chapter when we described how `System.out::println` is a method reference: a shorthand for a lambda expression. For example, we replaced a lambda expression like this:*

```
Stream.of("Elain", "Jerry", "George", "Kramer")
    .forEach(n -> System.out.println(n)); // uses a lambda expression
```

*with a method reference like this:*

```
Stream.of("Elain", "Jerry", "George", "Kramer")
    .forEach(System.out::println); // uses a method reference
```

*Method references simply replace lambda expressions that do nothing except call another function. For this instance method reference, `System.out` is the class and `println` is the instance method. We use the `:::` syntax to indicate the method reference.*

*Take another look at `Duck::getColor`, `Duck::getName`, and `Duck::getAge`. These, too, are method references to instance methods. They are a type of method reference that refers to a nonstatic method of an instance of a type. If we wrote those method references out as lambda expressions, they'd look like this:*

```
Comparator<Duck> byColor = Comparator.comparing(d -> d.getColor());
Comparator<Duck> byName = Comparator.comparing(d -> d.getName());
Comparator<Duck> byAge = Comparator.comparing(d -> d.getAge());
```

*So, for instance, the lambda `d -> d.getColor()` is a Function whose functional method takes an object of type `Duck` and returns an object of type `String`, mapping a `Duck` to his or her color.*

*The code*

```
Comparator<Duck> byColor = Comparator.comparing(Duck::getColor);
```

*uses an instance method reference to replace the lambda, where the instance we're using is an instance of `Duck` and the method is `getColor()`. Remember, method references are shorthand for lambdas that just turn around and call another method.*

## Methods to Search and Sort Streams

The searching and sorting methods on streams are straightforward; the main thing is to get the types correct! [Table 9-5](#) details these methods.

**TABLE 9-5** Searching and Sorting on Streams

Interface	Method	Description
<b>Stream</b>	<code>allMatch(Predicate&lt;? super T&gt; predicate)</code>	Returns a boolean; true if all elements in the stream pass the Predicate test. Stops when any element fails the test.
<b>Stream</b>	<code>anyMatch(Predicate&lt;? super T&gt; predicate)</code>	Returns a boolean; true if any of the elements in the stream pass the Predicate test. Stops when any element passes the test.
<b>Stream</b>	<code>noneMatch(Predicate&lt;? super T&gt; predicate)</code>	Returns a boolean; true if none of the elements in the stream pass the Predicate test. Stops when any element passes the test.
<b>Stream</b>	<code>findFirst()</code>	Returns an <code>Optional&lt;T&gt;</code> with the first element from the stream or an empty optional if the stream is empty.
<b>Stream</b>	<code>findAny()</code>	Returns an <code>Optional&lt;T&gt;</code> with an element from the stream (no guarantees about which one!) or an empty optional if the stream is empty.
<b>DoubleStream,</b> <b>IntStream,</b> <b>LongStream</b>	<code>allMatch(DoublePredicate predicate),</code>  <code>allMatch(IntPredicate predicate),</code>  <code>allMatch(LongPredicate predicate)</code>	Returns a boolean; true if all elements in the stream pass the Predicate test. Stops when any element fails the test.
<b>DoubleStream,</b> <b>IntStream,</b> <b>LongStream</b>	<code>anyMatch(DoublePredicate predicate),</code>  <code>anyMatch(IntPredicate predicate),</code>  <code>anyMatch(LongPredicate predicate)</code>	Returns a boolean; true if any of the elements in the stream pass the Predicate test. Stops when any element passes the test.



Interface	Method	Description
<code>DoubleStream</code> , <code>IntStream</code> , <code>LongStream</code>	<code>noneMatch(DoublePredicate predicate),</code>  <code>noneMatch(IntPredicate predicate),</code>  <code>noneMatch(LongPredicate predicate)</code>	Returns a boolean; true if none of the elements in the stream pass the Predicate test. Stops when any element passes the test.
<code>DoubleStream</code>	<code>findFirst()</code>	Returns an <code>OptionalDouble</code> : the first element in the stream or an empty optional if the stream is empty.
<code>IntStream</code>	<code>findFirst()</code>	Returns an <code>OptionalInt</code> : the first element in the stream or an empty optional if the stream is empty.
<code>LongStream</code>	<code>findFirst()</code>	Returns an <code>OptionalLong</code> : the first element in the stream or an empty optional if the stream is empty.
<code>DoubleStream</code>	<code>findAny()</code>	Returns an <code>OptionalDouble</code> : an element in the stream or an empty optional if the stream is empty.
<code>IntStream</code>	<code>findAny()</code>	Returns an <code>OptionalInt</code> : an element in the stream or an empty optional if the stream is empty.
<code>LongStream</code>	<code>findAny()</code>	Returns an <code>OptionalDouble</code> : an element in the stream or an empty optional if the stream is empty.
<code>Stream</code> , <code>DoubleStream</code> , <code>IntStream</code> , <code>LongStream</code>	<code>sorted()</code>	Returns a <code>Stream&lt;T&gt;</code> (or primitive stream type). Sort elements of a stream by natural order
<code>Stream</code>	<code>sorted(Comparator&lt;? super T&gt; comparator)</code>	Returns a <code>Stream&lt;T&gt;</code> . Sort elements of a stream by the provided Comparator.
<code>Stream</code> , <code>DoubleStream</code> , <code>IntStream</code> , <code>LongStream</code>	<code>distinct()</code>	Returns a <code>Stream&lt;T&gt;</code> (or primitive stream type) with distinct elements. For <code>Stream&lt;T&gt;</code> , the <code>Object.equals(Object)</code> test is used.

## Don't Modify the Source of a Stream

You might have heard that when you're using Java streams, you are programming in a “functional” style. Aside from being a more declarative way of writing code (thus the stream pipeline versus several statements to specify operations), the key tenet of functional programming is to avoid *side effects*—that is, changing state stored in variables and fields—during computation. Instead, you process data using operations that produce new values (rather than changing old ones). Functional programming style makes it easier to prove the correctness of your program and to replicate results, and avoiding state changes makes it easier to implement optimizations in the compiled code. Although Java is far from being a functional programming language, streams are a step toward a functional style of programming that encourages this idea of avoiding state changes on the original data structure being processed via a stream.

At this point in the chapter, we're going to share something really important about streams. That is, you should never, ever try to modify the source of a stream from within the stream pipeline.

You might be tempted at times, but don't. It just isn't done. Java won't give you a compile-time error if you try and may not even give you a runtime error, although your results will not be guaranteed. But just don't do it. Ever.

One reason you might be tempted, say, is if you're looking for elements that pass a test and you want to modify that element in the source, or perhaps even delete that element from the source. For example, you might want to remove all dogs who weigh less than 50 pounds from the list of dogs:

```
dogs.stream().filter(d -> {
    if (d.getWeight() < 50) {
        dogs.remove(d);
        return false;
    }
    return true;
}).forEach(System.out::println);
```

You might *think* that you'll see a list of the dogs whose weights are greater

than or equal to 50, and you might *think* the dogs list will now contain only dogs whose weights are greater than or equal to 50, but what you'll probably get instead is something like this:

```
aiko is 10 years old and weighs 50 pounds  
Exception in thread "main" java.lang.NullPointerException
```

So what do you do if you want to get a list of all dogs whose weight is greater than or equal to 50 after you've done the stream computation?

The best way to do this is to collect the dogs at the end of the stream pipeline into a new collection. And, of course, Java provides an easy way for you to do just that with the `Stream.collect()` method.

Let's collect all dogs who weigh 50 pounds or more. Remember, here are our dogs:

```
List<Dog> dogs = new ArrayList<>();  
Dog boi = new Dog("boi", 30, 6);  
Dog clover = new Dog("clover", 35, 12);  
Dog aiko = new Dog("aiko", 50, 10);  
Dog zooey = new Dog("zooey", 45, 8);  
Dog charis = new Dog("charis", 120, 7);  
dogs.add(boi); dogs.add(clover); dogs.add(aiko);  
dogs.add(zooey); dogs.add(charis);
```

Looking at that list, we'd expect to end up with two dogs: Aiko who weighs 50 pounds and Charis who weighs 120 pounds (that's a big dog!).

Here's how we can use a stream pipeline to filter all dogs who weigh 50 pounds or more and collect them into a new `List`, `heavyDogs`:

```
List<Dog> heavyDogs =  
    dogs.stream() // stream the dogs  
        .filter(d -> d.getWeight() >= 50) // filter only dogs >= 50 pounds  
        .collect(Collectors.toList()); // collect the dogs into a new List
```

We can then print this list:

```
heavyDogs.forEach(System.out::println);
```

And see the output:

```
aiko is 10 years old and weighs 50 pounds  
charis is 7 years old and weighs 120 pounds
```

## CERTIFICATION OBJECTIVE

### Collecting Values from Streams (OCP Objectives 3.8, 5.6, and 9.3)

3.8 *Use method references with Streams.*

5.6 *ave results to a collection using the collect method and group/partition data using the Collectors class.*

9.3 *Use Stream API with NIO.2.*

The `collect()` method is a reduction operation: it reduces a stream into a collection of objects or a value. The `Collector` you pass to the method specifies how to reduce the stream, say, into a `List` or a `Set`. The `Collectors` class provides implementations of `Collector` that each determine how you collect, everything from methods to make a simple list, like `toList()`, to methods that allow you to group items in your `Collection`, resulting in a map so values are organized by a key value.

As you collect, you can

- Group values together based on a `Function` (returns a `Map`)
- Partition values into true/false partitions based on a `Predicate` (returns a `Map`)
- Map values into other values using a `Function` (returns a `Collector`)
- Join values into a `String`
- Count values as you collect

**Note that *collectors* (with an s) is a helper class that contains implementations of the *Collector* interface. Don't get the two mixed up! This is similar to *Collections/Collection*.**

Let's make a Person class and a nice collection of people with names and ages, so we have some data to work with. Then we'll stream the Collection of people and use the various Collectors to process the data in a few different ways.

Here's the Person class:

```
class Person {  
    public String name;  
    public Integer age;  
  
    public Person(String name, Integer age) {  
        this.name = name; this.age = age;  
    }  
    public String getName() { return this.name; }  
    public Integer getAge() { return this.age; }  
    public String toString() {  
        return this.name + " is " + this.age + " years old";  
    }  
}
```

Now, we'll make a bunch of Persons and add them to a List:

```
Person beth = new Person("Beth", 30);
Person eric = new Person("Eric", 31);
Person deb = new Person("Deb", 31);

Person liz = new Person("Liz", 30);
Person wendi = new Person("Wendi", 34);
Person kathy = new Person("Kathy", 35);
Person bert = new Person("Bert", 32);
Person bill = new Person("Bill", 34);
Person robert = new Person("Robert", 38);
```

```
List<Person> people = new ArrayList<Person>();
people.add(beth); people.add(eric); people.add(deb);
people.add(liz); people.add(wendi); people.add(kathy);
people.add(bert); people.add(bill); people.add(robert);
```

Now that we have a great collection of people with names and ages, we can stream people and use collectors to organize and process our collection. Let's begin with the simplest way to collect people: we'll collect everyone whose age is 34 into a List:

```
List<Person> peopleAge34 =
    people.stream()                                // stream the people
        .filter(p -> p.getAge() == 34)            // find people age 34
        .collect(Collectors.toList());             // collect 34s into a new List
System.out.println("People aged 34: "           // print 34s
    + peopleAge34);
```

Here, we're streaming the List of Persons, using filter() with a Predicate to test to see whether the Person's age is 34, and collecting those Persons who pass

that test into a new List.

When we print that List, we see

```
People aged 34: [Wendi is 34 years old, Bill is 34 years old]
```

There's no guarantee what kind of List you get using `Collectors.toList()`. If you specifically want an `ArrayList`, you can use the `toCollection()` method instead, like this:

```
List<Person> peopleAge34 =  
    people.stream() // stream the people  
        .filter(p -> p.getAge() == 34) // find people age 34  
        .collect(Collectors.toCollection(ArrayList::new)); // make an ArrayList
```

And get the same output.



***Notice here that we're passing a method reference to the `Collectors.toCollection()` method. Looking at the documentation for `Collectors.toCollection()`, we see that this method takes a Supplier whose functional method must return a new empty collection of the "appropriate type." We could write the code like this:***

```
ArrayList<Dog> heavyDogs =  
    dogs.stream()  
        .filter(d -> d.getWeight() >= 50)  
        .collect(Collectors.toCollection(() -> new ArrayList<Dog>()));
```

***providing the Supplier `() -> new ArrayList<Dog>()` as the argument to `Collectors.toCollection()`. Recall that a method reference is shorthand for a lambda expression that just turns around and calls another method. So by writing `ArrayList::new` instead of the lambda, we are saying the same thing as the lambda expression above: that is, just call the constructor (with new) of an `ArrayList<Dog>` to create a new `ArrayList`.***

*This is a slightly different kind of method reference than you saw earlier in this chapter, when we used instance method references to refer to instance methods, like a Duck's `getColor()` method with `Duck::getColor`. This form of method reference is the constructor method reference, and it's just a shorthand for a lambda that creates a new instance of a class.*

## Using `collect()` with `Files.lines()`

Think back to how we handled getting DVDs from a file using `Files.lines()` earlier (in the section “Create a Stream from a File”). In that example, we streamed `String` data about DVDs from the file and added DVDs one at a time to a `List` in a `forEach()` consumer:

```
List<DVDInfo> dvds = new ArrayList<DVDInfo>();
try (Stream<String> stream = Files.lines(Paths.get(filename))) {
    stream.forEach(line -> {
        String[] dvdItems = line.split("/");
        DVDInfo dvd = new DVDInfo(dvdItems[0], dvdItems[1], dvdItems[2]);
        dvds.add(dvd); // need a better way to do this!
    });
}
```

Now that you know how to use `collect()`, you can probably think of a better way to do this, right? While the solution above works, it’s not the recommended way to collect items from a stream into a `List` because you are modifying an object that’s defined outside the stream pipeline—in other words, your stream pipeline has a *side effect*. Using `collect()` is the preferred way to do this, and, as you’ll see, along with avoiding an unnecessary side effect, it also makes your code clearer.

Here’s another quick example of using `Files.lines()` to stream data from a file, and this time we’ll add that data to a `List` using `collect()`. Imagine we have a file, “names.txt,” with the following names in it:

Jerry  
George  
Kramer  
Elaine  
Huey  
Louie  
Dewey

Using `collect()`, we can easily add these names to a `List`, like this:

```
String filename = "names.txt";
try (Stream<String> stream = Files.lines(Paths.get(filename))) {
    List<String> data = stream.collect(Collectors.toList()); // collect names
    data.forEach(System.out::println); // print names
} catch(IOException e) {
    System.out.println(e);
}
```

### EXERCISE 9-1

---

## Collecting Items in a List

Try rewriting the earlier DVDs example to use `collect()` instead of adding DVDs manually to the `dvds` `List` in the `forEach()` loop. Just as with the simple names example, you can use `Collectors.toList()` to add DVDs to the `dvds` `List`. Here's a hint: you'll need to map the lines (`Strings`) you're streaming from the “`dvdinfo.txt`” file to `DVDInfo` objects before you collect them.

Using `collect()` is a much better, and preferred way, to collect streamed items into a `List`.

---

## Grouping and Partitioning

Now what if we want to group people by age? We can use the `Collectors.groupingBy()` method. We specify *how* to group `Person` objects by passing a `Function` to the `groupingBy()` method, which returns a `Collector` that will collect data elements from the stream and group them in a `Map` by a key, according to that `Function`.

You can think of the `Function` as a “classification function.” If we want to group people by age, then we need to pass a `Function` to `groupingBy()` that maps a `Person` to their age:

```
Map<Integer, List<Person>> peopleByAge =  
    people.stream()  
        .collect(Collectors.groupingBy(Person::getAge));  
    System.out.println("People by age: " + peopleByAge);
```

When we print the resulting `Map`, we see

```
People by age: {32=[Bert is 32 years old], 34=[Wendi is 34 years old, Bill  
is 34 years old], 35=[Kathy is 35 years old], 38=[Robert is 38 years old],  
30=[Beth is 30 years old, Liz is 30 years old], 31=[Eric is 31 years old, Deb  
is 31 years old]}
```

Notice that the `Map` you get back uses an age (`Integer`) as a key and the value is a `List` of the same type of object in the stream; in this case, that’s `Person`. Also notice that the way `System.out.println()` shows us a `Person` in the output is by using the `toString()` method, but the values in the `List` associated with each are not `Strings`; they are `Persons` (you can see from the type parameter on `List<Person>` that that is the case).

