Chapter 7

Beyond Classes

OCP EXAM OBJECTIVES COVERED IN THIS CHAPTER:

✓ **Utilizing Java Object-Oriented Approach**

- Declare and instantiate Java objects including nested class objects, and explain the object life-cycle including creation, reassigning references, and garbage collection
- Create classes and records, and define and use instance and static fields and methods, constructors, and instance and static initializers
- Understand variable scopes, use local variable type inference, apply encapsulation, and make objects immutable
- Implement polymorphism and differentiate object type versus reference type. Perform type casting, identify object types using instanceof operator and pattern matching
- Create and use interfaces, identify functional interfaces, and utilize private, static, and default interface methods
- Create and use enumerations with fields, methods and constructors

In Chapter 6, "Class Design," we showed you how to create, initialize, and extend both abstract and concrete classes. In this chapter, we move beyond classes to other types available in Java, including interfaces, enums, sealed classes, and records. Many of the same basic rules you learned about in Chapter 5, "Methods," still apply, such as access modifiers and `static` members, although there are additional rules for each type. We also cover encapsulation and how to properly protect instance members. Finally, we conclude this chapter by discussing nested types and polymorphic inheritance.

For this chapter, remember that a Java file may have at most one `public` top-level type, and it must match the name of the file. This applies to classes, enums, records, and so on. Also, remember that a top-level type can only be declared with `public` or package access.

Another top-level type available in Java is annotations. Knowing how to create a custom annotation can be a useful skill in practice, although it is not required for the exam. You should still know how to use certain annotations for the exam, such as `@Override`.

Implementing Interfaces

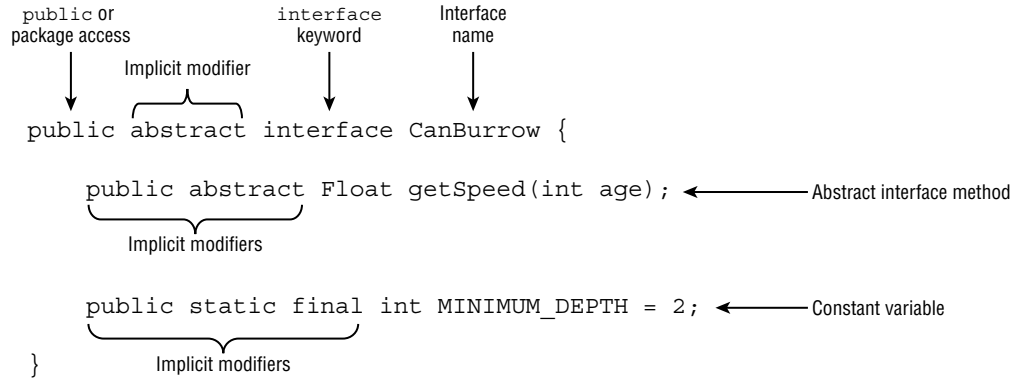
In Chapter 6, you learned about abstract classes, specifically how to create and extend one. Since classes can only extend one class, they had limited use for inheritance. On the other hand, a class may implement any number of interfaces. An *interface* is an abstract data type that declares a list of abstract methods that any class implementing the interface must provide.

Over time, the precise definition of an interface has changed, as new method types are now supported. In this chapter, we start with a rudimentary definition of an interface and expand it to cover all of the supported members.

Declaring and Using an Interface

In Java, an interface is defined with the `interface` keyword, analogous to the `class` keyword used when defining a class. Refer to Figure 7.1 for a proper interface declaration.

FIGURE 7.1 Defining an interface



In Figure 7.1, our interface declaration includes an abstract method and a constant variable. Interface variables are referred to as constants because they are assumed to be `public`, `static`, and `final`. They are initialized with a constant value when they are declared. Since they are `public` and `static`, they can be used outside the interface declaration without requiring an instance of the interface. Figure 7.1 also includes an abstract method that, like an interface variable, is assumed to be `public`.

For brevity, we often say "an instance of an interface" in this chapter to mean an instance of a class that implements the interface.

What does it mean for a variable or method to be assumed to be something? One aspect of an interface declaration that differs from an abstract class is that it contains implicit modifiers. An *implicit modifier* is a modifier that the compiler automatically inserts into the code. For example, an interface is always considered to be `abstract`, even if it is not marked so. We cover rules and examples for implicit modifiers in more detail shortly.

Let's start with a simple example. Imagine that we have an interface `WalksOnTwoLegs`, defined as follows:

```
public abstract interface WalksOnTwoLegs {}
```

It compiles because interfaces are not required to define any methods. The `abstract` modifier in this example is optional for interfaces, with the compiler inserting it if it is not provided. Now, consider the following two examples, which do not compile:

```
public class Biped {
   public static void main(String[] args) {
      var e = new WalksOnTwoLegs();         // DOES NOT COMPILE
   }
}

public final interface WalksOnEightLegs {}  // DOES NOT COMPILE
```

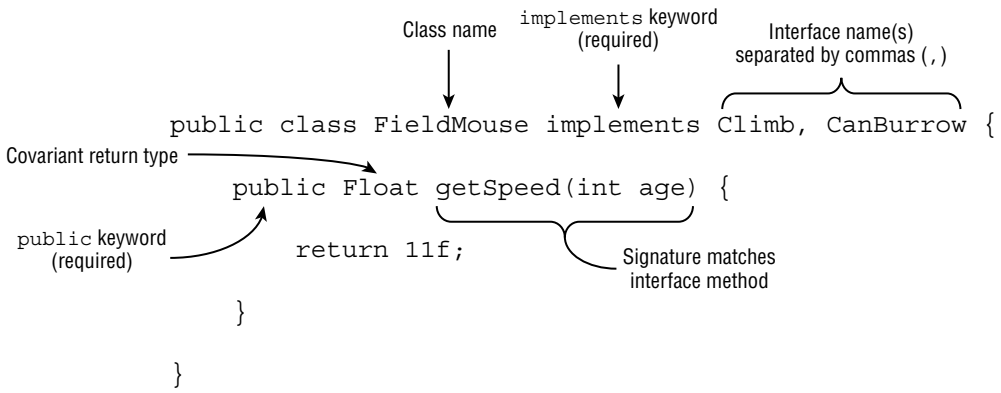
The first example doesn't compile, as `WalksOnTwoLegs` is an interface and cannot be instantiated. The second example, `WalksOnEightLegs`, doesn't compile because interfaces cannot be marked as `final` for the same reason that `abstract` classes cannot be marked as `final`. In other words, marking an interface `final` implies no class could ever implement it.

How do you use an interface? Let's say we have an interface `Climb`, defined as follows:

```
public interface Climb {
   Number getSpeed(int age);
}
```

Next, we have a concrete class `FieldMouse` that invokes the `Climb` interface by using the `implements` keyword in its class declaration, as shown in Figure 7.2.

FIGURE 7.2 Implementing an interface



The `FieldMouse` class declares that it implements the `Climb` interface and includes an overridden version of `getSpeed()` inherited from the `Climb` interface. The method signature of `getSpeed()` matches exactly, and the return type is covariant, since a `Float` can be implicitly cast to a `Number`. The access modifier of the interface method is implicitly `public` in `Climb`, although the concrete class `FieldMouse` must explicitly declare it.

As shown in Figure 7.2, a class can implement multiple interfaces, each separated by a comma (`,`). If any of the interfaces define abstract methods, then the concrete class is required to override them. In this case, `FieldMouse` implements the `CanBurrow` interface that we saw in Figure 7.1. In this manner, the class overrides two abstract methods at the same time with one method declaration. You learn more about duplicate and compatible interface methods in this chapter.

Extending an Interface

Like a class, an interface can extend another interface using the `extends` keyword.

```
public interface Nocturnal {}

public interface HasBigEyes extends Nocturnal {}
```

Unlike a class, which can extend only one class, an interface can extend multiple interfaces.

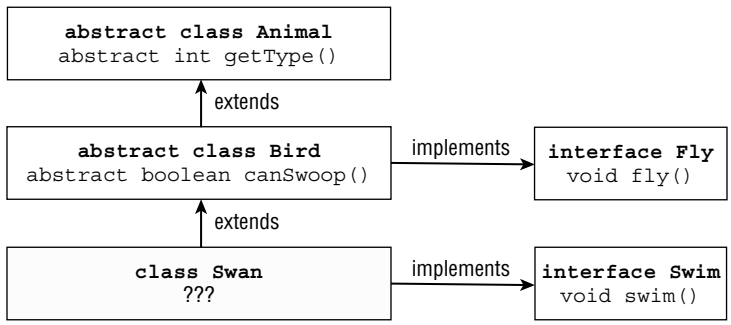
```
public interface Nocturnal {
   public int hunt();
}

public interface CanFly {
   public void flap();
}

public interface HasBigEyes extends Nocturnal, CanFly {}

public class Owl implements HasBigEyes {
   public int hunt() { return 5; }
   public void flap() { System.out.println("Flap!"); }
}
```

In this example, the `Owl` class implements the `HasBigEyes` interface and must implement the `hunt()` and `flap()` methods. Extending two interfaces is permitted because interfaces are not initialized as part of a class hierarchy. Unlike abstract classes, they do not contain constructors and are not part of instance initialization. Interfaces simply define a set of rules and methods that a class implementing them must follow.

