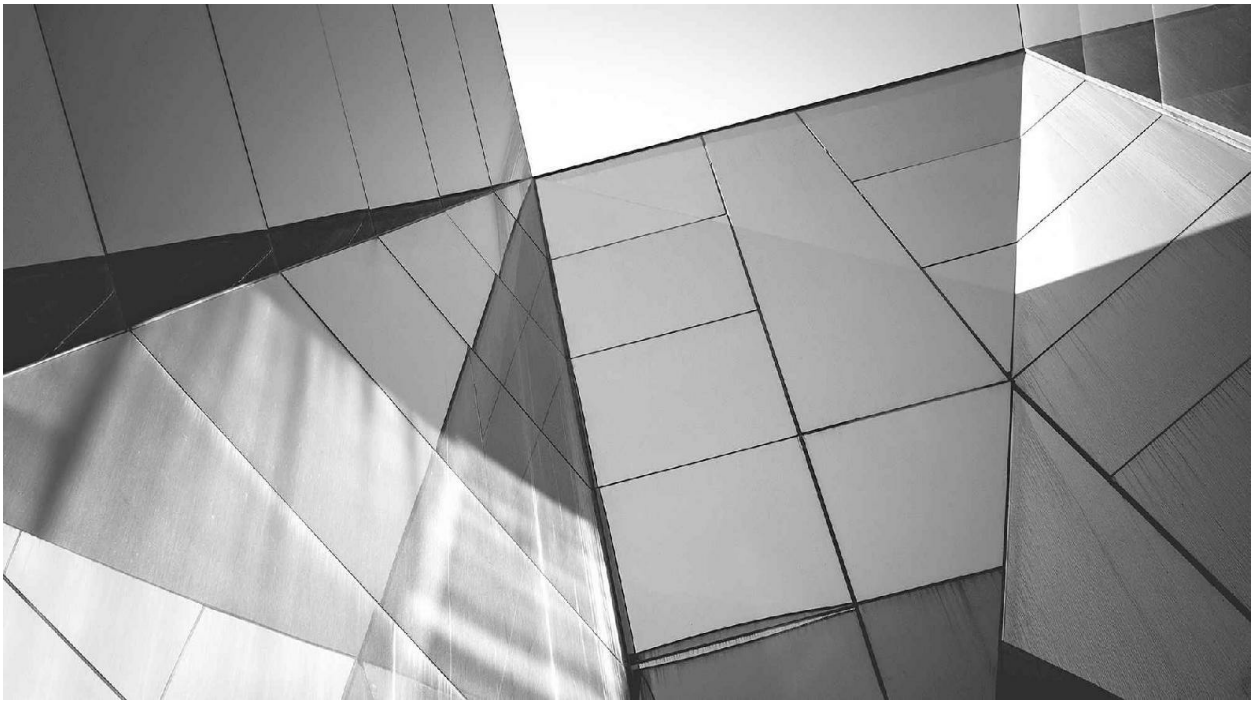

7

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 and outers?) 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—whoa!—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
```

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());
    } };
  dvdlist.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 `dvdlist 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.

SELF TEST

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)