So now we have a map with `Integers` (ages) for keys and `List<Person>s` for values, and we can see that, for instance, Bert is 32 and Wendi and Bill are 34, and so on.

The `groupingBy()` method is heavily overloaded with a variety of options for grouping values in a `Collection`. For instance, you can use a version of `groupingBy()` that takes a classification `Function` as a first argument and a `Collector` as a second argument. That `Collector` argument allows you to

reduce the values of the Map you get as the result of the `groupingBy()` method, thus reducing each value List into another value. Yeah, we know, that's tricky to understand, so let's look at an example.

Here, we're going to group people by age, but rather than create a List of the people associated with a certain age in a Map, as we did above, now we're going to count the number of people in the List associated with a given age and use that value in the resulting Map instead. So we have two reductions going on here: we have a `groupingBy()` reduction to group people by age and then we have a `counting()` reduction to count the people in the List associated with a particular age:

```
Map<Integer, Long> numPeopleWithAge =  
    people.stream()  
        .collect(Collectors.groupingBy( // we're going to group by...  
            Person::getAge,           // ... age  
            Collectors.counting())); // and count rather than List  
    System.out.println("People by age: " + numPeopleWithAge);
```

The result of this code is

```
People by age: {32=1, 34=2, 35=1, 38=1, 30=2, 31=2}
```

So now we can see we have 2 people who are 30, 2 people who are 34, 2 people who are 31, and 1 of each other age. To count the people in each list, we use the `Collectors.counting()` method, which returns a Collector that simply counts the number of input elements. The result of calling `collect()` on this Collector is a Map from ages to the number of people of a given age.

What if we want to group people by age, but list only their name rather than the entire Person object in the Map? That is, we want a map of ages and names (`Map<Integer, List<String>>`) rather than a map of ages and Person objects (`Map<Integer, List<Person>>`)?

We can do that by passing a `Collectors.mapping()` Collector as that second argument to `groupingBy()`:

```

Map<Integer, List<String>> namesByAge =
    people.stream()
        .collect(
            Collectors.groupingBy(
                Person::getAge,           // group by age
                Collectors.mapping(      // map from Person to...
                    Person::getName,     // .. name
                    Collectors.toList()  // collect names in a list
                )
            )
        );
System.out.println("People by age: " + namesByAge);

```

Here, we're streaming the people and calling the `collect()` method on the stream, passing in the `groupingBy` collector. This `groupingBy()` collector takes two arguments, the `Function` that determines how we're going to group (by age) and another `Collector` that tells us how to reduce the values in the `Map` associated with each age. Remember that for the simplest version of `groupingBy()`, each value is just a `List` of `Persons` who are that age. But now we're using the `mapping()` collector to further reduce or modify that `List` of `Persons`.

The `mapping()` method maps each `Person` to another value. What we'd like to do is map a `Person` to their name. Taking a look at the `mapping()` method, we see that its first argument is a `Function` whose functional method takes an object of the type we're mapping from (a `Person`) and returns another object, in this case, the `Person`'s name, a `String`. The second argument to `mapping()` is a `Collector` that tells us what to do with the potentially multiple values we're mapping. Remember that we can have multiple people of the same age, so we're mapping a `List` of `Persons` to a `List` of their names. That second argument specifies how we'd like to collect those names. We'll use `toList()` to keep it easy, but you could also choose `toCollection()` to create an `ArrayList` or some other collection for the names.

The output we see is

```
People by age: {32=[Bert], 34=[Wendi, Bill], 35=[Kathy], 38=[Robert],  
30=[Beth, Liz], 31=[Eric, Deb]}
```

So now our Map maps age keys to Lists of String names.

Partitioning as you collect is essentially a more specialized kind of groupingBy(). The partitioningBy() method organizes the results into a Map like groupingBy() does, but partitioningBy() takes a Predicate rather than a Function, so the results are split into two groups (partitions) based on whether the items in the stream pass the test in the Predicate. Let's partition our results by the test: is the person older than 34?

```
Map<Boolean, List<Person>> peopleOlderThan34 =  
    people.stream()  
        .collect(  
            Collectors.partitioningBy(p -> p.getAge() > 34));  
    System.out.println("People > 34: " + peopleOlderThan34);
```

As you might expect, the result is a Map that maps booleans to Persons, so all people 34 or younger will be mapped to the key false, and all people older than 34 will be mapped to the key true:

```
People > 34: {false=[Beth is 30 years old, Eric is 31 years old, Deb is 31  
years old, Liz is 30 years old, Wendi is 34 years old, Bert is 32 years old,  
Bill is 34 years old], true=[Kathy is 35 years old, Robert is 38 years old]}
```

Perhaps you can see how powerful streams and collectors are together? Although this code may not feel intuitive to you, it certainly is more concise than writing the equivalent code without using streams to create these Maps, and you avoid having to create some intermediate data structures yourself. The whole idea is that Java can optimize the code used to create the resulting Maps behind the scenes so these types of operations become a lot more efficient.



We slipped another example of an instance method reference by you in this section: `Person::getAge`. We did this to group people by age. `groupingBy()` takes a Function that maps a Person object to another value to use that value as the mapping key—in this case, telling `groupingBy()` to map people by age.

The method reference `Person::getAge` is an instance method reference that refers to a nonstatic method of an instance of a Person. If we wrote that method reference out as a lambda expression, it would look like this:

```
(p) -> p.getAge()
```

So this lambda is a Function whose functional method takes an object of type Person and returns an object of type Integer, mapping a Person to their age.

## Summing and Averaging

A couple of other useful collectors are `summingInt()` and `averagingInt()` (and their long and double counterparts). As you might guess, these need to work on numbers, so both take a `ToIntFunction` (that is a Function that maps an object to an int). To experiment with these, we'll need to add a few more people to our list of people. We're going to add people with the same names so we can group by name and then find the sum of ages and the average of ages. You'll see what we mean in just a moment. Let's first add three more people to our list:

```
Person bill2 = new Person("Bill", 40);
Person beth2 = new Person("Beth", 45);
Person bert2 = new Person("Bert", 38);
people.add(bill2); people.add(beth2); people.add(bert2);
```

Now we have two people named “Bert” in our list; the first Bert (from way back) is age 32, and the second Bert (bert2) is age 38. Likewise, for “Beth” and “Bill,” we now have two people with the name “Beth” and two people with the name “Bill,” each of different ages.

First, we want to sum the ages of the two Berts, the two Beths, and the two Bills, and group by name. Perhaps we're computing person-years of life

experience of all people whose names begin with “B.” (This scenario is completely contrived, of course, but if you were counting occurrences of Scrabble words, then this would make a lot more sense.)

Here’s how we get the sum of the ages of people whose names begin with “B” and group by name:

```
Map<String, Integer> sumOfBAge =  
    people.stream() // stream people  
        .filter(p -> p.getName().startsWith("B")) // filter "B" names  
        .collect( // collect  
            Collectors.groupingBy( // groupBy  
                Person::getName, // ... name  
                Collectors.summingInt(Person::getAge) // and sum of ages  
            )  
        );  
    System.out.println("People by sum of age: " + sumOfBAge);
```

The output of this code is

```
People by sum of age: {Bill=74, Beth=75, Bert=70}
```

The code took the ages of the two Bills and added them together. It does the same thing with the two Beths and the two Berts, and groups the results by name in a Map. So the total age of all the Bills is 74, all the Beths is 75, and all the Berts is 70.

What about the average age of the people whose names begin with “B”? Yep, that’s what averagingInt() is for:

```

Map<String, Double> avgOfBAge = // note we need Double not Integer
                                // for the values!
    people.stream()
        .filter(p -> p.getName().startsWith("B"))
        .collect(
            Collectors.groupingBy(
                Person::getName,
                // now average ages instead of sum of ages
                Collectors.averagingInt(Person::getAge)
            )
        );
    System.out.println("People by avg of age: " + avgOfBAge);

```

This code produces the output:

```
People by avg of age: {Bill=37.0, Beth=37.5, Bert=35.0}
```

Now, we've computed the average age of all the Bills, the average age of all the Beths, and the average age of all the Berts, grouped them by name, and collected the results in a Map.

It's important to note here that the `averagingInt()` Collector reduces the ages to a `Double`, so the type signature on this Map is `Map<String, Double>` (compare with the type signature on the result of the `summingInt()` Collector example, which is `Map<String, Integer>`).

In both these examples, we are first collecting `Person` objects, grouping people by name, and then doing a second reduction on the `Person` objects that are the values of the first (implicit) Map created by the grouping. In the case of `summingInt()`, we are reducing with a Collector that sums the ages of all the `Person` objects associated with a given name; in the case of `averagingInt()`, we are reducing with a collector that averages the ages of all the `Person` objects associated with a given name. Sometimes these second reductions on the value portions of the first reduction are called “downstream processing” because they work on the results of the first reduction on a stream.

## Counting, joining, maxBy, and minBy

So far we've looked at collectors that, when used with the `collect()` method, reduce a stream to a `Collection` type, like a `List` or a `Map`. There are a few other collectors that work with `collect()` to reduce a stream to a single value, such as a `String`, instead.

We used the `Collectors.counting()` method above to create a `Collector` that counts elements being collected. A simple example of using `counting()` is

```
Long n = people.stream().collect(Collectors.counting());  
System.out.println("Count: " + n);
```

This simply counts the items in the `people` stream. You already know how to do this with the `Stream` method `count()`, and there's no reason to create a `Collector` here to count elements in a stream, so we provide this example just to show how it works at the most basic level. More typically, you'll find `Collectors.counting()` used like we did above, as part of a `groupingBy()` operation.

The `Collectors.joining()` method returns a `Collector` that takes stream elements and concatenates them into a `String` by order in which they are encountered (which may or may not be the order of the original `Collection` you're streaming!).

For instance, we can get the name of every `Person` who's older than 34 and join those names together into one `String` like this:

```
String older34 =  
    people.stream() // stream people  
        .filter(p -> p.getAge() > 34) // filter for older than 34  
        .map(Person::getName) // map Person to name  
        .collect(Collectors.joining(", ")); // join names into one string  
  
System.out.println("Names of people older than 34: " + older34);
```

Here, we're streaming the `people` `List` (the original list, without the duplicate names), filtering to get only people older than 34, mapping those people to their names (so at that point in the pipeline, we have a stream of `String` names), and then collecting those names into one `String`, with each name separated by a

comma:

Names of people older than 34: Kathy, Robert

The joining() method requires as input an object that implements the CharSequence interface and includes String, which is what we're using here.

The methods maxBy() and minBy() do what you expect: they collect (reduce) to the max and min of the input elements, respectively. Let's use maxBy() to find the oldest person in the people stream:

```
Optional<Person> oldest =  
    people.stream()  
        .collect(Collectors.maxBy((p1, p2) -> p1.getAge() - p2.getAge()));  
    oldest.ifPresent(p -> System.out.println("Oldest person: " + p));
```

The maxBy() method takes a Comparator (as does minBy()) to compare the elements from the stream as they are being collected and returns a collector that, when used with the Stream.collect() method, will reduce the stream to the “max” of the elements in the stream as measured by that Comparator.

Notice that maxBy() returns an Optional<Person>, not a Person, because when finding the max as you are collecting a stream, if that stream is empty, there will be no value. So we use the ifPresent() method of the optional to make sure there is a value there before we try to print it out.

The advantage to finding the max of our Person stream using collect() with the maxBy() collector, rather than just using the max() terminal operation on the stream, like this:

```
people.stream().mapToInt(p -> p.getAge()).max();
```

is that our result is a Person object. Remember, IntStream.max() works only on numbers, so when we mapped our Person stream to a stream of Integers in order to use max() to find the oldest person, we got a number back. By using maxBy() with collect(), we can find the Person object who has the highest age and get the whole Person back, not just their age. (We could also use the Stream.max() method and provide a comparator to do essentially the same thing.)

## Stream Methods to Collect and Their Collectors

Using streams with collectors can get fairly complex, especially when you have multiple downstream operations. On the exam, you'll likely see only one or two levels of nesting when using collectors (e.g., a `groupingBy()` used with a `mapping()`) but be sure to get lots of practice using the `Stream collect()` method with collectors of various kinds.

**TABLE 9-6** Using Streams with Collectors

Interface	Method	Description
<b>Stream</b>	<code>collect(Collector&lt;? super T, A, R&gt; collector)</code>	Reduces a stream to an instance of a mutable type, like a Collection. The Collector stands in for a constructor, an accumulator, and a combiner (see below).
<b>Stream</b>	<code>collect(Supplier&lt;R&gt; supplier, BiConsumer&lt;R, ? super T&gt; accumulator, BiConsumer&lt;R, R&gt; combiner)</code>	Reduces a stream to an instance of a mutable type. Supplier provides a new instance of the mutable type such as a Collection; accumulator adds each element of the stream to the collection; combiner specifies how to merge elements from one collection to another. A Collector can be used to capture all three components.
<b>DoubleStream,</b> <b>IntStream,</b> <b>LongStream</b>	<pre>collect(Supplier&lt;R&gt; supplier, ObjDoubleConsumer(&lt;R&gt;     accumulator, BiConsumer&lt;R, R&gt; combiner)</pre> <pre>collect(Supplier&lt;R&gt; supplier, ObjIntConsumer(&lt;R&gt;     accumulator, BiConsumer&lt;R, R&gt; combiner)</pre> <pre>collect(Supplier&lt;R&gt; supplier, ObjLongConsumer(&lt;R&gt;     accumulator, BiConsumer&lt;R, R&gt; combiner)</pre>	Reduces a stream to an instance of a mutable type.

[Table 9-6](#) shows the variations on the `collect()` method. In this section we used only the first variation—the simplest method that takes a `Collector` created with one of the `Collectors` methods [e.g., `toList()`]—but be aware of the other variation where you need to supply supplier, accumulator, and combiner arguments to specify how the `collect()` method reduces elements in case it shows up on the exam. Most of the time, you'll typically use a `Collectors` method to create a `Collector` (which you can think of as an abstraction hiding the details of a supplier, accumulator, and combiner that actually determine how the collect reduction is done) and use that with `collect()`, as we did in this section.

[Table 9-7](#) shows some of the `Collectors` methods you can use to create `Collectors` to collect elements; for more options and variations (which will likely not be on the exam), check out the `Collectors` class in the documentation. Remember that all the methods in the `Collectors` class are static methods that each produce a `Collector`—and don't get those two types mixed up!

**TABLE 9-7** Collectors Methods

Method	Description
<code>toList()</code>	Returns a Collector that accumulates elements into a new List.
<code>toMap(Function&lt;? super T, ? extends K&gt; keyMapper, Function&lt;? super T, ? extends U&gt; valueMapper)</code>	Returns a Collector that accumulates elements into a new Map.
<code>toSet()</code>	Returns a Collector that accumulates elements into a new Set.
<code>toCollection( Supplier&lt;C&gt; collectionFactory)</code>	Returns a Collector that accumulates elements into a new Collection.
<code>groupingBy(Function&lt;? super T, ? extends K&gt; classifier)</code>	Returns a Collector that implements the group by operation, which groups elements by a classification Function, and returns the results in a Map. We used this to map Person objects by age.
<code>groupingBy(Function&lt;? super T, ? extends K&gt; classifier, Collector&lt;? super T, A, D&gt; downstream)</code>	Returns a Collector that groups elements according to a classification Function and then performs an additional reduction on the values associated with a given key in the resulting Map. We used this to count the number of people of a certain age and map the count to age.
<code>partitioningBy(Predicate&lt;? super T&gt; predicate)</code>	Returns a Collector that partitions elements according to the Predicate (into true and false partitions) and returns a Map. This is a more specific version of groupingBy.
<code>mapping(Function&lt;? super T, ? extends U&gt; mapper, Collector&lt;? super U, A, R&gt; downstream)</code>	Modifies a Collector that accepts elements of type U into a Collector that accepts elements of type T by applying the mapper Function to each element before it's accumulated in the Collector's reduction. We used this to map a Person object to a String (the person's name) before grouping by the person's age.