Inheriting an Interface

Like an abstract class, when a concrete class inherits an interface, all of the inherited abstract methods must be implemented. We illustrate this principle in Figure 7.3. How many abstract methods does the concrete `Swan` class inherit?

FIGURE 7.3 Interface Inheritance



Give up? The concrete `Swan` class inherits four abstract methods that it must implement: `getType()`, `canSwoop()`, `fly()`, and `swim()`. Let's take a look at another example involving an abstract class that implements an interface:

```
public interface HasTail {
   public int getTailLength();
}

public interface HasWhiskers {
   public int getNumberOfWhiskers();
}

public abstract class HarborSeal implements HasTail, HasWhiskers {}

public class CommonSeal extends HarborSeal {}  // DOES NOT COMPILE
```

The `HarborSeal` class compiles because it is `abstract` and not required to implement any of the `abstract` methods it inherits. The concrete `CommonSeal` class, though, must override all inherited `abstract` methods.

Mixing Class and Interface Keywords

The exam creators are fond of questions that mix class and interface terminology. Although a class can implement an interface, a class cannot extend an interface. Likewise, while an interface can extend another interface, an interface cannot implement another interface. The following examples illustrate these principles:

```
public interface CanRun {}
public class Cheetah extends CanRun {}   // DOES NOT COMPILE

public class Hyena {}
public interface HasFur extends Hyena {} // DOES NOT COMPILE
```

The first example shows a class trying to extend an interface and doesn't compile. The second example shows an interface trying to extend a class, which also doesn't compile. Be wary of examples on the exam that mix class and interface declarations.

Inheriting Duplicate Abstract Methods

Java supports inheriting two abstract methods that have compatible method declarations.

```
public interface Herbivore { public void eatPlants(); }

public interface Omnivore  { public void eatPlants(); }

public class Bear implements Herbivore, Omnivore {
```

```
   public void eatPlants() {
      System.out.println("Eating plants");
   } }
```

By *compatible*, we mean a method can be written that properly overrides both inherited methods: for example, by using covariant return types that you learned about in Chapter 6.

The following is an example of an incompatible declaration:

```
public interface Herbivore { public void eatPlants(); }

public interface Omnivore  { public int eatPlants(); }

public class Tiger implements Herbivore, Omnivore { // DOES NOT COMPILE
   ...
}
```

It's impossible to write a version of `Tiger` that satisfies both inherited `abstract` methods. The code does not compile, regardless of what is declared inside the `Tiger` class.

Inserting Implicit Modifiers

As mentioned earlier, an implicit modifier is one that the compiler will automatically insert. It's reminiscent of the compiler inserting a default no-argument constructor if you do not define a constructor, which you learned about in Chapter 6. You can choose to insert these implicit modifiers yourself or let the compiler insert them for you.

The following list includes the implicit modifiers for interfaces that you need to know for the exam:

- Interfaces are implicitly `abstract`.
- Interface variables are implicitly `public`, `static`, and `final`.
- Interface methods without a body are implicitly `abstract`.
- Interface methods without the `private` modifier are implicitly `public`.

The last rule applies to `abstract`, `default`, and `static` interface methods, which we cover in the next section.

Let's take a look at an example. The following two interface definitions are equivalent, as the compiler will convert them both to the second declaration:

```
public interface Soar {
   int MAX_HEIGHT = 10;
   final static boolean UNDERWATER = true;
   void fly(int speed);
   abstract void takeoff();
   public abstract double dive();
}
```

```
public abstract interface Soar {
   public static final int MAX_HEIGHT = 10;
   public final static boolean UNDERWATER = true;
   public abstract void fly(int speed);
   public abstract void takeoff();
   public abstract double dive();
}
```

In this example, we've marked in bold the implicit modifiers that the compiler automatically inserts. First, the `abstract` keyword is added to the interface declaration. Next, the `public`, `static`, and `final` keywords are added to the interface variables if they do not exist. Finally, each abstract method is prepended with the `abstract` and `public` keywords if it does not contain them already.

Conflicting Modifiers

What happens if a developer marks a method or variable with a modifier that conflicts with an implicit modifier? For example, if an abstract method is implicitly `public`, can it be explicitly marked `protected` or `private`?

```
public interface Dance {
   private int count = 4;  // DOES NOT COMPILE
   protected void step();  // DOES NOT COMPILE
}
```

Neither of these interface member declarations compiles, as the compiler will apply the `public` modifier to both, resulting in a conflict.

Differences between Interfaces and Abstract Classes

Even though abstract classes and interfaces are both considered abstract types, only interfaces make use of implicit modifiers. How do the `play()` methods differ in the following two definitions?

```
abstract class Husky {    // abstract required in class declaration
   abstract void play();  // abstract required in method declaration
}

interface Poodle {        // abstract optional in interface declaration
   void play();           // abstract optional in method declaration
}
```

Both of these method definitions are considered abstract. That said, the `Husky` class will not compile if the `play()` method is not marked `abstract`, whereas the method in the `Poodle` interface will compile with or without the `abstract` modifier.

What about the access level of the `play()` method? Can you spot anything wrong with the following class definitions that use our abstract types?

```
public class Webby extends Husky {
   void play() {}      // OK - play() is declared with package access in Husky
}

public class Georgette implements Poodle {
   void play() {}      // DOES NOT COMPILE - play() is public in Poodle
}
```

The `Webby` class compiles, but the `Georgette` class does not. Even though the two method implementations are identical, the method in the `Georgette` class reduces the access modifier on the method from `public` to package access.

Declaring Concrete Interface Methods

While interfaces started with abstract methods and constants, they've grown to include a lot more. Table 7.1 lists the six interface member types that you need to know for the exam. We've already covered abstract methods and constants, so we focus on the remaining four concrete methods in this section.

TABLE 7.1 Interface member types

	Membership type	Required modifiers	Implicit modifiers	Has value or body?
Constant variable	Class	—	`public static final`	Yes
`abstract` method	Instance	—	`public abstract`	No
`default` method	Instance	`default`	`public`	Yes
`static` method	Class	`static`	`public`	Yes
`private` method	Instance	`private`	—	Yes
`private static` method	Class	`private static`	—	Yes

In Table 7.1, the membership type determines how it is able to be accessed. A method with a membership type of *class* is shared among all instances of the interface, whereas a method with a membership type of *instance* is associated with a particular instance of the interface.

What About *protected* or Package Interface Members?

Alongside `public` methods, interfaces now support `private` methods. They do not support `protected` access, though, as a class cannot extend an interface. They also do not support package access, although more likely for syntax reasons and backward compatibility. Since interface methods without an access modifier have been considered implicitly `public`, changing this behavior to package access would break many existing programs!

Writing a *default* Interface Method

The first type of concrete method you should be familiar with for the exam is a default method. A *default method* is a method defined in an interface with the `default` keyword and includes a method body. It may be optionally overridden by a class implementing the interface.

One use of `default` methods is for backward compatibility. You can add a new `default` method to an interface without the need to modify all of the existing classes that implement the interface. The older classes will just use the *default* implementation of the method defined in the interface. This is where the name `default` method comes from!

The following is an example of a `default` method defined in an interface:

```
public interface IsColdBlooded {
   boolean hasScales();
   default double getTemperature() {
      return 10.0;
   } }
```

This example defines two interface methods, one `abstract` and one `default`. The following `Snake` class, which implements `IsColdBlooded`, must implement `hasScales()`. It may rely on the default implementation of `getTemperature()` or override the method with its own version:

```
public class Snake implements IsColdBlooded {
   public boolean hasScales() {       // Required override
      return true;
   }
```

```
   public double getTemperature() {   // Optional override
      return 12;
   }
}
```

Note that the `default` interface method modifier is not the same as the `default` label used in a `switch` statement or expression. Likewise, even though package access is sometimes referred to as default access, that feature is implemented by omitting an access modifier. Sorry if this is confusing! We agree Java has overused the word *default* over the years!

For the exam, you should be familiar with various rules for declaring `default` methods.

Default Interface Method Definition Rules

1. A `default` method may be declared only within an interface.
2. A `default` method must be marked with the `default` keyword and include a method body.
3. A `default` method is implicitly `public`.
4. A `default` method cannot be marked `abstract`, `final`, or `static`.
5. A `default` method may be overridden by a class that implements the interface.
6. If a class inherits two or more `default` methods with the same method signature, then the class must override the method.

The first rule should give you some comfort in that you'll only see `default` methods in interfaces. If you see them in a class or enum on the exam, something is wrong. The second rule just denotes syntax, as `default` methods must use the `default` keyword. For example, the following code snippets will not compile because they mix up concrete and abstract interface methods:

```
public interface Carnivore {
   public default void eatMeat();        // DOES NOT COMPILE
   public int getRequiredFoodAmount() {  // DOES NOT COMPILE
      return 13;
   } }
```

The next three rules for `default` methods follow from the relationship with abstract interface methods. Like abstract interface methods, `default` methods are implicitly `public`. Unlike abstract methods, though, `default` interface methods cannot be marked `abstract` since they provide a body. They also cannot be marked as `final`, because they are designed so that they can be overridden in classes implementing the interface, just like abstract methods. Finally, they cannot be marked `static` since they are associated with the instance of the class implementing the interface.

Inheriting Duplicate *default* Methods

The last rule for creating a `default` interface method requires some explanation. For example, what value would the following code output?

```
public interface Walk {
   public default int getSpeed() { return 5; }
}

public interface Run {
   public default int getSpeed() { return 10; }
}

public class Cat implements Walk, Run {}  // DOES NOT COMPILE
```

In this example, `Cat` inherits the two `default` methods for `getSpeed()`, so which does it use? Since `Walk` and `Run` are considered siblings in terms of how they are used in the `Cat` class, it is not clear whether the code should output `5` or `10`. In this case, the compiler throws up its hands and says, "Too hard, I give up!" and fails.

All is not lost, though. If the class implementing the interfaces *overrides* the duplicate `default` method, the code will compile without issue. By overriding the conflicting method, the ambiguity about which version of the method to call has been removed. For example, the following modified implementation of `Cat` will compile:

```
public class Cat implements Walk, Run {
   public int getSpeed() { return 1; }
}
```

Calling a Hidden *default* Method

In the last section, we showed how our `Cat` class could override a pair of conflicting `default` methods, but what if the `Cat` class wanted to access the version of `getSpeed()` in `Walk` or `Run`? Is it still accessible?

Yes, but it requires some special syntax.

```
public class Cat implements Walk, Run {
   public int getSpeed() {
      return 1;
   }

   public int getWalkSpeed() {
      return Walk.super.getSpeed();
   } }
```

This is an area where a `default` method exhibits properties of both a `static` and instance method. We use the interface name to indicate which method we want to call, but we use the `super` keyword to show that we are following instance inheritance, not class inheritance. Note that calling `Walk.getSpeed()` or `Walk.this.getSpeed()` would not have worked. A bit confusing, we know, *but you need to be familiar with this syntax for the exam*.

Declaring *static* Interface Methods

Interfaces are also declared with `static` methods. These methods are defined explicitly with the `static` keyword and, for the most part, behave just like `static` methods defined in classes.

Static Interface Method Definition Rules

1. A `static` method must be marked with the `static` keyword and include a method body.
2. A `static` method without an access modifier is implicitly `public`.
3. A `static` method cannot be marked `abstract` or `final`.
4. A `static` method is not inherited and cannot be accessed in a class implementing the interface without a reference to the interface name.

These rules should follow from what you know so far of classes, interfaces, and `static` methods. For example, you can't declare `static` methods without a body in classes, either. Like `default` and abstract interface methods, `static` interface methods are implicitly `public` if they are declared without an access modifier. As you see shortly, you can use the `private` access modifier with `static` methods.

Let's take a look at a `static` interface method:

```
public interface Hop {
   static int getJumpHeight() {
      return 8;
   } }
```

Since the method is defined without an access modifier, the compiler will automatically insert the `public` access modifier. The method `getJumpHeight()` works just like a `static` method as defined in a class. In other words, it can be accessed without an instance of a class.

```
public class Skip {
   public int skip() {
      return Hop.getJumpHeight();
   } }
```

The last rule about inheritance might be a little confusing, so let's look at an example. The following is an example of a class `Bunny` that implements `Hop` and does not compile:

```
public class Bunny implements Hop {
   public void printDetails() {
      System.out.println(getJumpHeight());  // DOES NOT COMPILE
   } }
```

Without an explicit reference to the name of the interface, the code will not compile, even though Bunny implements Hop. This can be easily fixed by using the interface name:

```
public class Bunny implements Hop {
   public void printDetails() {
      System.out.println(Hop.getJumpHeight());
   } }
```

Notice we don't have the same problem we did when we inherited two `default` interface methods with the same signature. Java "solved" the multiple inheritance problem of `static` interface methods by not allowing them to be inherited!

Reusing Code with *private* Interface Methods

The last two types of concrete methods that can be added to interfaces are `private` and `private static` interface methods. Because both types of methods are `private`, they can only be used in the interface declaration in which they are declared. For this reason, they were added primarily to reduce code duplication. For example, consider the following code sample:

```
public interface Schedule {
   default void wakeUp()        { checkTime(7);  }
   private void haveBreakfast() { checkTime(9);  }
   static void workOut()        { checkTime(18); }
   private static void checkTime(int hour) {
      if (hour> 17) {
         System.out.println("You're late!");
      } else {
         System.out.println("You have "+(17-hour)+" hours left "
               + "to make the appointment");
      } } }
```

You could write this interface without using a `private` method by copying the contents of the `checkTime()` method into the places it is used. It's a lot shorter and easier to read if you don't. Since the authors of Java were nice enough to add this feature for our convenience, we might as well use it!

We could have also declared `checkTime()` as `public` in the previous example, but this would expose the method to use outside the interface. One important tenet of encapsulation is to not expose the internal workings of a class or interface when not required. We cover encapsulation later in this chapter.

The difference between a non-`static private` method and a `static` one is analogous to the difference between an instance and `static` method declared within a class. In particular, it's all about what methods each can be called from.

Private Interface Method Definition Rules

1. A `private` interface method must be marked with the `private` modifier and include a method body.
2. A `private static` interface method may be called by any method within the interface definition.
3. A `private` interface method may only be called by `default` and other `private` non-`static` methods within the interface definition.

Another way to think of it is that a `private` interface method is only accessible to non-`static` methods defined within the interface. A `private static` interface method, on the other hand, can be accessed by any method in the interface. For both types of `private` methods, a class inheriting the interface cannot directly invoke them.

Calling Abstract Methods

We've talked a lot about the newer types of interface methods, but what about `abstract` methods? It turns out `default` and `private` non-`static` methods can access `abstract` methods declared in the interface. This is the primary reason we associate these methods with instance membership. When they are invoked, there is an instance of the interface.

```
public interface ZooRenovation {
   public String projectName();
   abstract String status();
   default void printStatus() {
      System.out.print("The " + projectName() + " project " + status());
   } }
```

In this example, both `projectName()` and `status()` have the same modifiers (`abstract` and `public` are implicit) and can be called by the `default` method `printStatus()`.

Reviewing Interface Members

We conclude our discussion of interface members with Table 7.2, which shows the access rules for members within and outside an interface.

TABLE 7.2 Interface member access

	Accessible from `default` and `private` methods within the interface?	**Accessible from `static` methods within the interface?**	**Accessible from methods in classes inheriting the interface?**	**Accessible without an instance of the interface?**
Constant variable	Yes	Yes	Yes	Yes
`abstract` method	Yes	No	Yes	No
`default` method	Yes	No	Yes	No
`static` method	Yes	Yes	Yes (interface name required)	Yes (interface name required)
`private` method	Yes	No	No	No
`private static` method	Yes	Yes	No	No

While Table 7.2 might seem like a lot to remember, here are some quick tips for the exam:

- Treat `abstract`, `default`, and non-`static private` methods as belonging to an instance of the interface.
- Treat `static` methods and variables as belonging to the interface class object.
- All `private` interface method types are only accessible within the interface declaration.

Using these rules, which of the following methods do not compile?

```
public interface ZooTrainTour {
   abstract int getTrainName();
   private static void ride() {}
   default void playHorn() { getTrainName(); ride(); }
   public static void slowDown() { playHorn(); }
   static void speedUp() { ride(); }
}
```

The `ride()` method is `private` and `static`, so it can be accessed by any `default` or `static` method within the interface declaration. The `getTrainName()` is `abstract`, so it can be accessed by a `default` method associated with the instance. The `slowDown()`

method is `static`, though, and cannot call a `default` or `private` method, such as `playHorn()`, without an explicit reference object. Therefore, the `slowDown()` method does not compile.

Give yourself a pat on the back! You just learned a lot about interfaces, probably more than you thought possible. Now take a deep breath. Ready? The next type we are going to cover is enums.

Working with Enums

In programming, it is common to have a type that can only have a finite set of values, such as days of the week, seasons of the year, primary colors, and so on. An *enumeration*, or *enum* for short, is like a fixed set of constants.