Method	Description
<code>summingInt(     ToIntFunction&lt;? super T&gt; mapper)</code>	Returns a Collector that computes the sum of input elements. We used this to sum the ages (type Integer) of Person objects with the same name; we used the mapper to map a Person to their age. Similar methods are <code>summingDouble</code> and <code>summingLong</code> (producing a Double sum and a Long sum, respectively). If no elements are present, the result is 0.
<code>averagingInt(     ToIntFunction&lt;? super T&gt;     mapper)</code>	Returns a Collector that computes the average of input elements. We used this to create a Map of Person names and their average age (type Double). Similar methods are <code>averagingDouble</code> and <code>averagingLong</code> (both of which produce Double values for the average). The mapper we used maps a Person to their age.
<code>counting()</code>	Returns a Collector that counts the number of elements and returns a long. If no elements are present, the result is 0.
<code>joining()</code>	Returns a Collector that concatenates elements into a String. We used this to concatenate Person names to associate a String of names with an age.
<code>maxBy(Comparator&lt;? super T&gt; comparator)</code>	Returns a Collector that returns the maximum element determined by the Comparator.
<code>minBy(Comparator&lt;? super T&gt; comparator)</code>	Returns a Collector that returns the minimum element determined by the Comparator.

## CERTIFICATION OBJECTIVE

### Streams of Streams (OCP Objective 5.7)

5.7 *Use flatMap() methods in the Stream API.*

Imagine you have a file containing space-separated words in multiple lines. Something like this:

```
rabbits dogs giraffes lions Java tigers  
penguins Java deer birds Java Java monkeys  
horses whales Java antelope bears Java  
Java insects Java raccoons rats zebras  
koalas snakes spiders cats hippopotamuses
```

You want to read the file and determine how many times the word “Java” appears in the file.

You already know you can create a stream from a file, so you do that to get started:

```
Stream<String> input = Files.lines(Paths.get("java.txt"));
```

So far so good.

Now, you know the input stream is going to stream one line at a time from the file “java.txt,” and so to get words, rather than lines, you can split each line, like this:

```
input.map(line -> line.split(" "))
```

What does this produce? You know the `split()` method on `String` creates an array of `Strings`, and you know `map()` produces a stream, so what this line of code does is take the stream of lines coming from the file, splits each line into a `String[]`, and generates a `Stream` of `String` arrays, which we write as `Stream<String[]>:`

```
Stream<String[]> inputStream = input.map(line -> line.split(" "));
```

So what do we do with a stream of `String` arrays? What we want is the individual strings in each array, so we can filter the words that are equal to “Java” and count them to see how many times “Java” appears in the file. So how do we turn the `Stream<String[]>` into a stream of `String`s?

What if we stream each of the arrays in the stream? We know we can create a stream from an array with `Arrays.stream()` and that `inputStream` is a stream of `String` arrays. What about mapping each array to a stream, so each array becomes a stream of strings, and then process that? Let’s try:

```
inputStream.map(array -> Arrays.stream(array)).forEach(System.out::println);
```

This is what you’ll see:

```
java.util.stream.ReferencePipeline$Head@72ea2f77
java.util.stream.ReferencePipeline$Head@33c7353a
java.util.stream.ReferencePipeline$Head@681a9515
java.util.stream.ReferencePipeline$Head@3af49f1c
java.util.stream.ReferencePipeline$Head@19469ea2
```

Hmm, something’s not quite right... We expected each word from the file to print, but instead we see what looks like streams. Why?

Well, what we’re creating with

```
inputStream.map(array -> Arrays.stream(array))
```

is actually a stream of streams:

```
Stream<Stream<String>> ss =
    inputStream.map(array -> Arrays.stream(array));
```

So when we try to print this out:

```
ss.forEach(System.out::println);
```

each item we see displayed is a `Stream<String>`, rather than a `String`. When we streamed each array of `String`s created by the split, each of those streams becomes an element in the main stream that is streaming the arrays. Thus, a

stream of streams.

What can we do with a stream of streams? Well, it turns out there's a method just for this situation in the Stream API: `flatMap()`.

The `flatMap()` method is similar to `map()` in that it maps a stream of one type into a stream of another type, but it does something extra; `flatMap()` “flattens” out the streams, essentially concatenating them into one stream. It replaces each stream with its contents, creating one stream from many.

So instead of mapping each array to a stream, we're going to *flat map* each array to a stream. The stream that results from the `flatMap()` is one big flat stream, rather than a stream of streams. That's really hard to see without an example, so let's give this a try:

```
Stream<String> ss =  
    inputStream.flatMap(array -> Arrays.stream(array));  
ss.forEach(System.out::println);
```

Think of taking a two-dimensional array and flattening it to a one-dimensional array, like this:

```
[ [ 1, 2 ], [ 3, 4 ] ] -> [ 1, 2, 3, 4 ]
```

That's what `flatMap()` does with streams. The result of using `flatMap()` on the stream of arrays is a flat stream of all the contents of each array, which is the words in the file.

If you run that code, you'll see all the words in the file:

```
rabbits  
dogs  
giraffes  
lions  
Java  
tigers  
penguins  
Java  
deer
```

```
...  
.
```

Great! We've made progress. Our goal is count the number of words equal to "Java." Hopefully that task is easier now; you just need to `filter()` and `count()`.

Here's the whole code so you can run it yourself:

```
try {  
    long n = Files.lines(Paths.get("java.txt")) // stream lines from the file  
        .map(line -> line.split(" ")) // split each line into String[]  
        .flatMap(array -> Arrays.stream(array)) // stream arrays, and flatten  
        .filter(w -> w.equals("Java")) // filter the words = "Java"  
        .count(); // count "Java"  
  
    System.out.println("Number of times 'Java' appears: " + n);  
} catch (IOException e) {  
    System.out.println("Oops, error! " + e.getMessage());  
}
```

Run that and you should see that "Java" appears in the file eight times.

The `Stream` interface also includes `flatMapToDouble()`, `flatMapToInt()`, and `flatMapToLong()`, to flat map to a `DoubleStream`, an `IntStream`, and a

`LongStream`, respectively, in case you need that. Each of the primitive stream types just mentioned also include their own `flatMap()` methods that take a `DoubleFunction`, an `IntFunction`, and a `LongFunction`, respectively.

What do you think? The final code is actually pretty concise, and it's fairly easy to process the data in the file. It's certainly a different way of thinking about processing data. Do you find the code easier or more difficult to read? It definitely takes some getting used to.

Keeping track of your streams—e.g., What is the type of your stream? Is it a stream of `Strings` or a stream of streams?—can get a bit tricky at times. Paying attention to exactly what each method produces so you know what type you're dealing with is important in all aspects of working with streams, as we hope you've discovered in this chapter.

## CERTIFICATION OBJECTIVE

### Generating Streams (OCP Objective 3.4)

#### 3.4 Collections, Streams, and Filters.

What do you think will happen if you run the following code:

```
Stream.iterate(0, s -> s + 1);
```

Looking at the documentation for `Stream.iterate()`, it says `iterate()` returns an “infinite sequential order Stream.” Oh my, did you get yourself an infinite loop here?

Actually no, you didn’t. Remember that streams are lazy. `Stream.iterate()` returns a stream, and you know that nothing starts to happen until you tack a terminal operation, like `count()` or `forEach()` or `collect()`, onto the stream pipeline.

The `iterate()` operation creates an infinite sequential stream, starting with the first argument (known as the “seed”) followed by elements created by the `UnaryOperator` that you supply as the second argument. In our example above, the stream will generate whole numbers starting at 0, adding 1 to the seed first and then each subsequent number, forever.

Okay, so here’s code you probably shouldn’t try (okay, *definitely* shouldn’t try):

```
Stream
```

```
.iterate(0, s -> s + 1)           // create an infinite stream  
.forEach(System.out::println); // don't try this at home!
```

Here, the `forEach()` terminal operation will never end because the stream is infinite. Because `iterate()` creates an infinite stream, we need some way to limit how much we get from the stream so we don't get ourselves in trouble. We can do that with the `limit()` operation:

```
Stream
```

```
.iterate(0, s -> s + 1)           // create an infinite stream...  
.limit(4)                         // but limit it to 4 things  
.forEach(System.out::println); // print the 4 things
```

Now it's safe to run this code, because we are limiting the stream to 4 numbers, starting with 0, so we see the output:

```
0  
1  
2  
3
```

To work safely with infinite streams, you need a short-circuiting operation. That could be `limit()`, like we used above, or it can also be an operation like `findFirst()`, `findAny()`, or `anyMatch()`.

Let's say you've got a sensor that generates an infinite stream of data. We won't actually build one of those, but we'll simulate one, like this:

```
class Sensor {  
    String value = "up";  
    int i = 0;  
    public Sensor() { }
```

```
public String next() {
    i = i + 1;
    return i > 10 ? "down" : "up";
}
}
```

This sensor has a `next()` method that returns the status of the sensor. We return “up” values for the status until `i` is 10 and return “down” when `i > 10`. The value `i` is incremented each time we call `next()`, so the first 10 results will be “up” and all subsequent results will be “down.” In a real sensor, we’d get data from the sensor and keep returning the latest status (use your imagination).

Now let’s use this sensor with an infinite stream. For this example, we’ll use the `Stream` method `generate()`, which takes a `Supplier`. The `generate()` method generates an infinite stream from elements supplied by the `Supplier`. Our `Supplier` is going to get values from the sensor to stream:

```
Sensor s = new Sensor();
Stream<String> sensorStream = Stream.generate(() -> s.next());
```

So `sensorStream` is an infinite stream of values we get by calling the sensor’s `next()` method. That infinite stream contains 10 “up” elements and a potentially infinite number of “down” elements.

Now we can write code to look for the first “down” value in the infinite stream, like this:

```
Optional<String> result =
    sensorStream
        .filter(v -> v.equals("down")) // filter to get all down values
        .findFirst();                // find the first down value and stop
    result.ifPresent(System.out::println);
```

The `findFirst()` method is a short-circuiting method, so as soon as we find a “down” value in the stream, everything stops. Whew! We averted another infinity problem. Of course, that assumes there is a “down” value somewhere in

the stream; if there's not, then the stream will keep generating values and `findFirst()` will never complete.

Infinite streams are handy when you're dealing with a source of potentially infinite data, like a sensor. In practice, because we'd actually like to do something with the data we get from a sensor, we need to process the data into chunks, defined perhaps by a timestamp or by the number of results so far or by a particular value that indicates a change. Use caution when dealing with infinite streams because you need a way to specify a stopping point with a short-circuiting operation in order to perform the reduction and get a result.

Although `Stream.iterate()` is a good way to generate numbers up to a certain point, another way you can generate numbers is with `range()`. In practice, this might actually be more useful. This is a method on the primitive streams `IntStream` and `LongStream`:

```
IntStream numStream = IntStream.range(10, 15); // generate numbers 10...14
numStream.forEach(System.out::println);
```

This produces the output:

```
10
11
12
13
14
```

If you want a stream of numbers inclusive of the second argument, use `rangeClosed()`:

```
IntStream numStream =
    IntStream.rangeClosed(10, 15); // generate numbers 10...15
numStream.forEach(System.out::println);
```

You'll get the output:

```
10
11
12
13
14
15
```

As you saw, the `limit()` method allows you to limit the number of elements in a stream, so you can, for instance, limit a stream to the first five even numbers in a stream:

```
IntStream evensBefore10 =
    IntStream                                // rangeClosed is static
        .rangeClosed(0, 20)                    // generate numbers 0...20
        .filter(i -> i % 2 == 0)              // filter evens
        .limit(5);                          // limit to 5 results
evensBefore10.forEach(System.out::println);
```

And get the output:

```
0
2
4
6
8
```

What if you want to skip the first five items instead and print only the even numbers between 10 and 20? You can use `skip()`:

```
IntStream evensAfter10 =  
    IntStream  
        .rangeClosed(0, 20)          // generate numbers 0...20  
        .filter(i -> i % 2 == 0)    // filter evens  
        .skip(5);                  // skip first 5 results  
evensAfter10.forEach(System.out::println);
```

And you see

```
10  
12  
14  
16  
18  
20
```

## Methods to Generate Streams

All the methods in [Table 9-8](#) are static stream methods to generate streams, except `limit()` and `skip()`, which you'll use to control how many items and which items go into the stream. Be extra careful with `iterate()` and `generate()`; both methods create infinite streams so you need to use these with `limit()` or one of the short-circuiting methods discussed earlier.

**TABLE 9-8** Stream Methods to Generate Streams

Method	Description
<code>iterate(T seed, UnaryOperator&lt;T&gt; function)</code>	<p>Applies a function to a seed and then to each subsequent value, and returns an infinite stream consisting of the resulting elements. For example, we created an infinite stream by starting with a seed of 0 and using a function that adds 1 to its input, resulting in a stream of Integers: 0, 1, 2, 3....</p> <p>Primitive streams DoubleStream, IntStream, and LongStream each have their own versions that use the primitive unary operators for the function (DoubleUnaryOperator, IntUnaryOperator, and LongUnaryOperator) and whose seeds are of type double, int, and long, respectively.</p>
<code>generate(Supplier&lt;T&gt; s)</code>	<p>Returns an infinite stream consisting of elements generated by the provided Supplier. We used this to generate a stream of "up" and "down" values.</p> <p>Primitive streams DoubleStream, IntStream, and LongStream have their own versions that use the primitive suppliers (DoubleSupplier, IntSupplier, and LongSupplier) and generate primitive streams, respectively.</p>
<code>range(int startInclusive, int endExclusive),  range(long startInclusive, long endExclusive)</code>	<p>Primitive streams IntStream and LongStream only: this method produces a sequential stream of elements, incrementing by 1 (e.g., 0, 1, 2, 3...), not including the endExclusive value.</p>
<code>rangeClosed(int startInclusive, int endInclusive),  rangeClosed(long startInclusive, long endInclusive)</code>	<p>Primitive streams IntStream and LongStream only: this method produces a sequential stream of elements, incrementing by 1 (e.g., 0, 1, 2, 3...), including the endInclusive value.</p>
<code>limit(long maxSize)</code>	<p>Returns a stream consisting of the elements of the stream on which it's called, limited to maxSize number of elements (for all stream types).</p>
<code>skip(long n)</code>	<p>Returns a stream consisting of the elements of the stream on which it's called, after n elements from the stream have been skipped (for all stream types).</p>

## Caveat Time Again

Streams are fun to play with, and they are cool because they are still kind of new. But just because it's new doesn't mean it's always better. We sold you pretty hard on streams being concise and potentially more efficient for data processing, but there are times when you may not want to use streams. For example, just doing a simple iteration over a `List` might be *slower* using a stream than a `for` loop. It's worth testing to find out for your particular use case. In addition, functional code with streams and lambdas isn't always easy to read, particularly if you're used to reading code written in a more imperative style. So our caveat is this: just because you *can* use streams doesn't mean they are always appropriate. Keep this in mind as you go beyond the exam and into the real world again.

Where streams can really shine is when you can parallelize them. We're going to briefly talk about parallel streams now and then talk about them more later in the book when we talk about concurrency in depth.

## CERTIFICATION OBJECTIVE

### A Taste of Parallel Streams

10.6 *Use parallel Streams including reduction, decomposition, merging processes, pipelines and performance.*

We've mentioned a couple of times in this chapter that it's possible to process streams in parallel. So far, all the streams we've worked with have been serial streams: streams that process one data element at a time. *Parallel streams* can process elements in a stream concurrently. The way Java does this is to split a stream into substreams and then execute the operations defined in the stream pipeline on each of these substreams concurrently, meaning each substream is processed in a thread. If you have multiple cores, then Java will use multiple threads to process the stream and you might get some performance benefits. (We say "might" here because whether you get those benefits depends on your system as well as the data you're processing and how you're processing it. We won't go into an in-depth discussion on the performance tradeoffs of parallel streams, but if you're interested, it's well worth a deep dive into the literature about what makes for good use cases for parallel streams).