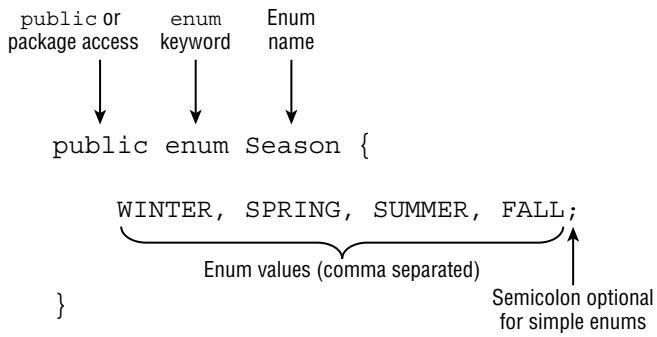
Using an enum is much better than using a bunch of constants because it provides type-safe checking. With numeric or `String` constants, you can pass an invalid value and not find out until runtime. With enums, it is impossible to create an invalid enum value without introducing a compiler error.

Enumerations show up whenever you have a set of items whose types are known at compile time. Common examples include the compass directions, the months of the year, the planets in the solar system, and the cards in a deck (well, maybe not the planets in a solar system, given that Pluto had its planetary status revoked).

Creating Simple Enums

To create an enum, declare a type with the `enum` keyword, a name, and a list of values, as shown in Figure 7.4.

FIGURE 7.4 Defining a simple enum



We refer to an enum that only contains a list of values as a *simple* enum. When working with simple enums, the semicolon at the end of the list is optional. Keep the `Season` enum handy, as we use it throughout this section.

Enum values are considered constants and are commonly written using snake case. For example, an enum declaring a list of ice cream flavors might include values like `VANILLA`, `ROCKY_ROAD`, `MINT_CHOCOLATE_CHIP`, and so on.

Using an enum is super easy.

```
var s = Season.SUMMER;
System.out.println(Season.SUMMER);      // SUMMER
System.out.println(s == Season.SUMMER); // true
```

As you can see, enums print the name of the enum when `toString()` is called. They can be compared using `==` because they are like `static final` constants. In other words, you can use `equals()` or `==` to compare enums, since each enum value is initialized only once in the Java Virtual Machine (JVM).

One thing that you can't do is extend an enum.

```
public enum ExtendedSeason extends Season {} // DOES NOT COMPILE
```

The values in an enum are fixed. You cannot add more by extending the enum.

Calling the *values()*, *name()*, and *ordinal()* Methods

An enum provides a `values()` method to get an array of all of the values. You can use this like any normal array, including in a for-each loop:

```
for(var season: Season.values()) {
   System.out.println(season.name() + " " + season.ordinal());
}
```

The output shows that each enum value has a corresponding `int` value, and the values are listed in the order in which they are declared:

```
WINTER 0
SPRING 1
SUMMER 2
FALL 3
```

The `int` value will remain the same during your program, but the program is easier to read if you stick to the human-readable enum value.

You can't compare an `int` and an enum value directly anyway since an enum is a type, like a Java class, and *not* a primitive `int`.

```
if ( Season.SUMMER == 2) {} // DOES NOT COMPILE
```

Calling the *valueOf*() Method

Another useful feature is retrieving an enum value from a `String` using the `valueOf()` method. This is helpful when working with older code or parsing user input. The `String` passed in must match the enum value exactly, though.

```
Season s = Season.valueOf("SUMMER"); // SUMMER

Season t = Season.valueOf("summer"); // IllegalArgumentException
```

The first statement works and assigns the proper enum value to `s`. Note that this line is not creating an enum value, at least not directly. Each enum value is created once when the enum is first loaded. Once the enum has been loaded, it retrieves the single enum value with the matching name.

The second statement encounters a problem. There is no enum value with the lowercase name `summer`. Java throws up its hands in defeat and throws an `IllegalArgumentException`.

```
Exception in thread "main" java.lang.IllegalArgumentException:
   No enum constant enums.Season.summer
```

Using Enums in *switch* Statements

Enums can be used in `switch` statements and expressions. Pay attention to the `case` values in this code:

```
Season summer = Season.SUMMER;
switch(summer) {
   case WINTER:
      System.out.print("Get out the sled!");
      break;
   case SUMMER:
      System.out.print("Time for the pool!");
      break;
   default:
      System.out.print("Is it summer yet?");
}
```

The code prints `"Time for the pool!"` since it matches `SUMMER`. In each `case` statement, we just typed the value of the enum rather than writing `Season.WINTER`. After all, the compiler already knows that the only possible matches can be enum values. Java treats the enum type as implicit. In fact, if you were to type `case Season.WINTER`, it would not compile. Don't believe us? Take a look at this equivalent example using a `switch` expression:

```
Season summer = Season.SUMMER;
var message = switch(summer) {
   case Season.WINTER -> "Get out the sled!";  // DOES NOT COMPILE
```

```
   case 0 -> "Time for the pool!";            // DOES NOT COMPILE
   default -> "Is it summer yet?";
};
System.out.print(message);
```

The first `case` statement does not compile because `Season` is used in the `case` value. If we changed `Season.FALL` to just `FALL`, then the line would compile. What about the second `case` statement? Just as earlier we said that you can't compare enums with `int` values, you cannot use them in a `switch` statement with `int` values. On the exam, pay special attention when working with enums that they are used only as enums.

Adding Constructors, Fields, and Methods

While a simple enum is composed of just a list of values, we can define a *complex* enum with additional elements. Let's say our zoo wants to keep track of traffic patterns to determine which seasons get the most visitors.

```
1: public enum Season {
2:    WINTER("Low"), SPRING("Medium"), SUMMER("High"), FALL("Medium");
3:    private final String expectedVisitors;
4:    private Season(String expectedVisitors) {
5:       this.expectedVisitors = expectedVisitors;
6:    }
7:    public void printExpectedVisitors() {
8:       System.out.println(expectedVisitors);
9:    } }
```

There are a few things to notice here. On line 2, the list of enum values ends with a semicolon (`;`). While this is optional when our enum is composed solely of a list of values, it is required if there is anything in the enum besides the values.

Lines 3–9 are regular Java code. We have an instance variable, a constructor, and a method. We mark the instance variable `private` and `final` on line 3 so that our enum properties cannot be modified.

Although it is possible to create an enum with instance variables that can be modified, it is a very poor practice to do so since they are shared within the JVM. When designing an enum, the values should be immutable.

All enum constructors are implicitly `private`, with the modifier being optional. This is reasonable since you can't extend an enum and the constructors can be called only within the enum itself. In fact, an enum constructor will not compile if it contains a `public` or `protected` modifier.

What about the parentheses on line 2? Those are constructor calls, but without the new keyword normally used for objects. The first time we ask for any of the enum values, Java constructs all of the enum values. After that, Java just returns the already constructed enum values. Given that explanation, you can see why this calls the constructor only once:

```
public enum OnlyOne {
   ONCE(true);
   private OnlyOne(boolean b) {
      System.out.print("constructing,");
   }
}

public class PrintTheOne {
   public static void main(String[] args) {
      System.out.print("begin,");
      OnlyOne firstCall = OnlyOne.ONCE;   // Prints constructing,
      OnlyOne secondCall = OnlyOne.ONCE;  // Doesn't print anything
      System.out.print("end");
   }
}
```

This class prints the following:

```
begin,constructing,end
```

If the `OnlyOne` enum was used earlier in the program, and therefore initialized sooner, then the line that declares the `firstCall` variable would not print anything.

How do we call an enum method? That's easy, too: we just use the enum value followed by the method call.

```
Season.SUMMER.printExpectedVisitors();
```

Sometimes you want to define different methods for each enum. For example, our zoo has different seasonal hours. It is cold and gets dark early in the winter. We can keep track of the hours through instance variables, or we can let each enum value manage hours itself.

```
public enum Season {
   WINTER {
      public String getHours() { return "10am-3pm"; }
   },
   SPRING {
      public String getHours() { return "9am-5pm"; }
   },
   SUMMER {
      public String getHours() { return "9am-7pm"; }
   },
```

```
   FALL {
      public String getHours() { return "9am-5pm"; }
   };
   public abstract String getHours();
}
```

What's going on here? It looks like we created an `abstract` class and a bunch of tiny subclasses. In a way, we did. The enum itself has an `abstract` method. This means that each and every enum value is required to implement this method. If we forget to implement the method for one of the values, we get a compiler error:

```
The enum constant WINTER must implement the abstract method getHours()
```

But what if we don't want each and every enum value to have a method? No problem. We can create an implementation for all values and override it only for the special cases.

```
public enum Season {
   WINTER {
      public String getHours() { return "10am-3pm"; }
   },
   SUMMER {
      public String getHours() { return "9am-7pm"; }
   },
   SPRING, FALL;
   public String getHours() { return "9am-5pm"; }
}
```

This looks better. We only code the special cases and let the others use the enum-provided implementation.

An enum can even implement an interface, as this just requires overriding the abstract methods:

```
public interface Weather { int getAverageTemperature(); }

public enum Season implements Weather {
   WINTER, SPRING, SUMMER, FALL;
   public int getAverageTemperature() { return 30; }
}
```

Just because an enum can have lots of methods doesn't mean that it should. Try to keep your enums simple. If your enum is more than a page or two, it is probably too long. When enums get too long or too complex, they are hard to read.

You might have noticed that in each of these enum examples, the list of values came first. This was not an accident. Whether the enum is simple or complex, the list of values always comes first.

Sealing Classes

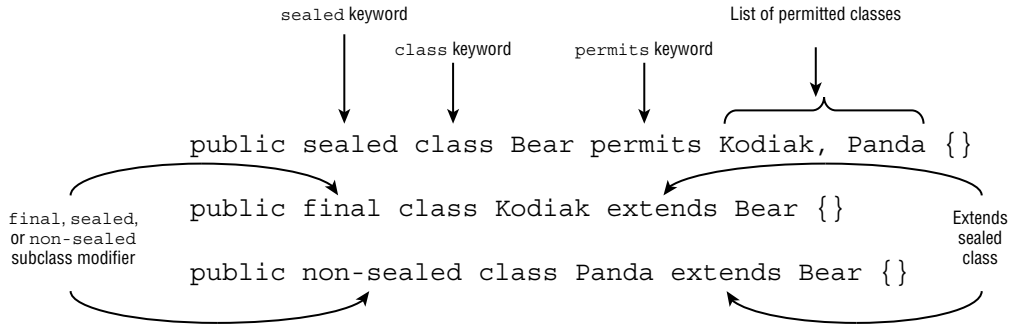
An enum with many constructors, fields, and methods may start to resemble a full-featured class. What if we could create a class but limit the direct subclasses to a fixed set of classes? Enter sealed classes! A *sealed class* is a class that restricts which other classes may directly extend it. These are brand new to Java 17, so expect to see at least one question about them on the exam.

Did you happen to notice that we said *directly extend* in the definition of a sealed class? As you see shortly, there is a way for a class not named in the sealed class declaration to extend it indirectly. Unless we say otherwise, though, assume that we're referring to subclasses that directly extend the sealed class.

Declaring a Sealed Class

Let's start with a simple example. A sealed class declares a list of classes that can extend it, while the subclasses declare that they extend the sealed class. Figure 7.5 declares a sealed class with two direct subclasses.

FIGURE 7.5 Defining a sealed class



Notice anything new? Java 17 includes three new keywords that you should be familiar with for the exam. We often use `final` with sealed subclasses, but we get into each of these after we cover the basics.

Sealed Class Keywords

- **`sealed`**: Indicates that a class or interface may only be extended/implemented by named classes or interfaces

- **permits**: Used with the `sealed` keyword to list the classes and interfaces allowed
- **non-sealed**: Applied to a class or interface that extends a sealed class, indicating that it can be extended by unspecified classes

Pretty easy so far, right? The exam is just as likely to test you on what sealed classes cannot be used for. For example, can you see why each of these sets of declarations does not compile?

```
public class sealed Frog permits GlassFrog {}  // DOES NOT COMPILE
public final class GlassFrog extends Frog {}

public abstract sealed class Wolf permits Timber {}
public final class Timber extends Wolf {}
public final class MyWolf extends Wolf {}  // DOES NOT COMPILE
```

The first example does not compile because the `class` and `sealed` modifiers are in the wrong order. The modifier has to be before the `class` type. The second example does not compile because `MyWolf` isn't listed in the declaration of `Wolf`.

Sealed classes are commonly declared with the `abstract` modifier, although this is certainly not required.

Declaring a sealed class with the `sealed` modifier is the easy part. Most of the time, if you see a question on the exam about sealed classes, they are testing your knowledge of whether the subclass extends the sealed class properly. There are a number of important rules you need to know for the exam, so read the next sections carefully.

Compiling Sealed Classes

Let's say we create a `Penguin` class and compile it in a new package without any other source code. With that in mind, does the following compile?

```
// Penguin.java
package zoo;
public sealed class Penguin permits Emperor {}
```

No, it does not! Why? The answer is that a sealed class needs to be declared (and compiled) in the same package as its direct subclasses. But what about the subclasses themselves? They must each extend the sealed class. For example, the following does not compile.

```
// Penguin.java
package zoo;
public sealed class Penguin permits Emperor {}  // DOES NOT COMPILE
```

```
// Emperor.java
package zoo;
public final class Emperor {}
```

Even though the `Emperor` class is declared, it does not extend the `Penguin` class.

But wait, there's more! In Chapter 12, "Modules," you learn about *named modules*, which allow sealed classes and their direct subclasses in different packages, provided they are in the same named module.

Specifying the Subclass Modifier

While some types, like interfaces, have a certain number of implicit modifiers, sealed classes do not. *Every class that directly extends a sealed class must specify exactly one of the following three modifiers*: `final`, `sealed`, or `non-sealed`. Remember this rule for the exam!

A *final* Subclass

The first modifier we're going to look at that can be applied to a direct subclass of a sealed class is the `final` modifier.

```
public sealed class Antelope permits Gazelle {}

public final class Gazelle extends Antelope {}

public class George extends Gazelle {} // DOES NOT COMPILE
```

Just as with a regular class, the `final` modifier prevents the subclass `Gazelle` from being extended further.

A *sealed* Subclass

Next, let's look at an example using the `sealed` modifier:

```
public sealed class Mammal permits Equine {}

public sealed class Equine extends Mammal permits Zebra {}

public final class Zebra extends Equine {}
```

The `sealed` modifier applied to the subclass `Equine` means the same kind of rules that we applied to the parent class `Mammal` must be present. Namely, `Equine` defines its own list of permitted subclasses. Notice in this example that `Zebra` is an indirect subclass of `Mammal` but is not named in the `Mammal` class.

Despite allowing indirect subclasses not named in `Mammal`, the list of classes that can inherit `Mammal` is still fixed. If you have a reference to a `Mammal` object, it must be a `Mammal`, `Equine`, or `Zebra`.

A *non-sealed* Subclass

The `non-sealed` modifier is used to open a sealed parent class to potentially unknown subclasses. Let's modify our earlier example to allow `MyWolf` to compile without modifying the declaration of `Wolf`:

```
public sealed class Wolf permits Timber {}

public non-sealed class Timber extends Wolf {}

public class MyWolf extends Timber {}
```

In this example, we are able to create an indirect subclass of `Wolf`, called `MyWolf`, not named in the declaration of `Wolf`. Also notice that `MyWolf` is not `final`, so it may be extended by any subclass, such as `MyFurryWolf`.

```
public class MyFurryWolf extends MyWolf {}
```

At first glance, this might seem a bit counterintuitive. After all, we were able to create subclasses of `Wolf` that were not declared in `Wolf`. So is `Wolf` still sealed? Yes, but that's thanks to polymorphism. Any instance of `MyWolf` or `MyFurryWolf` is also an instance of `Timber`, which is named in the `Wolf` declaration. We discuss polymorphism more toward the end of this chapter.

If you're still worried about opening a sealed class too much with a `non-sealed` subclass, remember that the person writing the sealed class can see the declaration of all direct subclasses at compile time. They can decide whether to allow the `non-sealed` subclass to be supported.

Omitting the *permits* Clause

Up until now, all of the examples you've seen have required a `permits` clause when declaring a sealed class, but this is not always the case. Imagine that you have a `Snake.java` file with two top-level classes defined inside it:

```
// Snake.java
public sealed class Snake permits Cobra {}
final class Cobra extends Snake {}
```

In this case, the `permits` clause is optional and can be omitted. The `extends` keyword is still required in the subclass, though:

```
// Snake.java
public sealed class Snake {}
final class Cobra extends Snake {}
```

If these classes were in separate files, this code would not compile! This rule also applies to sealed classes with nested subclasses.

```
// Snake.java
public sealed class Snake {
   final class Cobra extends Snake {}
}
```

Referencing Nested Subclasses

While it makes the code easier to read if you omit the `permits` clause for nested subclasses, you are welcome to name them. However, the syntax might be different than you expect.

```
public sealed class Snake permits Cobra {  // DOES NOT COMPILE
   final class Cobra extends Snake {}
}
```

This code does not compile because `Cobra` requires a reference to the `Snake` namespace. The following fixes this issue:

```
public sealed class Snake permits Snake.Cobra {
   final class Cobra extends Snake {}
}
```

When all of your subclasses are nested, we strongly recommend omitting the `permits` class.

We cover nested classes shortly. For now, you just need to know that a nested class is a class defined inside another class and that the omit rule also applies to nested classes.

Table 7.3 is a handy reference to these cases.