You're going to learn a whole lot more about threads and parallel streams in the upcoming chapter on threads, so for now, we're just going to give you a taste.

Let's say you have some numbers and you want to sum them.

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
int sum = nums.stream()                                // stream the numbers
    .mapToInt(n -> n)                                 // map Integer to int for sum
    .sum();                                            // sum the ints
System.out.println("Sum is: " + sum);                  // result is 55
```

This stream is a serial stream, so each number is added to the sum one at a time.

To make this a parallel stream, we simply call the method `parallel()` on the stream before we sum:

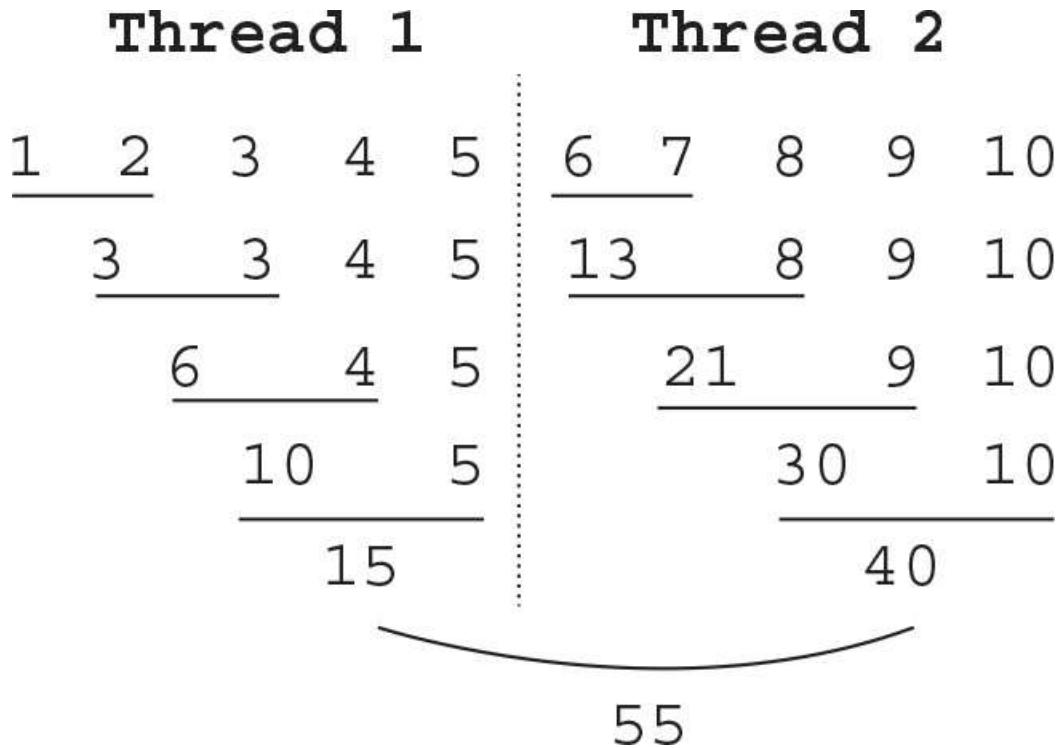
```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
int sum = nums.stream()
    .parallel()                                       // make the stream parallel
    .mapToInt(n -> n)
    .sum();
System.out.println("Sum is: " + sum);      // result is still 55 (whew!)
```

Now, the stream is processed concurrently, meaning the stream is split into substreams, and (if you have enough cores) each substream is processed on a separate thread. Let's visualize how that works with some diagrams.

For a serial stream, we have one thread handling the entire sum operation:

$$\begin{array}{cccccccccc}
 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
 \hline
 3 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
 \hline
 6 & & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
 \hline
 & 10 & & 5 & 6 & 7 & 8 & 9 & 10 \\
 \hline
 & & 15 & & 6 & 7 & 8 & 9 & 10 \\
 \hline
 & & & 21 & & 7 & 8 & 9 & 10 \\
 \hline
 & & & & 28 & & 8 & 9 & 10 \\
 \hline
 & & & & & 36 & & 9 & 10 \\
 \hline
 & & & & & & 45 & & 10 \\
 \hline
 & & & & & & & 55 &
 \end{array}$$

For a parallel stream, we have multiple threads handling the sum operation concurrently:



Because each thread can compute the sum simultaneously with the other threads, the whole operation should take less time to compute than if you're using a serial stream.

One thing to be careful of with parallel streams is performing an operation that relies on a specific ordering. If you use a parallel stream, you may get unexpected results. Let's compare using `forEach()` to display the items in a stream when the stream is serial and when the stream is parallel.

Here's the code using a serial stream:

```
Arrays.asList("boi", "charis", "zooey", "aiko")
    .stream()                                // stream the names
    .forEach(System.out::println); // display them
```

The output you see is

```
boi
charis
zooey
aiko
```

In other words, you see the data in the same order as it appears in the source of the stream, the original `List` of names. That is because, by default, the stream is serial, and the elements in the stream are processed in order.

If you run the same code with a parallel stream instead:

```
Arrays.asList("boi", "charis", "zooey", "aiko")
    .stream()                                // stream the names
    .parallel()                               // ... in parallel
    .forEach(System.out::println);           // display them
```

You'll potentially get a different ordering in the output every time you run the code. For instance, when we ran this code, we saw

```
zooey
charis
boi
aiko
```

And if we run it again, we'll likely see a different ordering again. That's because the final result depends on the order in which the threads complete, not the order of the original collection. The ordering of the stream didn't matter when we were summing numbers (the sum is the same, independent of order), but the order matters a lot if you are expecting to see the content of the stream in the same order as the content in the original `Collection`.

That is just a small taste of parallel streams, and you'll learn a lot more about them a bit later on in [Chapter 11](#), where we cover concurrency.

## CERTIFICATION SUMMARY

---

A stream is a fairly abstract concept: it is a sequence of elements supporting operations. Knowing the difference between a stream and a data structure is key to understanding streams, and the analogies we used in this chapter—real-life streams and assembly lines—are meant to remind you that streams are for specifying a sequence of operations—the stream pipeline—to perform on a source. Streams never actually hold data like a data structure does. Once you wrap your head around that, then you'll have a much easier time getting the hang

of the pipeline operations, of which there are many that you can perform on a stream.

You create a stream from a source, like a `List` or a `File`; specify the sequence of intermediate operations to perform; and finally, provide the terminal operation that terminates the stream and perhaps reduces the stream into a single value or displays the values from the stream in the console. But streams are lazy, so none of the intermediate operations in the pipeline actually do anything until that terminal operation is executed.

The main way you operate on streams is with mapping functions, filter functions, and reduction functions. Mapping functions map values to other values, perhaps changing the type along the way. Filter functions filter out some of the values (or none or all, depending on the filter), so you can eliminate values that don't pass a test. Finally, reduction functions provide the terminal operations on streams to reduce a stream to a single value or to a collection. This map-filter-reduce perspective on stream operations is a handy way to organize the many operations you'll encounter as you work with streams.

Streams can be sorted and searched, like data structures can—but remember you can't modify the source of a stream. When you, say, sort a stream, you need to collect the sorted values into a new data structure if you want to keep them around. Searching produces a value, but because a stream might be empty, you need a way to represent the concept of a value that may or may not be there. This is when you first encountered the optionals: values that are wrapped in the `Optional` type so you can work with empty streams without creating any problems. Of course, there are a variety of ways to get values back out of the `Optional` wrapper so you can work with the result as you normally do.

If you want to keep multiple values from a stream rather than reducing the stream to a single value, then you need a way to collect those values into a new data structure. This is what the `collect()` reduction method does. This method takes a `Collector`, which is an operation that accumulates the elements into a data structure like a `List` or a `Map`. As you're collecting elements, you can group them, partition them, count them, map them, reduce them, and more. Collectors are versatile, providing many different ways of collecting values from the stream into a result. We touched on some of the collection strategies in this chapter for the operations you'll apply on the exam, but you can take this topic a lot further if you want to explore on your own.

Generating streams is a way of pushing a (potentially) infinite number of values through the stream pipeline. However, in practice, you can't actually work with an infinite stream (at least not on today's computers!), so you need to

limit it in some way, either by limiting how many items to use or by short-circuiting the stream pipeline with a searching method like `findAny()`.

At the end of the chapter, we took a sneak peek at parallel streams, which are streams that can be operated on concurrently. Parallel streams are where you are likely to see the main performance benefits of using streams in your code, and parallel streams are certainly one of the easiest ways to take advantage of concurrency in your code, as you'll see when we get to [Chapter 11](#) (although, as with all things related to concurrency, there are some potential landmines you'll need to watch out for when you get there).



## TWO-MINUTE DRILL

Here are some of the key points from this chapter.

### What Is a Stream? (OCP Objective 3.4)

- A stream is a sequence of elements that can be processed with operations.
- A stream is not a data structure; it does not store any data.
- A stream pipeline consists of a source, an optional sequence of intermediate operations, and a terminal operation.
- The source of a stream can be a collection, an array, a file, or one or more values.
- You can create a stream of objects or a stream of primitive values.
- The type of a stream of objects is `Stream<T>`, and the primitive stream types are `DoubleStream`, `IntStream`, and `LongStream`.
- Pay close attention to the difference between `Stream<Double>` and `DoubleStream` and the operations allowed on each (and the same with other primitive type streams).
- Intermediate operations operate on a stream and return a stream.
- Terminal operations terminate the stream, returning a value other than a stream or void.
- `filter()` is an example of an intermediate operation. Pass `filter()` a `Predicate` and use `filter()` to filter values out of the stream that don't pass the `Predicate` test. For example, you can filter out elements that are

less than 5 from a stream like this:

```
Stream.of(0, 1, 1, 2, 3, 5, 8, 13).filter(i -> i >= 5);
```

The stream returned by this filter operation is a stream of numbers: 5, 8, 13.

- `forEach()` is a terminal operation on a stream. `forEach()` takes a Consumer and consumes each element of the stream on which it's called. We often use `forEach()` to display the values in the stream at the end of the stream pipeline, like this:

```
Stream.of(0, 1, 1, 2, 3, 5, 8, 13).filter(i -> i >= 5);
```

- `count()` is a terminal operation on a stream that counts the elements in the stream.
- Streams can only be used once. Streams are lightweight objects, so you can easily create another stream.

## How to Create a Stream (OCP Objectives 3.5 and 9.3)

- Create a stream from a collection, like a `List`, using the `stream()` method.
- Create a stream from one or more values using `Stream.of()`.
- Create an empty stream using `Stream.empty()`.
- Create a stream from an array using `Arrays.stream()` or `Stream.of()` (and `IntStream.of()`, `LongStream.of()`, and `DoubleStream.of()`).
- Create a stream (`Stream<String>`) from a `File` using `Files.lines()`.
- Streams have several benefits: you can define multiple intermediate operations on streams. The JDK can optimize the operations, so you may see a performance enhancement, especially when you can parallelize the stream.

## The Stream Pipeline (OCP Objectives 3.6)

- A stream pipeline consists of a source, a sequence of intermediate operations, and a terminal operation.
- A common analogy to use for the stream pipeline is the assembly line. Each assembly-line station is analogous to an intermediate operation, so that a stream element visits each operation, where it is (usually) modified

and passed on until it reaches the terminal operation or is discarded (via a filter).

- ❑ Streams are lazy, meaning until you define and execute a terminal operation on the stream, no processing happens.

## Operating on Streams (OCP Objectives 3.7 and 5.1)

- ❑ Map-filter-reduce is a general abstraction to describe functions that operate on a sequence of elements.
- ❑ Stream operations `map()`, `filter()`, and `reduce()` implement mapping operations, filtering operations, and reduction operations (terminal operations), respectively.
- ❑ Mapping operations typically modify elements from the stream, transforming an element from one value to another or from one type to another.
- ❑ Filter operations typically winnow elements from the stream so that any element that doesn't pass a test is discarded from the stream.
- ❑ Reduction operations reduce the stream to a single value, or a collection.
- ❑ `map()` and `filter()`, and their variations, are intermediate operations.
- ❑ `reduce()` and its variations are terminal operations.
- ❑ `map()` takes a `Function`, which takes an input value and produces an output value.
- ❑ `filter()` takes a `Predicate`, which tests the input value and passes the stream element on to the next intermediate operation if the result of the test is true.
- ❑ The methods `average()`, `count()`, `sum()`, `max()`, and `min()` are predefined reductions on streams.
- ❑ `sum()` and `average()` are defined only on the primitive stream types.
- ❑ `max()` and `min()` are defined on all streams. On `Stream<T>`, `max()` and `min()` take a `Comparator` to determine which stream element is the maximum or minimum, respectively.
- ❑ `reduce()` takes an (optional) identity value and a `BinaryOperator` to reduce the stream to one value. If the identity value is provided, this is used as the result if the stream is empty; otherwise, the identity is used as the basis for the `BinaryOperator` accumulator. If no identity value is

provided, `reduce()` returns an `Optional`.

- The `peek()` method is a way to “peek” into the values currently in the stream. The method makes no changes to the values in the stream and is often used with a `Consumer` that displays the values to the console for debugging purposes. `peek()` should not be used in production code.

## Map-Filter-Reduce with `average()` and Optionals (OCP Objectives 5.3 and 5.4)

- Some stream methods produce `Optional` values.
- `Optional` is a wrapper around a value that may or may not be there.
- `average()` is an example of a reduction operation that produces an optional value, an `OptionalDouble`.
- The `reduce()` method used without an identity argument produces an optional because there may be no value if the stream is empty.
- If you supply an identity argument to `reduce()`, the method will produce a value (not an optional), and use the identity value if the stream is empty.
- The `sum()` method does not produce an optional because, by default, it uses 0 as the identity value.
- The `average()` method cannot be replaced with your own implementation using `reduce()` because `average` is not an associative operation. All `reduce()` operations must be associative operations, meaning you can accumulate the elements for the reduction in any order and get the same result. `Sum` is an example of an associative operation, so you could implement `sum()` yourself using `reduce()`.

## Optionals (OCP Objective 5.3)

- An optional is a container, or wrapper, that may or may not contain a value.
- Operations that produce no value if a stream is empty produce optionals. For example, `findFirst()`, `findAny()`, `max()`, `min()`, and `average()` all produce optionals.
- Before getting a value from an optional, first test to see if the optional contains a value with `isPresent()`.
- The types of optionals are `Optional<T>` (for objects) and primitive

optionals, `OptionalDouble`, `OptionalInt`, and `OptionalLong`.

- ❑ To get a value from a nonempty optional, use `get()`, `getAsDouble()`, `getAsInt()`, or `getAsLong()` for the different types of optionals.
- ❑ The `ifPresent()` method tests to see whether an optional is present and then passes the unwrapped value to the `Consumer` argument to `ifPresent()`.
- ❑ Most of the time optionals are created by stream operations; however, you can create your own optional using `Optional.of()`.
- ❑ The `ofNullable()` method creates an `Optional` value from an object, first testing to see whether the object is `null`. If the object is `null`, then an empty `Optional` is created.
- ❑ You can create your own empty `Optional` with `Optional.empty()`.
- ❑ Use the `orElse()` method to get a value from an optional and provide a default value to use if the optional is empty.

## Searching and Sorting with Streams (OCP Objectives 5.2 and 5.5)

- ❑ Searching operations on streams are short-circuiting, meaning the stream processing stops once the `Predicate` test passed to the search method is passed.
- ❑ The search methods on streams include `allMatch()`, `anyMatch()`, `noneMatch()`, `findFirst()`, and `findAny()`.
- ❑ The methods `allMatch()`, `anyMatch()`, and `noneMatch()` are terminal operations on a stream that return a boolean indicating if a match was found (or not).
- ❑ The methods `findFirst()` and `findAny()` are terminal operations that return an `Optional` value whose parameterized type depends on the type of the stream. In case no item is found, the value returned is an empty `Optional`.
- ❑ You can sort elements of a stream using the `sorted()` method. By default, the `sorted()` method uses natural ordering. If the objects in the source of the stream are `Comparable`, the sort will use that order. If the elements of a stream are primitive, the natural order for sorting is used for `double`, `int`, and `long`.

- You can pass a Comparator to the sorted() method to override the natural sort order and define your own sort order.
- Comparator is a functional interface with one functional method, compare(), so you can use a lambda expression to pass a Comparator to the sorted() method.
- You can use the Comparator.comparing() method with a Function to create a Comparator to use with sorted().
- Use thenComparing() to combine comparators for sorting by one value then another.
- Instance method references refer to methods of an instance. Use the :: syntax to create a method reference as a shorthand for a lambda that simply calls another method. For example, you can replace a lambda expression that calls the getColor() method of a Duck instance:  
`d -> d.getColor()`  
with an instance method reference:  
`Duck::getColor`
- Streams are a functional style of programming in which you must be careful to avoid side effects. Stream patterns encourage you to avoid modifying the source of the stream in any stream pipeline operations, and in fact, modifying the source of a stream can lead to unexpected results.

## Collecting Values from Streams (OCP Objectives 3.8 and 5.6)

- The source of a stream is often a Collection. Many reduction methods on streams, such as sum() and average(), turn stream elements into a single value. You can collect the elements in a stream into a new Collection using the stream method collect().
- The collect() stream method is a reduction (terminal) operation.
- A Collector is an operation that accumulates elements into a result. You can specify a Collector with four functions: a supplier, an accumulator, a combiner, and a finisher (optional). Or, you can use one of the methods in the Collectors class to create a Collector.
- The Collectors class has ready-made Collectors, such as a Collector to accumulate elements into a List.

- Use the `Collectors.toList()` method to create a Collector that accumulates stream elements into a `List`. `Collectors.toMap()` and `Collectors.toSet()` accumulate stream elements into a `Map` and `Set`, respectively.
- If you want a more specific Collection type, use the `Collectors.toCollection()` method, passing a `Supplier` that provides a new instance of the Collection type you want (such as `ArrayList`).
- The method reference `ArrayList::new` is a constructor method reference that is shorthand for a lambda expression `() -> new ArrayList<T>()` (where `T` is a valid type parameter, such as `String`, and depends on the type of the elements in the stream).
- You can use constructor method references as `Suppliers` for the `Collectors.toCollection()` method.
- Use the `Collectors.groupingBy()` method to group elements as you collect them. The `Collectors.groupingBy()` method returns a Collector that produces a `Map`.
- Use the `Collectors.partitioningBy()` method to group elements into two partitions as you collect them. The `Collectors.partitioningBy()` method returns a Collector that produces a `Map` with two keys, `true` and `false`.
- The `Collectors.groupingBy()` method can accept downstream collectors to further reduce results. For instance, instead of creating a `Map` that maps age keys to a `List` of `Person` objects that have that age, you can use the `Collectors.counting()` method to reduce the `List` of `Person` objects to their count, resulting in a `Map` from age keys to number of `Persons` with that age values.
- `Collectors.counting()` returns a Collector that counts the number of input elements and returns a `Long`.
- You can use `Collectors.mapping()` as a downstream Collector that maps values from one type to another before they are accumulated. For instance, we used `Collectors.mapping()` to create a Collector that maps `Person` objects to names and to collect them in a `List`.
- `Collectors.summingInt()` and `Collectors.averagingInt()`, and their `Double` and `Long` counterparts, produce Collectors that sum values and average values, respectively. These can be used as downstream collectors.
- `Collectors.joining()` produces a Collector that concatenates `String`

elements (and any CharSequence-based types) to one String.

- `Collectors.counting()` produces a collector that counts elements and returns a Long.
- `Collectors.maxBy()` and `Collectors.minBy()` produce a collector that finds the maximum or minimum element in a stream when supplied with a Comparator.

## Streams of Streams (OCP Objective 5.7)

- Sometimes the type of Stream in a pipeline is a Stream of Streams. For instance: `Stream<Stream<String>>` is a stream of string streams.
- You can use the Stream method `flatMap()` to flatten streams, essentially concatenating the values from each stream within the stream. For instance, to flatten a `Stream<Stream<String>>` sss that you created with code like this:

```
Stream<Stream<String>> sss =  
    Stream.of(Stream.of("one", "two"),  
              Stream.of("three", "four"),  
              Stream.of("five", "six"));
```

you could write:

```
sss.flatMap(s -> s).forEach(System.out::println);
```

## Generating Streams (OCP Objective 3.4)

- You can create infinite streams with the Stream methods `iterate()` and `generate()`.
- The Stream method `iterate()` method takes a seed (e.g., 0) and a UnaryOperator to generate a sequential ordered stream of values. The primitive streams have corresponding `iterate()` methods that generate streams of primitive values.
- The Stream method `generate()` method takes a Supplier that generates values for a Stream.
- Be careful when working with infinite streams. To do something useful with them, you need to limit the size of the stream with `limit()` or by using a short-circuiting method like `anyMatch()` or `findAny()`.

- ❑ You can generate `IntStreams` and `LongStreams` streams using the `range()` and `rangeClosed()` methods, which produce a sequence of numbers from a start value to an end value.

## A Taste of Parallel Streams (OCP Objective 10.6)

- ❑ By default, streams are sequential, meaning the operations in the stream pipeline are processed sequentially on the source data.
- ❑ You can parallelize a stream using the `parallel()` method. But be careful how you use this method; not all stream pipelines can be parallelized!
- ❑ We'll take a more in-depth look at parallel streams in [Chapter 11](#), which covers concurrency.

## Q SELF TEST

The following questions will help you practice using stream operations, and build and measure your understanding of the material in this chapter.

1. Given the code fragment:

```
int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
int sum = Arrays.stream(nums)  
    // L1
```

Which code fragment inserted at line `// L1` will compile correctly and sum the numbers in the array `nums` and store the value in the variable `sum`?

- A. `.reduce( (n1, n2) -> n1 + n2);`
  - B. `.reduce(0, (n1, n2) -> n1 + n2);`
  - C. `.reduce(nums);`
  - D. `.reduce(0, n1 + n2);`
2. Given the following code fragment:

```
try (Stream<String> stream = Files.lines(Paths.get("names.txt"))) {  
    // L1  
    System.out.println("Sorted names in the file: " + names);  
} catch(IOException e) {  
    System.out.println("Error reading names.txt");  
    e.printStackTrace();  
}
```

and a file, “names.txt,” with one name per line, which of the following code fragments inserted at line `// L1` will produce a sorted `List` of names in the variable `names`?

A.

```
List<String> names = stream.sorted().toList();
```

B.

```
List<String> names = stream  
.comparing((n1, n2) -> n1.compareTo(n2))  
.collect(Collectors.toList());
```

C.

```
List<String> names = stream.collect(Collectors.toList()).sorted();
```

D.

```
List<String> names = stream.sorted().collect(Collectors.toList());
```

3. What are the correct types for variables `s1` and `s2` in the code fragment below?

```
TYPE s1 = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);  
TYPE s2 = s1.mapToInt(i -> i);
```

- A. Stream<Integer> and Stream<Integer>
  - B. IntStream and IntStream
  - C. IntStream and Stream<Integer>
  - D. Stream<Integer> and IntStream
  - E. Stream<Integer> and List<Integer>
4. Given the code fragments:

```
class Duck implements Comparable<Duck> {
    String name;
    String color;
    int age;

    public Duck(String name, String color, int age) {
        this.name = name; this.color = color; this.age = age;
    }
    // getters and setters here
    public String toString() {
        return getName() + " is " + getColor() + " and is "
               + getAge() + " years old.";
    }
    @Override
    public int compareTo(Duck duck) {
        return this.getName().compareTo(duck.getName());
    }
}

List<Duck> ducks = Arrays.asList(
    new Duck("Jerry", "yellow", 3),
    new Duck("George", "brown", 4),
    new Duck("Kramer", "mottled", 6),
    new Duck("Elaine", "white", 2),
    new Duck("Huey", "mottled", 2),
    new Duck("Louie", "white", 4),
    new Duck("Dewey", "brown", 6)
);
```

Which code fragment could you use for CODE in the call to `map()` in the following code fragment:

```
System.out.print("Duck names: ");
ducks.stream().map(CODE).forEach(d -> System.out.print(d + " "));
System.out.println();
```

to correctly print the string (choose all that apply):

Duck names: Jerry George Kramer Elaine Huey Louie Dewey

- A. `Duck::getName`
  - B. `d::d.getName()`
  - C. `d -> d.getName()`
  - D. `d.getName()`
5. Given the `Duck` class and `ducks` List:

```
class Duck implements Comparable<Duck> {
    String name;
    String color;
    int age;

    public Duck(String name, String color, int age) {
        this.name = name; this.color = color; this.age = age;
    }
    // getters and setters here
    public String toString() {
        return getName() + " is " + getColor() + " and is "
               + getAge() + " years old.";
    }
    @Override
    public int compareTo(Duck duck) {
        return this.getName().compareTo(duck.getName());
    }
}

List<Duck> ducks = Arrays.asList(
    new Duck("Jerry", "yellow", 3),
    new Duck("George", "brown", 4),
    new Duck("Kramer", "mottled", 6),
    new Duck("Elaine", "white", 2),
    new Duck("Huey", "mottled", 2),
    new Duck("Louie", "white", 4),
    new Duck("Dewey", "brown", 6)
);
```

What output does the following code fragment produce?

```
ducks.stream()  
    .filter(d -> d.getColor().equals("mottled"))  
    .map(d -> d.getName())  
    .foreach(d -> System.out.print(d + " "));
```

- A. Kramer Huey
  - B. Jerry George Kramer Elaine Huey Louie Dewey
  - C. No output, the code does not compile
  - D. Kramer is mottled and is 6 years old. Huey is mottled and is 2 years old.
6. Given the Duck class and ducks List:

```
class Duck implements Comparable<Duck> {
    String name;
    String color;
    int age;

    public Duck(String name, String color, int age) {
        this.name = name; this.color = color; this.age = age;
    }
    // getters and setters here
    public String toString() {
        return getName() + " is " + getColor() + " and is "
               + getAge() + " years old.";
    }
    @Override
    public int compareTo(Duck duck) {
        return this.getName().compareTo(duck.getName());
    }
}

List<Duck> ducks = Arrays.asList(
    new Duck("Jerry", "yellow", 3),
    new Duck("George", "brown", 4),
    new Duck("Kramer", "mottled", 6),
    new Duck("Elaine", "white", 2),
    new Duck("Huey", "mottled", 2),
    new Duck("Louie", "white", 4),
    new Duck("Dewey", "brown", 6)
);
```

Which code fragment would you use to compute the average age of the ducks as a whole number?

A.

```
double avgAge =  
    ducks.stream().mapToInt(d -> d.getAge()).average();
```

B.

```
OptionalDouble avgAge =  
    ducks.stream().mapToInt(d -> d.getAge()).average();
```

C.

```
Double avgAge =  
    ducks.stream().mapToDouble(d -> d.getAge()).average();
```

D.

```
double avgAge =  
    ducks.stream().map(d -> d.getAge()).average();
```

7. Given the Duck class and ducks List:

```
class Duck implements Comparable<Duck> {
    String name;
    String color;
    int age;

    public Duck(String name, String color, int age) {
        this.name = name; this.color = color; this.age = age;
    }
    // getters and setters here
    public String toString() {
        return getName() + " is " + getColor() + " and is "
               + getAge() + " years old.";
    }
    @Override
    public int compareTo(Duck duck) {
        return this.getName().compareTo(duck.getName());
    }
}

List<Duck> ducks = Arrays.asList(
    new Duck("Jerry", "yellow", 3),
    new Duck("George", "brown", 4),
    new Duck("Kramer", "mottled", 6),
    new Duck("Elaine", "white", 2),
    new Duck("Huey", "mottled", 2),
    new Duck("Louie", "white", 4),
    new Duck("Dewey", "brown", 6)
);
```

Which code fragment would you use to count how many mottled ducks there are?

A.

```
long count = ducks.stream().filter(d -> d.getColor().equals("mottled")).count();
```

B.

```
int count = ducks.stream().filter(d -> d.getColor().equals("mottled")).count();
```

C.

```
long count = ducks.stream().filter(d -> d.equals("mottled")).count();
```

D.

```
long count = ducks.stream().filter(Duck::getColor().equals("mottled")).count();
```

8. Given the Duck class and ducks List:

```
class Duck implements Comparable<Duck> {
    String name;
    String color;
    int age;

    public Duck(String name, String color, int age) {
        this.name = name; this.color = color; this.age = age;
    }
    // getters and setters here
    public String toString() {
        return getName() + " is " + getColor() + " and is "
               + getAge() + " years old.";
    }
    @Override
    public int compareTo(Duck duck) {
        return this.getName().compareTo(duck.getName());
    }
}

List<Duck> ducks = Arrays.asList(
    new Duck("Jerry", "yellow", 3),
    new Duck("George", "brown", 4),
    new Duck("Kramer", "mottled", 6),
    new Duck("Elaine", "white", 2),
    new Duck("Huey", "mottled", 2),
    new Duck("Louie", "white", 4),
    new Duck("Dewey", "brown", 6)
);
```

And the following code fragment:

```
ducks.stream()
    .collect(Collectors.groupingBy(d -> d.getColor()))
    .forEach((c, dl) -> {
        System.out.print("Ducks who are " + c + ": ");
        dl.forEach(d -> System.out.print(d.getName() + " "));
        System.out.println();
    });
}
```

What is the result?

- A. Compilation fails
- B. An exception is thrown at runtime
- C.

{Ducks who are color:[Kramer Huey ], Ducks who are color:[Elaine Louie ], Ducks who are color:[Jerry ], Ducks who are color:[George Dewey ]}

- D.

Ducks who are mottled: Kramer Huey  
Ducks who are white: Elaine Louie  
Ducks who are yellow: Jerry  
Ducks who are brown: George Dewey

- 9. Given the Duck class and ducks List:

```
class Duck implements Comparable<Duck> {
    String name;
    String color;
    int age;

    public Duck(String name, String color, int age) {
        this.name = name; this.color = color; this.age = age;
    }
    // getters and setters here
    public String toString() {
        return getName() + " is " + getColor() + " and is "
               + getAge() + " years old.";
    }
    @Override
    public int compareTo(Duck duck) {
        return this.getName().compareTo(duck.getName());
    }
}

List<Duck> ducks = Arrays.asList(
    new Duck("Jerry", "yellow", 3),
    new Duck("George", "brown", 4),
    new Duck("Kramer", "mottled", 6),
    new Duck("Elaine", "white", 2),
    new Duck("Huey", "mottled", 2),
    new Duck("Louie", "white", 4),
    new Duck("Dewey", "brown", 6)
);
```

and the following code fragment:

```
TYPE duckMap =  
    ducks.stream()  
        .collect(Collectors.groupingBy(d -> d.getColor(), Collectors.toList()));
```

What is the correct type for `duckMap` in `TYPE`?

- A. `Map<String, Duck>`
- B. `List<Duck>`
- C. `Map<String, List<Duck>>`
- D. `Map<List<Duck>, String>`

10. Given the code fragment:

```
List<Integer> myInts = Arrays.asList(5, 10, 7, 2, 8);  
TYPE minInteger = myInts.stream().mapToInt(i -> i).min();
```

What is the correct `TYPE` for `minInteger`?

- A. `int`
- B. `Integer`
- C. `OptionalInt`
- D. `Optional<Integer>`
- E. `Stream<Integer>`
- F. `IntStream`

11. Given the following code:

```
class Temperature {
    String location;
    Double temp;
    public Temperature(String location, Double temp) {
        this.location = location; this.temp = temp;
    }
    public String getLocation() { return this.location; }
    public Double getTemp() { return this.temp; }
    public String toString() {
        return "July average temp at " + location + " was " + temp;
    }
}
List<Temperature> julyAvgs = new ArrayList<Temperature>();
julyAvgs.add(new Temperature("Death Valley, CA", 107.4));
julyAvgs.add(new Temperature("Salt Lake City, UT", 85.3));
julyAvgs.add(new Temperature("Reno, NV", 80.5));
julyAvgs.add(new Temperature("Bishop, CA", 80.8));
julyAvgs.add(new Temperature("Phoenix, AZ", 106.0));
julyAvgs.add(new Temperature("Miami, FL", 85.7));
```

and the following code fragment:

```
Comparator<Temperature>
tCompare = ((t1, t2) -> t1.getTemp().compareTo(t2.getTemp()));
TYPE max = julyAvgs.stream().max(tCompare);
```

What is the correct TYPE for max?

- A. Optional<Double>
- B. Optional<Temperature>
- C. OptionalDouble
- D. Temperature
- E. Double
- F. double

12. Given the Temperature class and julyAvgs List:

```
class Temperature {
    String location;
    Double temp;
    public Temperature(String location, Double temp) {
        this.location = location; this.temp = temp;
    }
    public String getLocation() { return this.location; }
    public Double getTemp() { return this.temp; }
    public String toString() {
        return "July average temp at " + location + " was " + temp;
    }
}
List<Temperature> julyAvgs = new ArrayList<Temperature>();
julyAvgs.add(new Temperature("Death Valley, CA", 107.4));
julyAvgs.add(new Temperature("Salt Lake City, UT", 85.3));
julyAvgs.add(new Temperature("Reno, NV", 80.5));
julyAvgs.add(new Temperature("Bishop, CA", 80.8));
julyAvgs.add(new Temperature("Phoenix, AZ", 106.0));
julyAvgs.add(new Temperature("Miami, FL", 85.7));
```

and the following code fragment:

```
TYPE maxT = julyAvgs.stream().mapToDouble(t -> t.getTemp()).max();
```

What is the correct **TYPE** for **maxT**?

- A. Optional<Double>
- B. Optional<Temperature>
- C. OptionalDouble
- D. Temperature
- E. Double
- F. double

13. Given the Temperature class and julyAvgs List:

```
class Temperature {
    String location;
    Double temp;
    public Temperature(String location, Double temp) {
        this.location = location; this.temp = temp;
    }
    public String getLocation() { return this.location; }
    public Double getTemp() { return this.temp; }
    public String toString() {
        return "July average temp at " + location + " was " + temp;
    }
}
List<Temperature> julyAvgs = new ArrayList<Temperature>();
julyAvgs.add(new Temperature("Death Valley, CA", 107.4));
julyAvgs.add(new Temperature("Salt Lake City, UT", 85.3));
julyAvgs.add(new Temperature("Reno, NV", 80.5));
julyAvgs.add(new Temperature("Bishop, CA", 80.8));
julyAvgs.add(new Temperature("Phoenix, AZ", 106.0));
julyAvgs.add(new Temperature("Miami, FL", 85.7));
```

and the following code fragment:

```

Comparator<Temperature> tCompare =
    ((t1, t2) -> t1.getTemp().compareTo(t2.getTemp()));
Optional<Temperature> min = julyAvgs.stream().min(tCompare);
min.ifPresent(m -> System.out.println("Min: " + m));
Optional<Temperature> coolerSpot =
    julyAvgs.stream().filter(t -> t.getTemp() < 100.0).findAny();
if (coolerSpot.isPresent()) {
    System.out.println("A cooler spot for July: " + coolerSpot.get());
} else {
    System.out.println("No place cool in July!");
}

```

Will the minimum temperature, `min`, be the same as `coolerSpot`?

- A. Yes
- B. No
- C. Maybe

**14.** Given the `Temperature` class and `julyAvgs` List:

```

class Temperature {
    String location;
    Double temp;
    public Temperature(String location, Double temp) {
        this.location = location; this.temp = temp;
    }
    public String getLocation() { return this.location; }
}

```

```
    public Double getTemp() { return this.temp; }
    public String toString() {
        return "July average temp at " + location + " was " + temp;
    }
}
List<Temperature> julyAvgs = new ArrayList<Temperature>();
julyAvgs.add(new Temperature("Death Valley, CA", 107.4));
julyAvgs.add(new Temperature("Salt Lake City, UT", 85.3));
julyAvgs.add(new Temperature("Reno, NV", 80.5));
julyAvgs.add(new Temperature("Bishop, CA", 80.8));
julyAvgs.add(new Temperature("Phoenix, AZ", 106.0));
julyAvgs.add(new Temperature("Miami, FL", 85.7));
```

and the following code fragment:

```
Comparator<Temperature> tCompare =
    ((t1, t2) -> t1.getTemp().compareTo(t2.getTemp()));
List<Temperature> sortedTemps =
    julyAvgs.stream()
        // L1
    sortedTemps.forEach(System.out::println);
```

Which fragment(s), inserted independently at // L1, cause the code to print the Temperatures from high temp to low temp?

A.

```
.sorted().reversed().collect(Collectors.toList());
```

B.

```
.sorted(Comparator.comparing(t -> t.getTemp())).collect(Collectors.toList());
```

C.

```
.sorted(tCompare).collect(Collectors.toList());
```

D.

```
.sorted(tCompare.reversed()).collect(Collectors.toList());
```

**15.** Given the Temperature class and julyAvgs List:

```
class Temperature {  
    String location;  
    Double temp;  
    public Temperature(String location, Double temp) {  
        this.location = location; this.temp = temp;  
    }  
    public String getLocation() { return this.location; }  
    public Double getTemp() { return this.temp; }  
}
```

```
public String toString() {
    return "July average temp at " + location + " was " + temp;
}
}

List<Temperature> julyAvgs = new ArrayList<Temperature>();
julyAvgs.add(new Temperature("Death Valley, CA", 107.4));
julyAvgs.add(new Temperature("Salt Lake City, UT", 85.3));
julyAvgs.add(new Temperature("Reno, NV", 80.5));
julyAvgs.add(new Temperature("Bishop, CA", 80.8));
julyAvgs.add(new Temperature("Phoenix, AZ", 106.0));
julyAvgs.add(new Temperature("Miami, FL", 85.7));
```

and the following code fragment:

```
julyAvgs.stream()
    .mapToLong(t -> Math.round(t.getTemp()))
    .sorted()
    .distinct()
    .forEach(t -> System.out.print(t + " "));
```

What does this code display?

- A. 80.5 80.8 85.3 85.7 106.0 107.4
- B. 81 81 85 86 106 107
- C. 81 85 86 106 107
- D. 107 85 81 81 106 86

**16.** Given the Temperature class and julyAvgs List:

```
class Temperature {
    String location;
    Double temp;
    public Temperature(String location, Double temp) {
        this.location = location; this.temp = temp;
    }
    public String getLocation() { return this.location; }
    public Double getTemp() { return this.temp; }
    public String toString() {
        return "July average temp at " + location + " was " + temp;
    }
}
List<Temperature> julyAvgs = new ArrayList<Temperature>();
julyAvgs.add(new Temperature("Death Valley, CA", 107.4));
julyAvgs.add(new Temperature("Salt Lake City, UT", 85.3));
julyAvgs.add(new Temperature("Reno, NV", 80.5));
julyAvgs.add(new Temperature("Bishop, CA", 80.8));
julyAvgs.add(new Temperature("Phoenix, AZ", 106.0));
julyAvgs.add(new Temperature("Miami, FL", 85.7));
```

What does the following code fragment display?

```
Map<Boolean, List<String>> temp100 =
    julyAvgs.stream().collect(
        Collectors.partitioningBy(t -> t.getTemp() >= 100.0,
            Collectors.mapping(t -> t.getLocation(),
                Collectors.toList())));
System.out.println(temp100);
```

A.

Salt Lake City, UT, Reno, NV, Bishop, CA, Miami, FL, Death Valley, CA, Phoenix, AZ

B.

{false=[Salt Lake City, UT, Reno, NV, Bishop, CA, Miami, FL], true=[Death Valley, CA, Phoenix, AZ]}

C.

106 107

D.

Death Valley, CA, Phoenix, AZ

**17.** Given the Temperature class and julyAvgs List:

```
class Temperature {
    String location;
    Double temp;
    public Temperature(String location, Double temp) {
        this.location = location; this.temp = temp;
    }
    public String getLocation() { return this.location; }
    public Double getTemp() { return this.temp; }
    public String toString() {
        return "July average temp at " + location + " was " + temp;
    }
}
List<Temperature> julyAvgs = new ArrayList<Temperature>();
julyAvgs.add(new Temperature("Death Valley, CA", 107.4));
julyAvgs.add(new Temperature("Salt Lake City, UT", 85.3));
julyAvgs.add(new Temperature("Reno, NV", 80.5));
julyAvgs.add(new Temperature("Bishop, CA", 80.8));
julyAvgs.add(new Temperature("Phoenix, AZ", 106.0));
julyAvgs.add(new Temperature("Miami, FL", 85.7));
```

Which of the following fragments will print Temperature objects sorted by location?

A.

```
julyAvgs.stream()  
    .map(t -> t.getLocation())  
    .sorted()  
    .forEach(System.out::println);
```

B.

```
julyAvgs.stream()  
    .sorted(Temperature::getLocation)  
    .forEach(System.out::println);
```

C.

```
julyAvgs.stream()  
    .sorted((t1, t2) -> t1.getLocation().compareTo(t2.getLocation()))  
    .forEach(System.out::println);
```

D.

```
julyAvgs.stream()  
    .map(t -> t.getLocation())  
    .sorted((l1, l2) -> l1.compareTo(l2))  
    .forEach(System.out::println);
```

# A SELF TEST ANSWERS

1.  **B** is correct. We provide an identity for `reduce()` so we get back an `int`, rather than an optional.

**A** is incorrect because `reduce()` needs an identity or it creates an `Optional`. **C** and **D** are incorrect syntax for `reduce()`, and both result in compile errors. (OCP Objective 3.4)
2.  **D** is correct. We call `sorted()` to sort the stream and `collect()` to collect the results in a `List`.

**A**, **B**, and **C** are incorrect. For **A**, `toList()` is a method of `Collectors`, not `Stream`. For **B**, `comparing()` is a method of `Comparator`, not `Stream` [you pass a `Comparator` to `sorted()`]. For **C**, `sorted()` is an intermediate stream operation that should be called before the `collect()` reduction, a terminal operation. (OCP Objectives 3.4, 3.6, 5.5, 5.6, and 9.3)
3.  **D** is correct. The `ints` in the call to `Stream.of()` get autoboxed to `Integers`. (If we'd used `IntStream.of()` instead, then we could use `IntStream` as the type for `s1`.) Given that `s1` is a `Stream<Integer>`, when we call `mapToInt()` on elements of the stream (which maps objects to `ints`), we are converting the `Integer` objects to `ints`, so the type of `s2` must be `IntStream`.

**A**, **B**, **C**, and **E** are incorrect based on the above information. (OCP Objectives 3.4 and 5.1)
4.  **A** and **C** are correct. **A** is a method reference shorthand for **C**.

**B** and **D** are incorrect. **B** has incorrect syntax for a lambda expression and method reference. **D** is not a lambda expression and should be. (OCP Objective 4.4)
5.  **A** is correct. Filter all ducks who have "mottled" color and map ducks to their name.

**B**, **C**, and **D** are incorrect. **B** shows all the ducks instead of just "mottled" ducks. **D** shows the result if you print the full `Duck` [via `toString()`] but you are mapping a duck to its name before printing. (OCP Objectives 3.4, 3.5, 3.7, and 5.1)
6.  **B** is correct. We stream the ducks and map each duck to its age (an `int`) and then reduce to the average of the ages. Note that we are using `mapToInt()` to map a duck to its age. `average()` returns an optional.

**A**, **C**, **D**, and **E** are incorrect. **A**, **C**, and **D** don't compile. `average()`

returns an `OptionalDouble`. In **D**, we have an additional problem with `map()` because the lambda maps a duck to its age (a primitive), but `map()` is used to map an object to an object. (OCP Objectives 3.4, 5.1, 5.3, and 5.4)

7.  **A** is correct. We stream the ducks and filter for ducks whose color is “mottled.” We call `count()` to reduce to the number of mottled ducks.  
 **B**, **C**, and **D** are incorrect. **B** doesn’t compile [`count()` returns a `long`]. In **C**, `d` in `filter` is a `Duck`, not a `String`, so it will not equal “mottled” and the count will be 0. **D** doesn’t compile because of invalid syntax for the expected `Predicate` in `filter()`. (OCP Objectives 3.4, 3.7, and 5.4)
8.  **D** is correct. We stream the ducks and group them by their color, creating a `Map`. We then use `forEach()` to take each map entry (ducks by their color) and display the `Map` key—the color—and the `Map` value, which is a list of `Ducks`. We further iterate through each list of ducks to display just the names.  
 **A**, **B**, and **C** are incorrect. **A** and **B** are incorrect based on the above information. **C** is incorrect because it shows the `Duck` name printed as a `List` rather than as `Strings` and shows the word “color” rather than the `color` value of each duck (the `Map` key for the `Map` produced by `Collectors.groupingBy()`). (OCP Objectives 5.1 and 5.6)
9.  **C** is correct. `Collectors.groupingBy()` creates a `Map` with `color` `Strings` as keys and `Lists` of `Ducks` as values.  
 **A**, **B**, and **D** are incorrect based on the above information. (OCP Objectives 3.4 and 5.6)
10.  **C** is correct. `min()` is a reduction method that produces an optional type. We start with a `Stream<Integer>`; `mapToInt()` creates an `IntStream`, so `min()` produces an `OptionalInt`.  
 **A**, **B**, and **D** are incorrect based on the above information. (OCP Objectives 3.4, 5.1, and 5.4)
11.  **B** is correct. `max()` takes a `Comparator`, which takes two `Temperature` objects and compares them using the `temp` field, producing the maximum `Temperature` object in the `List`.  
 **A**, **C**, **D**, **E**, and **F** are incorrect based on the above information. (OCP Objectives 3.4, 5.3, and 5.4)
12.  **C** is correct. We map each `Temperature` object to its (unboxed) `double`

temp and take the `max()` of the stream of doubles, producing an `OptionalDouble`.

**A, B, D, E, and F** are incorrect. (OCP Objectives 3.4, 5.1, 5.3, and 5.4)

- 13.**  **C** is correct. The minimum is 80.5 in Reno, Nevada. `findAny()` finds any Temperature from the filtered stream of Temperatures  $< 100$ ; for a nonparallel stream, it's usually the first one, 85.3 in Salt Lake City, but there is no guarantee that `findAny()` will return the first Temperature in the filtered stream.
- A** and **B** are incorrect based on the above information. (OCP Objectives 3.4, 3.7, 5.1, 5.2, 5.3, and 5.4)
- 14.**  **D** is correct. We stream the Temperatures using the `tCompare` Comparator, which sorts the Temperatures from low to high based on the temp value, so we then reverse that ordering with `reversed()`. Then we collect the results in a `List`.
- A, B, and C** are incorrect. **A** has two problems: Temperature isn't Comparable, and `reversed()` is a method on Comparators, so this code will not compile. **B** and **C** work but print the temperatures from low to high instead of high to low. (OCP Objectives 3.4, 5.5, and 5.6)
- 15.**  **C** is correct. We stream the Temperatures and map each temperature to a long value by rounding the Temperature's temp value. We then remove any duplicates by calling `distinct()` (eliminating the repeated 81 value) and then print them out.
- A, B, and D** are incorrect. For **A**, notice in the code fragment that we are using `Math.round()` to cut off the decimal points (so we process longs, not doubles). We are using `distinct()` to eliminate duplicates, so we shouldn't see 81 twice as we do in **B**. **D** is sorted in the wrong order; natural order is ascending for long values. (OCP Objectives 3.4, 3.5, 3.6, 5.1, and 5.5)
- 16.**  **B** is correct. We are creating a partition, so we will get (and display) a Map with two keys, `true` and `false`, partitioned by whether the temp value of a Temperature is  $\geq 100.0$ . Each Temperature is mapped (downstream) to its location and the locations for each partition are collected into a `List`. The locations are `Strings`, so we see a `List` of `String` locations for each partition.
- A, C, and D** are incorrect based on the above information. (OCP Objectives 3.4 and 5.6)

17.  **C** is correct. We sort the Temperature objects by location and print. `sorted()` takes a Comparator.
- A, B, and D** are incorrect. **A** and **D** display only locations, not the whole Temperature object. **B** produces a compile error because `sorted()` requires a Comparator, and we've provided a Function. (OCP Objectives 3.4, 3.8, 5.1, and 5.5)

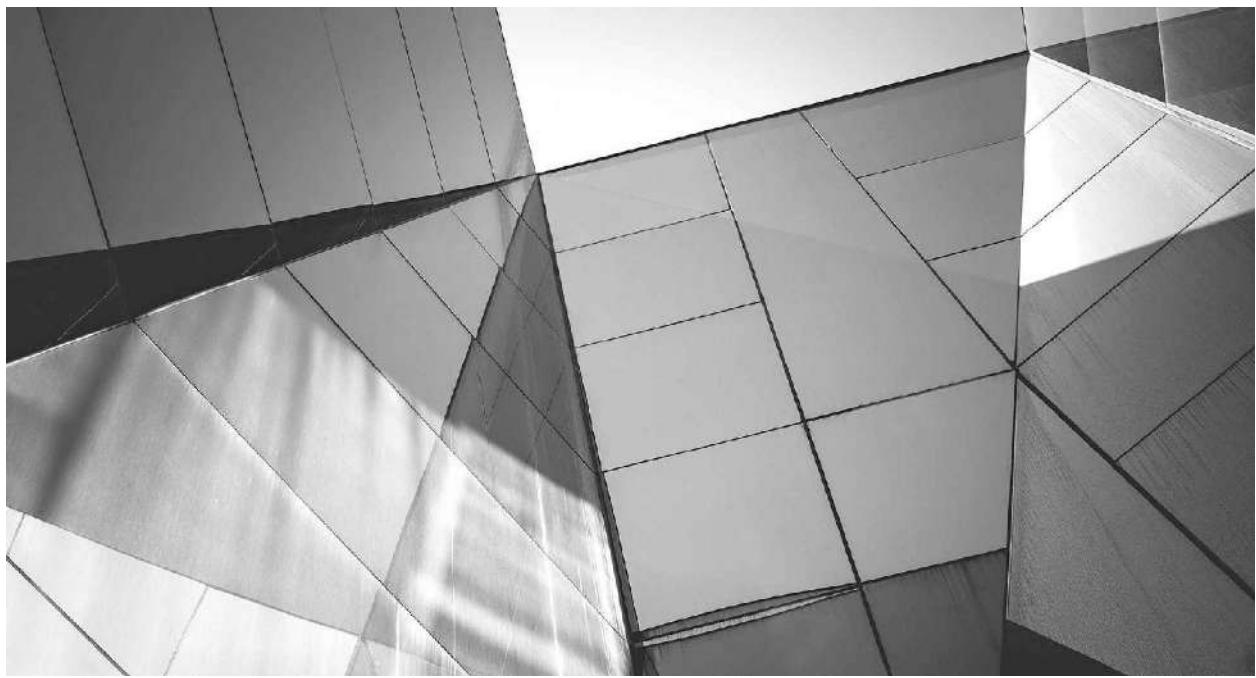
## EXERCISE ANSWER

### Exercise 9-1: Collecting Items in a List

Using `collect()` to collect DVDs into a `List` of `DVDInfo` objects as we read the dvds from a file, we can rewrite our `loadDVDs()` method as shown on the next page.