TABLE 7.3 Usage of the `permits` clause in sealed classes

Location of direct subclasses	`permits` clause
In a different file from the sealed class	Required
In the same file as the sealed class	Permitted, but not required
Nested inside of the sealed class	Permitted, but not required

Sealing Interfaces

Besides classes, interfaces can also be sealed. The idea is analogous to classes, and many of the same rules apply. For example, the sealed interface must appear in the same package or named module as the classes or interfaces that directly extend or implement it.

One distinct feature of a sealed interface is that the `permits` list can apply to a class that implements the interface or an interface that extends the interface.

```
// Sealed interface
public sealed interface Swims permits Duck, Swan, Floats {}

// Classes permitted to implement sealed interface
public final class Duck implements Swims {}
public final class Swan implements Swims {}

// Interface permitted to extend sealed interface
public non-sealed interface Floats extends Swims {}
```

What about the modifier applied to interfaces that extend the sealed interface? Well, remember that interfaces are implicitly `abstract` and cannot be marked `final`. For this reason, interfaces that extend a sealed interface can only be marked `sealed` or `non-sealed`. They cannot be marked `final`.

Reviewing Sealed Class Rules

Any time you see a sealed class on the exam, pay close attention to the subclass declaration and modifiers.

Sealed Class Rules

- Sealed classes are declared with the `sealed` and `permits` modifiers.
- Sealed classes must be declared in the same package or named module as their direct subclasses.

- Direct subclasses of sealed classes must be marked `final`, `sealed`, or `non-sealed`.
- The `permits` clause is optional if the sealed class and its direct subclasses are declared within the same file or the subclasses are nested within the sealed class.
- Interfaces can be sealed to limit the classes that implement them or the interfaces that extend them.

Real World Scenario

Why Have Sealed Classes?

In Chapter 3, "Making Decisions," you learned about `switch` expressions and pattern matching. Imagine if we could treat a sealed class like an enum in a `switch` expression by applying pattern matching. Given a sealed class `Fish` with two direct subclasses, it might look something like this:

```
public void printName(Fish fish) {
   System.out.println(switch(fish) {
      case Trout t -> t.getTroutName();
      case Bass b -> b.getBassName();
   });
}
```

If `Fish` wasn't sealed, the `switch` expression would require a `default` branch, or the code would not compile. Since it's sealed, the compiler knows all the options! The good news is that this feature is on the way, but the bad news is that it's still in Preview in Java 17 and not officially released. We just wanted to give you an idea of where some of these new features were heading.

Encapsulating Data with Records

We saved the best new Java type for last! If you've heard anything about the new features in Java, you have probably heard about records. Records are exciting because they remove a ton of boilerplate code. Before we get into records, it helps to have some context of why they were added to the language, so we start with encapsulation.

Understanding Encapsulation

A *POJO*, which stands for Plain Old Java Object, is a class used to model and pass data around, often with few or no complex methods (hence the "plain" part of the definition). You might have also heard of a JavaBean, which is POJO that has some additional rules applied.

Let's create a simple POJO with two fields:

```
public class Crane {
   int numberEggs;
   String name;
   public Crane(int numberEggs, String name) {
      this.numberEggs = numberEggs;
      this.name = name;
   }
}
```

Uh oh, the fields are package access. Why do we care? That means someone outside the class in the same package could change these values and create invalid data such as this:

```
public class Poacher {
   public void badActor() {
      var mother = new Crane(5, "Cathy");
      mother.numberEggs = -100;
   }
}
```

This is clearly no good. We do not want the mother `Crane` to have a negative number of eggs! Encapsulation to the rescue. *Encapsulation* is a way to protect class members by restricting access to them. In Java, it is commonly implemented by declaring all instance variables `private`. Callers are required to use methods to retrieve or modify instance variables.

Encapsulation is about protecting a class from unexpected use. It also allows us to modify the methods and behavior of the class later without someone already having direct access to an instance variable within the class. For example, we can change the data type of an instance variable but maintain the same method signatures. In this manner, we maintain full control over the internal workings of a class.

Let's take a look at the newly encapsulated (and immutable) `Crane` class:

```
1:  public final class Crane {
2:     private final int numberEggs;
3:     private final String name;
4:     public Crane(int numberEggs, String name) {
5:        if (numberEggs >= 0) this.numberEggs = numberEggs; // guard condition
```

```
6:        else throw new IllegalArgumentException();
7:        this.name = name;
8:     }
9:     public int getNumberEggs() {         // getter
10:       return numberEggs;
11:    }
12:    public String getName() {            // getter
13:       return name;
14:    }
15: }
```

Note that the instance variables are now `private` on lines 2 and 3. This means only code within the class can read or write their values. Since we wrote the class, we know better than to set a negative number of eggs. We added a method on lines 9–11 to read the value, which is called an *accessor method* or a getter.

You might have noticed that we marked the class and its instance variables `final`, and we don't have any *mutator methods*, or setters, to modify the value of the instance variables. That's because we want our class to be immutable in addition to being well encapsulated. As you saw in Chapter 6, the immutable objects pattern is an object-oriented design pattern in which an object cannot be modified after it is created. Instead of modifying an immutable object, you create a new object that contains any properties from the original object you want copied over.

To review, remember that data (an instance variable) is `private` and getters/setters are `public` for encapsulation. You don't even have to provide getters and setters. As long as the instance variables are `private`, you are good. For example, the following class is well encapsulated, although it is not terribly useful since it doesn't declare any non-`private` methods:

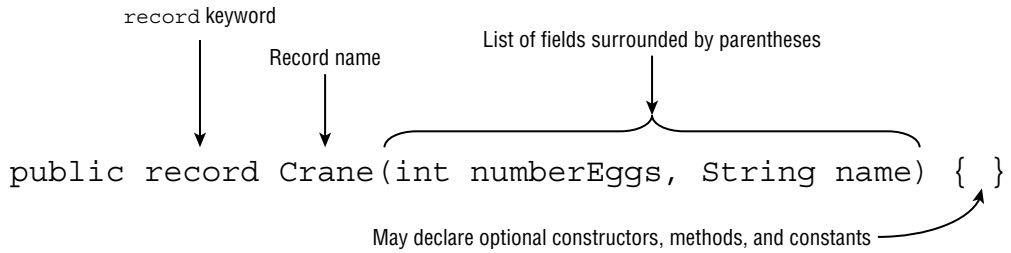
```
public class Vet {
   private String name = "Dr Rogers";
   private int yearsExperience = 25;
}
```

You must omit the setters for a class to be immutable. Review Chapter 6 for the additional rules on creating immutable objects.

Applying Records

Our `Crane` class was 15 lines long. We can write that much more succinctly, as shown in Figure 7.6. Putting aside the guard clause on `numberEggs` in the constructor for a moment, this record is equivalent and immutable!

FIGURE 7.6 Defining a record



Wow! It's only one line long! A *record* is a special type of data-oriented class in which the compiler inserts boilerplate code for you.

In fact, the compiler inserts *much more* than the 14 lines we wrote earlier. As a bonus, the compiler inserts *useful* implementations of the `Object` methods `equals()`, `hashCode()`, and `toString()`. We've covered a lot in one line of code!

Now imagine that we had 10 data fields instead of 2. That's a lot of methods we are saved from writing. And we haven't even talked about constructors! Worse yet, any time someone changes a field, dozens of lines of related code may need to be updated. For example, `name` may be used in the constructor, `toString()`, `equals()` method, and so on. If we have an application with hundreds of POJOs, a record can save us valuable time.

Creating an instance of a `Crane` and printing some fields is easy:

```
var mommy = new Crane(4, "Cammy");
System.out.println(mommy.numberEggs());  // 4
System.out.println(mommy.name());        // Cammy
```

A few things should stand out here. First, we never defined any constructors or methods in our `Crane` declaration. How does the compiler know what to do? Behind the scenes, it creates a constructor for you with the parameters in the same order in which they appear in the record declaration. Omitting or changing the type order will lead to compiler errors:

```
var mommy1 = new Crane("Cammy", 4);  // DOES NOT COMPILE
var mommy2 = new Crane("Cammy");     // DOES NOT COMPILE
```

For each field, it also creates an accessor as the field name, plus a set of parentheses. Unlike traditional POJOs or JavaBeans, the methods don't have the prefix `get` or `is`. Just a few more characters that records save you from having to type! Finally, records override a number of methods in `Object` for you.

Members Automatically Added to Records

- **Constructor**: A constructor with the parameters in the same order as the record declaration
- **Accessor method**: One accessor for each field

- **`equals()`**: A method to compare two elements that returns `true` if each field is equal in terms of `equals()`
- **`hashCode()`**: A consistent `hashCode()` method using all of the fields
- **`toString()`**: A `toString()` implementation that prints each field of the record in a convenient, easy-to-read format

The following shows examples of the new methods. Remember that the `println()` method will call the `toString()` method automatically on any object passed to it.

```
var father = new Crane(0, "Craig");
System.out.println(father);                // Crane[numberEggs=0, name=Craig]

var copy = new Crane(0, "Craig");
System.out.println(copy);                  // Crane[numberEggs=0, name=Craig]
System.out.println(father.equals(copy));   // true
System.out.println(father.hashCode() + ", " + copy.hashCode()); // 1007, 1007
```

That's the basics of records. We say "basics" because there's a lot more you can do with them, as you see in the next sections.

Given our one-line declaration of `Crane`, imagine how much code and work would be required to write an equivalent class. It could easily take 40+ lines! It might be a fun exercise to try to write all the methods that records supply.

Fun fact: it is legal to have a record without any fields. It is simply declared with the `record` keyword and parentheses:

```
public record Crane() {}
```

Not the kind of thing you'd use in your own code, but it could come up on the exam.

Understanding Record Immutability

As you saw, records don't have setters. Every field is inherently `final` and cannot be modified after it has been written in the constructor. In order to "modify" a record, you have to make a new object and copy all of the data you want to preserve.

```
var cousin = new Crane(3, "Jenny");
var friend = new Crane(cousin.numberEggs(), "Janeice");
```

Just as interfaces are implicitly `abstract`, records are also implicitly `final`. The `final` modifier is optional but assumed.

```
public final record Crane(int numberEggs, String name) {}
```

Like enums, that means you can't extend or inherit a record.

```
public record BlueCrane() extends Crane {} // DOES NOT COMPILE
```

Also like enums, a record can implement a regular or sealed interface, provided it implements all of the abstract methods.

```
public interface Bird {}
public record Crane(int numberEggs, String name) implements Bird {}
```

Although well beyond the scope of this book, there are some good reasons to make data-oriented classes immutable. Doing so can lead to less error-prone code, as a new object is established any time the data is modified. It also makes them inherently thread-safe and usable in concurrent frameworks.

Declaring Constructors

What if you need to declare a record with some guards as we did earlier? In this section, we cover two ways we can accomplish this with records.

The Long Constructor

First, we can just declare the constructor the compiler normally inserts automatically, which we refer to as the *long constructor*.

```
public record Crane(int numberEggs, String name) {
   public Crane(int numberEggs, String name) {
      if (numberEggs < 0) throw new IllegalArgumentException();
      this.numberEggs = numberEggs;
      this.name = name;
   }
}
```

The compiler will not insert a constructor if you define one with the same list of parameters in the same order. Since each field is `final`, the constructor must set every field. For example, this record does not compile:

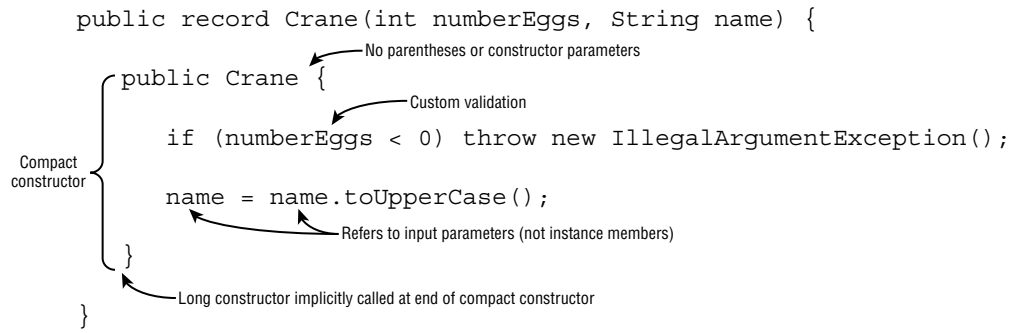
```
public record Crane(int numberEggs, String name) {
   public Crane(int numberEggs, String name) {}  // DOES NOT COMPILE
}
```

While being able to declare a constructor is a nice feature of records, it's also problematic. If we have 20 fields, we'll need to declare assignments for every one, introducing the boilerplate we sought to remove. Oh, bother!

Compact Constructors

Luckily, the authors of Java added the ability to define a compact constructor for records. A *compact constructor* is a special type of constructor used for records to process validation and transformations succinctly. It takes no parameters and implicitly sets all fields. Figure 7.7 shows an example of a compact constructor.

FIGURE 7.7 Declaring a compact constructor



Great! Now we can check the values we want, and we don't have to list all the constructor parameters and trivial assignments. Java will execute the full constructor after the compact constructor. You should also remember that a compact constructor is declared without parentheses, as the exam might try to trick you on this. As shown in Figure 7.7, we can even transform constructor parameters as we discuss more in the next section.

You might think that you need custom methods for every field in the record, like the negative check we did with `setNumberEggs()`. In practice, many POJOs are created for general-purpose use with little validation.

Transforming Parameters

Compact constructors give you the opportunity to apply transformations to any of the input values. See if you can figure out what the following compact constructor does:

```
public record Crane(int numberEggs, String name) {
   public Crane {
      if (name == null || name.length() < 1)
         throw new IllegalArgumentException();
      name = name.substring(0,1).toUpperCase()
         + name.substring(1).toLowerCase();
   }
}
```

Give up? It validates the string, then formats it such that only the first letter is capitalized. As before, Java calls the full constructor after the compact constructor but with the modified constructor parameters.

While compact constructors can modify the constructor parameters, *they cannot modify the fields of the record*. For example, this does not compile:

```
public record Crane(int numberEggs, String name) {
   public Crane {
      this.numberEggs = 10;  // DOES NOT COMPILE
   }
}
```

Removing the `this` reference allows the code to compile, as the constructor parameter is modified instead.

Although we covered both the long and compact forms of record constructors in this section, it is highly recommended that you stick with the compact form unless you have a good reason not to.

Overloaded Constructors

You can also create overloaded constructors that take a completely different list of parameters. They are more closely related to the long-form constructor and don't use any of the syntactical features of compact constructors.

```
public record Crane(int numberEggs, String name) {
   public Crane(String firstName, String lastName) {
      this(0, firstName + " " + lastName);
   }
}
```

The first line of an overloaded constructor must be an explicit call to another constructor via `this()`. If there are no other constructors, the long constructor must be called. Contrast this with what you learned about in Chapter 6, where calling `super()` or `this()` was often optional in constructor declarations. Also, unlike compact constructors, you can only transform the data on the first line. After the first line, all of the fields will already be assigned, and the object is immutable.

```
public record Crane(int numberEggs, String name) {
   public Crane(int numberEggs, String firstName, String lastName) {
      this(numberEggs + 1, firstName + " " + lastName);
      numberEggs = 10; // NO EFFECT (applies to parameter, not instance field)
      this.numberEggs = 20; // DOES NOT COMPILE
   }
}
```

As you saw in Chapter 6, you also can't declare two record constructors that call each other infinitely or as a cycle.

```
public record Crane(int numberEggs, String name) {
   public Crane(String name) {
      this(1);  // DOES NOT COMPILE
   }
   public Crane(int numberEggs) {
      this("");  // DOES NOT COMPILE
   }
}
```

Customizing Records

Since records are data-oriented, we've focused on the features of records you are likely to use. Records actually support many of the same features as a class. Here are some of the members that records can include and that you should be familiar with for the exam:

- Overloaded and compact constructors
- Instance methods including overriding any provided methods (accessors, `equals()`, `hashCode()`, `toString()`)
- Nested classes, interfaces, annotations, enum, and records

As an illustrative example, the following overrides two instance methods using the optional `@Override` annotation:

```
public record Crane(int numberEggs, String name) {
   @Override public int numberEggs() { return 10; }
   @Override public String toString() { return name; }
}
```

While you can add methods, `static` fields, and other data types, you cannot add instance fields outside the record declaration, even if they are `private`. Doing so defeats the purpose of using a record and could break immutability!

```
public record Crane(int numberEggs, String name) {
   private static int type = 10;
   public int size;              // DOES NOT COMPILE
   private boolean friendly;     // DOES NOT COMPILE
}
```

Records also do not support instance initializers. All initialization for the fields of a record must happen in a constructor.

While it's a useful feature that records support many of the same members as a class, try to keep them simple. Like the POJOs and JavaBeans they were born out of, the more complicated they get, the less usable they become.

This is the second time we've mentioned nested types, the first being with sealed classes and now records. Don't worry; we're covering them next!

Creating Nested Classes

A *nested class* is a class that is defined within another class. A nested class can come in one of four flavors.

- *Inner class*: A non-`static` type defined at the member level of a class
- *Static nested class:* A `static` type defined at the member level of a class
- *Local class*: A class defined within a method body
- *Anonymous class*: A special case of a local class that does not have a name

There are many benefits of using nested classes. They can define helper classes and restrict them to the containing class, thereby improving encapsulation. They can make it easy to create a class that will be used in only one place. They can even make the code cleaner and easier to read.