```
public static List<DVDInfo> loadDVDs(String filename) {  
    List<DVDInfo> dvds = new ArrayList<DVDInfo>();  
    try (Stream<String> stream = Files.lines(Paths.get(filename))) {  
        dvds = stream.map(line -> {  
            String[] dvdItems = line.split("/");  
            DVDInfo dvd =  
                new DVDInfo(dvdItems[0], dvdItems[1], dvdItems[2]);  
            return dvd;  
        }).collect(Collectors.toList());  
    } catch (IOException e) {  
        System.out.println("Error reading DVDs");  
        e.printStackTrace();  
    }  
    return dvds;  
}
```

This solution has the benefit of being more clear and avoids the side effect of modifying an object that's defined outside the stream pipeline.



# 10

## Threads

### CERTIFICATION OBJECTIVES

- Create Worker Threads Using Runnable, Callable, and Use an ExecutorService to Concurrently Execute Tasks
- Identify Potential Threading Problems among Deadlock, Starvation, Livelock, and Race Conditions
- Use Synchronized keyword and java.util.concurrent.atomic Package to Control the Order of Thread Execution



Two-Minute Drill

**Q&A** Self Test

### CERTIFICATION OBJECTIVE

## Defining, Instantiating, and Starting Threads (OCP Objective 10.1)

*10.1 Create worker threads using Runnable, Callable, and use an ExecutorService to concurrently execute tasks.*

Imagine a stockbroker application with a lot of complex capabilities. One of its functions is “download last stock option prices,” another is “check prices for warnings,” and a third time-consuming operation is “analyze historical data for company XYZ.”

In a single-threaded runtime environment, these actions execute one after another. The next action can happen *only* when the previous one is finished. If a

historical analysis takes half an hour, and the user selects to perform a download and check afterward, the warning may come too late to, say, buy or sell stock as a result.

We just imagined the sort of application that cries out for multithreading. Ideally, the download should happen in the background (that is, in another thread). That way, other processes could happen at the same time so that, for example, a warning could be communicated instantly. All the while, the user is interacting with other parts of the application. The analysis, too, could happen in a separate thread so the user can work in the rest of the application while the results are being calculated.

So what exactly is a thread? In Java, “thread” means two different things:

- An instance of class `java.lang.Thread`
- A thread of execution

An instance of `Thread` is just...an object. Like any other object in Java, it has variables and methods, and it lives and dies on the heap. But a *thread of execution* is an individual process (a “lightweight” process) that has its own call stack. In Java, there is *one thread per call stack*—or, to think of it in reverse, *one call stack per thread*. Even if you don’t create any new threads in your program, threads are back there running.

The `main()` method, which starts the whole ball rolling, runs in one thread, called (surprisingly) the *main* thread. If you looked at the main call stack (and you can any time you get a stack trace from something that happens after `main` begins, but not within another thread), you’d see that `main()` is the first method on the stack—the method at the bottom. But as soon as you create a *new* thread, a new stack materializes and methods called from *that* thread run in a call stack that’s separate from the `main()` call stack. That second new call stack is said to run concurrently with the main thread, but we’ll refine that notion as we go through this chapter.

You might find it confusing that we’re talking about code running *concurrently*—what gives? The JVM, which gets its turn at the CPU by whatever scheduling mechanism the underlying OS uses, operates like a mini-OS and schedules *its* own threads, regardless of the underlying operating system. In some JVMs, the Java threads are actually mapped to native OS threads, but we won’t discuss that here; native threads are not on the exam. Nor is it required to understand how threads behave in different JVM environments. In fact, the most important concept to understand from this entire chapter is this:

When it comes to threads, very little is guaranteed.

So be very cautious about interpreting the behavior you see on *one* machine as “the way threads work.” The exam expects you to know what is and is not guaranteed behavior so that you can design your program in such a way that it will work, regardless of the underlying JVM. *That’s part of the whole point of Java.*



***Don’t make the mistake of designing your program to be dependent on a particular implementation of the JVM. As you’ll learn a little later, different JVMs can run threads in profoundly different ways. For example, one JVM might be sure that all threads get their turn, with a fairly even amount of time allocated for each thread in a nice, happy, round-robin fashion. But in other JVMs, a thread might start running and then just hog the whole show, never stepping out so others can have a turn. If you test your application on the “nice turn-taking” JVM and you don’t know what is and is not guaranteed in Java, then you might be in for a big shock when you run it under a JVM with a different thread-scheduling mechanism.***

The thread questions are among the most difficult questions on the exam. In fact, for most people, they *are* the toughest questions on the exam, and with three objectives for threads, you’ll be answering a *lot* of thread questions. If you’re not already familiar with threads, you’ll probably need to spend some time experimenting. Also, one final disclaimer: *This chapter makes almost no attempt to teach you how to design a good, safe multithreaded application. We only scratch the surface of that huge topic in this chapter!* You’re here to learn the basics of threading and what you need to get through the thread questions on the exam. Before you can write decent multithreaded code, however, you really need to study more of the complexities and subtleties of multithreaded code.

Note: The topic of daemon threads is NOT on the exam. All of the threads discussed in this chapter are “user” threads. You and the operating system can create a second kind of thread called a daemon thread. The difference between these two types of threads (user and daemon) is that the JVM exits an application only when all user threads are complete—the JVM doesn’t care about letting daemon threads complete, so once all user threads are complete, the JVM will

shut down, regardless of the state of any daemon threads. Once again, this topic is NOT on the exam.

## Making a Thread

A thread in Java begins as an instance of `java.lang.Thread`. You'll find methods in the `Thread` class for managing threads, including creating, starting, and pausing them. For the exam, you'll need to know, at a minimum, the following methods:

```
start()  
yield()  
sleep()  
run()
```

The action happens in the `run()` method. Think of the code you want to execute in a separate thread as *the job to do*. In other words, you have some work that needs to be done—say, downloading stock prices in the background while other things are happening in the program—so what you really want is that *job* to be executed in its own thread. So, if the *work* you want done is the *job*, the one *doing* the work (actually executing the job code) is the *thread*. And the *job always starts from a run() method*, as follows:

```
public void run() {  
    // your job code goes here  
}
```

You always write the code that needs to be run in a separate thread in a `run()` method. The `run()` method will call other methods, of course, but the thread of execution—the new call stack—always begins by invoking `run()`. So where does the `run()` method go? In one of the two classes you can use to define your thread job.

You can define and instantiate a thread in one of two ways:

- Extend the `java.lang.Thread` class.
- Implement the `Runnable` interface.

You need to know about both for the exam, although in the real world, you're much more likely to implement `Runnable` than extend `Thread`. Extending the `Thread` class is the easiest, but it's usually not good OO practice. Why? Because subclassing should be reserved for specialized versions of more general superclasses. So the only time it really makes sense (from an OO perspective) to extend `Thread` is when you have a more specialized version of a `Thread` class. In other words, because *you have more specialized thread-specific behavior*. Chances are, though, that the thread work you want is really just a job to be done *by* a thread. In that case, you should design a class that implements the `Runnable` interface, which also leaves your class free to extend some *other* class.

## Defining a Thread

To define a thread, you need a place to put your `run()` method, and as we just discussed, you can do that by extending the `Thread` class or by implementing the `Runnable` interface. We'll look at both in this section.

### Extending `java.lang.Thread`

The simplest way to define code to run in a separate thread is to

- Extend the `java.lang.Thread` class.
- Override the `run()` method.

It looks like this:

```
class MyThread extends Thread {  
    public void run() {  
        System.out.println("Important job running in MyThread");  
    }  
}
```

The limitation with this approach (besides being a poor design choice in most cases) is that if you extend `Thread`, *you can't extend anything else*. And it's not as if you really need that inherited `Thread` class behavior; because in order to use a thread, you'll need to instantiate one anyway.

Keep in mind that you're free to overload the `run()` method in your `Thread`

subclass:

```
class MyThread extends Thread {  
    public void run() {  
        System.out.println("Important job running in MyThread");  
    }  
    public void run(String s) {  
        System.out.println("String in run is " + s);  
    }  
}
```

But know this: The overloaded `run(String s)` method will be ignored by the `Thread` class unless you call it yourself. The `Thread` class expects a `run()` method with no arguments, and it will execute this method for you in a separate call stack after the thread has been started. With a `run(String s)` method, the `Thread` class won't call the method for you, and even if you call the method directly yourself, execution won't happen in a new thread of execution with a separate call stack. It will just happen in the same call stack as the code that you made the call from, just like any other normal method call.

## Implementing `java.lang.Runnable`

Implementing the `Runnable` interface gives you a way to extend any class you like but still define behavior that will be run by a separate thread. It looks like this:

```
class MyRunnable implements Runnable {  
    public void run() {  
        System.out.println("Important job running in MyRunnable");  
    }  
}
```

Notice that the `Runnable` interface is a functional interface, that is, an interface with one abstract method, `run()`. That means we can also write a

Runnable like this:

```
Runnable r = () ->
    System.out.println("Important job running in a Runnable");
```

Regardless of which mechanism you choose, you've now got yourself some code that can be run by a thread of execution. Let's take a look at *instantiating* your thread-capable class, and then we'll figure out how to actually get the thing *running*.

## Instantiating a Thread

Remember, every thread of execution begins as an instance of class `Thread`. Regardless of whether your `run()` method is in a `Thread` subclass or a `Runnable` implementation class, you still need a `Thread` object to do the work.

If you extended the `Thread` class, instantiation is dead simple (we'll look at some additional overloaded constructors in a moment):

```
MyThread t = new MyThread();
```

If you implement `Runnable`, instantiation is only slightly less simple. To have code run by a separate thread, *you still need a Thread instance*. But rather than combining both the *thread* and the *job* (the code in the `run()` method) into one class, you've split it into two classes—the `Thread` class for the *thread-specific* code and your `Runnable` implementation class for your *job-that-should-be-run-by-a-thread* code. (Another common way to think about this is that the `Thread` is the “worker,” and the `Runnable` is the “job” to be done.)

First, you instantiate your `Runnable` class:

```
MyRunnable r = new MyRunnable();
```

Next, you get yourself an instance of `java.lang.Thread` (*somebody* has to run your job...), and you *give it your job!*

```
Thread t = new Thread(r); // Pass your Runnable to the Thread
```

Or, if you want to use a lambda expression, you can eliminate `MyRunnable` and write:

```
Thread t = new Thread(  
    () -> System.out.println("Important job running in a Runnable")  
);
```

If you create a thread using the no-arg constructor, the thread will call its own `run()` method when it's time to start working. That's exactly what you want when you extend `Thread`, but when you use `Runnable`, you need to tell the new thread to use *your* `run()` method rather than its own. The `Runnable` you pass to the `Thread` constructor is called the *target* or the *target Runnable*.

You can pass a single `Runnable` instance to multiple `Thread` objects so that the same `Runnable` becomes the target of multiple threads, as follows:

```
public class TestThreads {  
    public static void main (String [] args) {  
        // MyRunnable r = new MyRunnable(); // OR get rid of MyRunnable and write:  
        Runnable r = () ->  
            System.out.println("Important job running in a Runnable");  
        Thread foo = new Thread(r);  
        Thread bar = new Thread(r);  
        Thread bat = new Thread(r);  
    }  
}
```

Giving the same target to multiple threads means that several threads of execution will be running the very same job (and that the same job will be done multiple times).



***The `Thread` class itself implements `Runnable`. (After all, it has a `run()` method that we were overriding.) This means that you could pass a `Thread`***

**to another Thread's constructor:**

```
Thread t = new Thread(new MyThread());
```

***This is a bit silly, but it's legal. In this case, you really just need a Runnable, and creating a whole other Thread is overkill.***

Besides the no-arg constructor and the constructor that takes a Runnable (the target, i.e., the instance with the job to do), there are other overloaded constructors in class Thread. The constructors we care about are

- Thread()
- Thread(Runnable target)
- Thread(Runnable target, String name)
- Thread(String name)

You need to recognize all of them for the exam! A little later, we'll discuss some of the other constructors in the preceding list.

So now you've made yourself a Thread instance, and it knows which run() method to call. *But nothing is happening yet.* At this point, all we've got is a plain-old Java object of type Thread. *It is not yet a thread of execution.* To get an actual thread—a new call stack—we still have to *start* the thread.

When a thread has been instantiated but not started (in other words, the start() method has not been invoked on the Thread instance), the thread is said to be in the *new* state. At this stage, the thread is not yet considered *alive*. Once the start() method is called, the thread is considered *alive* (even though the run() method may not have actually started executing yet). A thread is considered *dead* (no longer *alive*) after the run() method completes. The isAlive() method is the best way to determine if a thread has been started but has not yet completed its run() method. (Note: The getState() method is very useful for debugging, but you don't have to know it for the exam.)

## Starting a Thread

You've created a Thread object and it knows its target (either the passed-in Runnable or itself, if you extended class Thread). Now it's time to get the whole thread thing happening—to launch a new call stack. It's so simple; it hardly deserves its own subheading:

```
t.start();
```

Prior to calling `start()` on a `Thread` instance, the thread (when we use lowercase `t`, we're referring to the *thread of execution* rather than the `Thread` class) is said to be in the *new* state, as we said. The new state means you have a `Thread object` but you don't yet have a *true thread*. So what happens after you call `start()`? The good stuff:

- A new thread of execution starts (with a new call stack).
- The thread moves from the *new* state to the *Runnable* state.
- When the thread gets a chance to execute, its target `run()` method will run.

Be *sure* you remember the following: You start a `Thread`, not a `Runnable`. You call `start()` on a `Thread` instance, not on a `Runnable` instance. The following example demonstrates what we've covered so far—defining, instantiating, and starting a thread:

```
public class TestThreads {  
    public static void main (String [] args) {  
        Runnable r = () -> {  
            for (int x = 1; x < 6; x++) {  
                System.out.println("Runnable running " + x);  
            }  
        };  
        Thread t = new Thread(r);  
        t.start();  
    }  
}
```

Running the preceding code prints out exactly what you'd expect:

```
% java TestThreads
Runnable running 1
Runnable running 2
Runnable running 3
Runnable running 4
Runnable running 5
```

(If this isn't what you expected, go back and reread everything in this objective.)



*There's nothing special about the `run()` method as far as Java is concerned. Like `main()`, it just happens to be the name (and signature) of the method that the new thread knows to invoke. So if you see code that calls the `run()` method on a `Runnable` (or even on a `Thread` instance), that's perfectly legal. But it doesn't mean the `run()` method will run in a separate thread! Calling a `run()` method directly just means you're invoking a method from whatever thread is currently executing, and the `run()` method goes onto the current call stack rather than at the beginning of a new call stack. The following code does not start a new thread of execution:*

```
Thread t = new Thread();
t.run(); // Legal, but does not start a new thread
```

So what happens if we start multiple threads? We'll run a simple example in a moment, but first we need to know how to print out which thread is executing. We can use the `getName()` method of class `Thread` and have each `Runnable` print out the name of the thread executing that `Runnable` object's `run()` method. The following example instantiates a thread and gives it a name, and then the name is printed out from the `run()` method:

```
class NameRunnable implements Runnable {  
    public void run() {  
        System.out.println("NameRunnable running");  
        System.out.println("Run by "  
            + Thread.currentThread().getName());  
    }  
}  
public class NameThread {  
    public static void main (String [] args) {  
        NameRunnable nr = new NameRunnable();  
        Thread t = new Thread(nr);  
        t.setName ("Fred");  
        t.start();  
    }  
}
```

Running this code produces the following extra-special output:

```
% java NameThread  
NameRunnable running  
Run by Fred
```

To get the name of a thread, you call—who would have guessed—`getName()` on the `Thread` instance. But the target `Runnable` instance doesn’t even *have* a reference to the `Thread` instance, so we first invoked the static `Thread.currentThread()` method, which returns a reference to the currently executing thread, and then we invoked `getName()` on that returned reference.

Even if you don’t explicitly name a thread, it still has a name. Let’s look at the previous code, commenting out the statement that sets the thread’s name:

```

public class NameThread {
    public static void main (String [] args) {
        NameRunnable nr = new NameRunnable();
        Thread t = new Thread(nr);
        // t.setName("Fred");
        t.start();
    }
}

```

Running the preceding code now gives us

```

% java NameThread
NameRunnable running
Run by Thread-0

```

And since we're getting the current thread by using the static `Thread.currentThread()` method, we can even get the name of the thread running our main code:

```

public class NameThreadTwo {
    public static void main (String [] args) {
        System.out.println("thread is "
            + Thread.currentThread().getName());
    }
}

```

which prints out

```

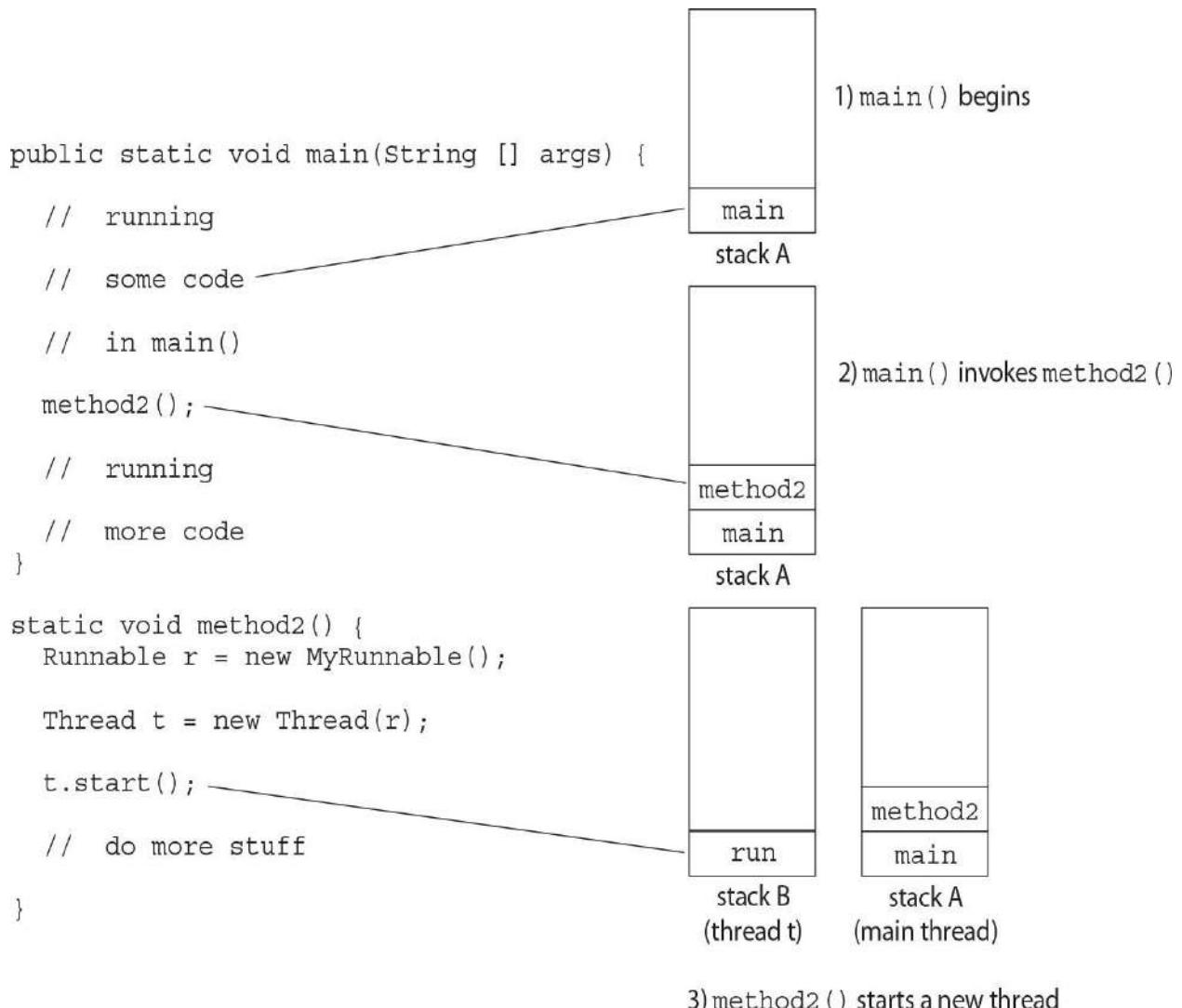
% java NameThreadTwo
thread is main

```

That's right, the main thread already has a name—*main*. (Once again, what are the odds?) [Figure 10-1](#) shows the process of starting a thread.

## **FIGURE 10-1**

Starting a thread



## Starting and Running Multiple Threads

Enough playing around here; let's actually get multiple threads going (more than two, that is). We already had two threads, because the `main()` method starts in a thread of its own, and then `t.start()` started a *second* thread. Now we'll do more. The following code creates a single `Runnable` instance and three `Thread` instances. All three `Thread` instances get the same `Runnable` instance, and each thread is given a unique name. Finally, all three threads are started by invoking `start()` on the `Thread` instances. And just to hammer in one more time how you can use a lambda expression in place of an explicit `Runnable` class, we'll eliminate `NameRunnable`, and replace it with a lambda.

```

// No need for NameRunnable if we use lambdas...
public class ManyNames {
    public static void main(String [] args) {

```

```

// Make one Runnable
Runnable nr = () -> {
    for (int x = 1; x <= 3; x++) {
        System.out.println("Run by " +
            Thread.currentThread().getName() + ", x is " + x);
    }
};

Thread one = new Thread(nr);
Thread two = new Thread(nr);
Thread three = new Thread(nr);

one.setName("Fred");
two.setName("Lucy");
three.setName("Ricky");
one.start();
two.start();
three.start();
}
}

```

Running this code **might** produce the following:

```
% java ManyNames
Run by Fred, x is 1
Run by Fred, x is 2
Run by Fred, x is 3
Run by Lucy, x is 1
Run by Lucy, x is 2
Run by Lucy, x is 3
Run by Ricky, x is 1
Run by Ricky, x is 2
Run by Ricky, x is 3
```

Well, at least that's what it printed when we ran it—this time, on our machine. But the behavior you see here is not guaranteed. This is so crucial that you need to stop right now, take a deep breath, and repeat after me, “The behavior is not guaranteed.” You need to know, for your future as a Java programmer as well as for the exam, that there is nothing in the Java specification that says threads will start running in the order in which they were started (in other words, the order in which `start()` was invoked on each thread). And there is no guarantee that once a thread starts executing, it will keep executing until it's done. Or that a loop will complete before another thread begins. No siree, Bob.

**Nothing is guaranteed in the preceding code except this:**

## Each thread will start, and each thread will run to completion.

Within each thread, things will happen in a predictable order. But the actions of different threads can mix in unpredictable ways. If you run the program multiple times or on multiple machines, you may see different output. Even if you don't see different output, you need to realize that the behavior you see is not guaranteed. Sometimes a little change in the way the program is run will cause a difference to emerge. Just for fun we bumped up the loop code so that each `run()` method ran the `for` loop 400 times rather than 3, and eventually we did start to see some wobbling:

```
Runnable nr = () -> {
    for (int x = 1; x <= 400; x++) {
        System.out.println("Run by " +
            Thread.currentThread().getName() + ", x is " + x);
    }
};
```

Running the preceding code, with each thread executing its run loop 400 times, started out fine but then became nonlinear. Here's just a snippet from the command-line output of running that code. To make it easier to distinguish each thread, we put Fred's output in italics and Lucy's in bold and left Ricky's alone:

```
Run by Fred, x is 345
Run by Ricky, x is 313
Run by Lucy, x is 341
Run by Ricky, x is 314
Run by Lucy, x is 342
Run by Ricky, x is 315
Run by Fred, x is 346
Run by Lucy, x is 343
Run by Fred, x is 347
Run by Lucy, x is 344
```

...it continues on...

Notice that there's not really any clear pattern here. If we look at only the output from Fred, we see the numbers increasing one at a time, as expected:

```
Run by Fred, x is 345
Run by Fred, x is 346
Run by Fred, x is 347
```

And similarly, if we look only at the output from Lucy or Ricky—each one

individually is behaving in a nice, orderly manner. But together—chaos! In the previous fragment we see Fred, then Lucy, then Ricky (in the same order we originally started the threads), but then Lucy butts in when it was Fred’s turn. What nerve! And then Ricky and Lucy trade back and forth for a while until finally Fred gets another chance. They jump around like this for a while after this. Eventually (after the part shown earlier), Fred finishes, then Ricky, and finally Lucy finishes with a long sequence of output. So even though Ricky was started third, he actually completed second. And if we run it again, we’ll get a different result. Why? Because it’s up to the scheduler, and we don’t control the scheduler! Which brings up another key point to remember: Just because a series of threads are started in a particular order doesn’t mean they’ll run in that order. For any group of started threads, order is not guaranteed by the scheduler. And duration is not guaranteed. You don’t know, for example, if one thread will run to completion before the others have a chance to get in, or whether they’ll all take turns nicely, or whether they’ll do a combination of both. There is a way, however, to start a thread but tell it not to run until some other thread has finished. You can do this with the `join()` method, which we’ll look at a little later.

***A thread is done being a thread when its target `run()` method completes.***

When a thread completes its `run()` method, the thread ceases to be a thread of execution. The stack for that thread dissolves, and the thread is considered dead. (Technically, the API calls a dead thread “terminated,” but we’ll use “dead” in this chapter.) Not dead and gone, however—just dead. It’s still a `Thread object`, just not a *thread of execution*. So if you’ve got a reference to a `Thread` instance, then even when that `Thread` instance is no longer a thread of execution, you can still call methods on the `Thread` instance, just like any other Java object. What you can’t do, though, is call `start()` again.

***Once a thread has been started, it can never be started again.***

If you have a reference to a `Thread` and you call `start()`, it’s started. If you call `start()` a second time, it will cause an exception (an `IllegalThreadStateException`, which is a kind of `RuntimeException`, but you don’t need to worry about the exact type). This happens whether or not the `run()` method has completed from the first `start()` call. Only a new thread can be started, and then only once. A runnable thread or a dead thread cannot be restarted.

So far, we've seen three thread states: *new*, *runnable*, and *dead*. We'll look at more thread states before we're done with this chapter.



***In addition to using `setName()` and `getName` to identify threads, you might see `getId()`. The `getId()` method returns a positive, unique long number, and that number will be that thread's only ID number for the thread's entire life.***

## The Thread Scheduler

The thread scheduler is the part of the JVM (although most JVMs map Java threads directly to native threads on the underlying OS) that decides which thread should run at any given moment and also takes threads *out* of the run state. Assuming a single processor machine, only one thread can actually *run* at a time. Only one stack can ever be executing at one time. And it's the thread scheduler that decides *which* thread—of all that are eligible—will actually *run*. When we say *eligible*, we really mean *in the runnable state*.

Any thread in the *runnable* state can be chosen by the scheduler to be the one and only running thread. If a thread is not in a runnable state, then it cannot be chosen to be the *currently running* thread. And just so we're clear about how little is guaranteed here:

***The order in which runnable threads are chosen to run is not guaranteed.***

Although *queue* behavior is typical, it isn't guaranteed. Queue behavior means that when a thread has finished with its "turn," it moves to the end of the line of the runnable pool and waits until it eventually gets to the front of the line, where it can be chosen again. In fact, we call it a *runnable pool*, rather than a *runnable queue*, to help reinforce the fact that threads aren't all lined up in some guaranteed order.

Although we don't *control* the thread scheduler (we can't, for example, tell a specific thread to run), we can sometimes influence it. The following methods give us some tools for *influencing* the scheduler. Just don't ever mistake influence for control.



***Expect to see exam questions that look for your understanding of what is and is not guaranteed! You must be able to look at thread code and determine whether the output is guaranteed to run in a particular way or is unpredictable.***

**Methods from the `java.lang.Thread` Class** Some of the methods that can help us influence thread scheduling are as follows:

```
public static void sleep(long millis) throws InterruptedException  
public static void yield()  
public final void join() throws InterruptedException  
public final void setPriority(int newPriority)
```

Note that both `sleep()` and `join()` have overloaded versions not shown here.

**Methods from the `java.lang.Object` Class** Every class in Java inherits the following three thread-related methods:

```
public final void wait() throws InterruptedException  
public final void notify()  
public final void notifyAll()
```

The `wait()` method has three overloaded versions (including the one listed here).

We'll look at the behavior of each of these methods in this chapter. First, though, we're going to look at the different states a thread can be in.

## Thread States and Transitions

We've already seen three thread states—*new*, *Runnable*, and *dead*—but wait! There's more! The thread scheduler's job is to move threads in and out of the