When used improperly, though, nested classes can sometimes make the code harder to read. They also tend to tightly couple the enclosing and inner class, but there may be cases where you want to use the inner class by itself. In this case, you should move the inner class out into a separate top-level class.

Unfortunately, the exam tests edge cases where programmers wouldn't typically use a nested class. This tends to create code that is difficult to read, so please never do this in practice!

By convention, and throughout this chapter, we often use the term nested *class* to refer to all nested *types*, including nested interfaces, enums, records, and annotations. You might even come across literature that refers to all of them as inner classes. We agree that this can be confusing!

Declaring an Inner Class

An *inner class*, also called a *member inner class*, is a non-`static` type defined at the member level of a class (the same level as the methods, instance variables, and constructors). Because they are not top-level types, they can use any of the four access levels, not just `public` and package access.

Inner classes have the following properties:

- Can be declared `public`, `protected`, package, or `private`
- Can extend a class and implement interfaces
- Can be marked `abstract` or `final`
- Can access members of the outer class, including `private` members

The last property is pretty cool. It means that the inner class can access variables in the outer class without doing anything special. Ready for a complicated way to print `Hi` three times?

```
1:  public class Home {
2:     private String greeting = "Hi";  // Outer class instance variable
3:
4:     protected class Room {           // Inner class declaration
5:        public int repeat = 3;
6:        public void enter() {
7:           for (int i = 0; i < repeat; i++) greet(greeting);
8:        }
9:        private static void greet(String message) {
10:          System.out.println(message);
11:       }
12:    }
13:
14:    public void enterRoom() {        // Instance method in outer class
15:       var room = new Room();        // Create the inner class instance
16:       room.enter();
17:    }
18:    public static void main(String[] args) {
19:       var home = new Home();        // Create the outer class instance
20:       home.enterRoom();
21: } }
```

An inner class declaration looks just like a stand-alone class declaration except that it happens to be located inside another class. Line 7 shows that the inner class just refers to `greeting` as if it were available in the `Room` class. This works because it is, in fact, available. Even though the variable is `private`, it is accessed within that same class.

Since an inner class is not `static`, it has to be called using an instance of the outer class. That means you have to create two objects. Line 19 creates the outer `Home` object, while line 15 creates the inner `Room` object. It's important to notice that line 15 doesn't require an explicit instance of `Home` because it is an instance method within `Home`. This works because `enterRoom()` is an instance method within the `Home` class. Both `Room` and `enterRoom()` are members of `Home`.

Nested Classes Can Now Have *static* Members

Eagle-eyed readers may have noticed that we included a `static` method in our inner `Room` class on line 9. In Java 11, this would have resulted in a compiler error. Previously, only `static` nested classes were allowed to include `static` methods. With the introduction of records in Java 16, the existing rule that prevented an inner class from having any `static` members (other than `static` constants) was removed. All four types of nested classes can now define `static` variables and methods!

Instantiating an Instance of an Inner Class

There is another way to instantiate `Room` that looks odd at first. Okay, well, maybe not just at first. This syntax isn't used often enough to get used to it:

```
20:    public static void main(String[] args) {
21:       var home = new Home();
22:       Room room = home.new Room();  // Create the inner class instance
23:       room.enter();
24:    }
```

Let's take a closer look at lines 21 and 22. We need an instance of `Home` to create a `Room`. We can't just call `new Room()` inside the `static main()` method, because Java won't know which instance of `Home` it is associated with. Java solves this by calling `new` as if it were a method on the `room` variable. We can shorten lines 21–23 to a single line:

```
21:       new Home().new Room().enter();  // Sorry, it looks ugly to us too!
```

Creating *.class* Files for Inner Classes

Compiling the `Home.java` class with which we have been working creates two class files. You should be expecting the `Home.class` file. For the inner class, the compiler creates `Home$Room.class`. You don't need to know this syntax for the exam. We mention it so that you aren't surprised to see files with `$` appearing in your directories. You do need to understand that multiple class files are created from a single `.java` file.

Referencing Members of an Inner Class

Inner classes can have the same variable names as outer classes, making scope a little tricky. There is a special way of calling `this` to say which variable you want to access. This is something you might see on the exam but, ideally, not in the real world.

In fact, you aren't limited to just one inner class. While the following is common on the exam, please never do this in code you write. Here is how to nest multiple classes and access a variable with the same name in each:

```
1:  public class A {
2:     private int x = 10;
3:     class B {
4:        private int x = 20;
5:        class C {
6:           private int x = 30;
7:           public void allTheX() {
8:              System.out.println(x);        // 30
9:              System.out.println(this.x);   // 30
10:             System.out.println(B.this.x); // 20
11:             System.out.println(A.this.x); // 10
12:    } } }
13:    public static void main(String[] args) {
14:       A a = new A();
15:       A.B b = a.new B();
16:       A.B.C c = b.new C();
17:       c.allTheX();
18: }}
```

Yes, this code makes us cringe too. It has two nested classes. Line 14 instantiates the outermost one. Line 15 uses the awkward syntax to instantiate a `B`. Notice that the type is `A.B`. We could have written `B` as the type because that is available at the member level of `A`. Java knows where to look for it. On line 16, we instantiate a `C`. This time, the `A.B.C` type is necessary to specify. `C` is too deep for Java to know where to look. Then line 17 calls a method on the instance variable `c`.

Lines 8 and 9 are the type of code that we are used to seeing. They refer to the instance variable on the current class—the one declared on line 6, to be precise. Line 10 uses `this` in a special way. We still want an instance variable. But this time, we want the one on the `B` class, which is the variable on line 4. Line 11 does the same thing for class `A`, getting the variable from line 2.

Inner Classes Require an Instance

Take a look at the following and see whether you can figure out why two of the three constructor calls do not compile:

```
public class Fox {
   private class Den {}
```

```
        public void goHome() {
           new Den();
        }
        public static void visitFriend() {
           new Den();  // DOES NOT COMPILE
        }
     }

     public class Squirrel {
        public void visitFox() {
           new Den();  // DOES NOT COMPILE
        }
     }
```

The first constructor call compiles because `goHome()` is an instance method, and therefore the call is associated with the `this` instance. The second call does not compile because it is called inside a `static` method. You can still call the constructor, but you have to explicitly give it a reference to a `Fox` instance.

The last constructor call does not compile for two reasons. Even though it is an instance method, it is not an instance method inside the `Fox` class. Adding a `Fox` reference would not fix the problem entirely, though. `Den` is `private` and not accessible in the `Squirrel` class.

Creating a *static* Nested Class

A *static nested class* is a `static` type defined at the member level. Unlike an inner class, a `static` nested class can be instantiated without an instance of the enclosing class. The trade-off, though, is that it can't access instance variables or methods declared in the outer class.

In other words, it is like a top-level class except for the following:

- The nesting creates a namespace because the enclosing class name must be used to refer to it.
- It can additionally be marked `private` or `protected`.
- The enclosing class can refer to the fields and methods of the `static` nested class.

Let's take a look at an example:

```
1: public class Park {
2:    static class Ride {
3:       private int price = 6;
4:    }
```

```
5:    public static void main(String[] args) {
6:       var ride = new Ride();
7:       System.out.println(ride.price);
8: } }
```

Line 6 instantiates the nested class. Since the class is `static`, you do not need an instance of `Park` to use it. You are allowed to access `private` instance variables, as shown on line 7.

Writing a Local Class

A *local class* is a nested class defined within a method. Like local variables, a local class declaration does not exist until the method is invoked, and it goes out of scope when the method returns. This means you can create instances only from within the method. Those instances can still be returned from the method. This is just how local variables work.

Local classes are not limited to being declared only inside methods. For example, they can be declared inside constructors and initializers. For simplicity, we limit our discussion to methods in this chapter.

Local classes have the following properties:

- They do not have an access modifier.
- They can be declared `final` or `abstract`.
- They have access to all fields and methods of the enclosing class (when defined in an instance method).
- They can access `final` and effectively final local variables.

Remember when we presented effectively final in Chapter 5? Well, we said it would come in handy later, and it's later! If you need a refresher on `final` and effectively final, turn back to Chapter 5 now. Don't worry; we'll wait!

Ready for an example? Here's a complicated way to multiply two numbers:

```
1:  public class PrintNumbers {
2:     private int length = 5;
3:     public void calculate() {
4:        final int width = 20;
5:        class Calculator {
6:           public void multiply() {
7:              System.out.print(length * width);
8:           }
9:        }
```

```
10:       var calculator = new Calculator();
11:       calculator.multiply();
12:    }
13:    public static void main(String[] args) {
14:       var printer = new PrintNumbers();
15:       printer.calculate();  // 100
16:    }
17: }
```

Lines 5–9 are the local class. That class's scope ends on line 12, where the method ends. Line 7 refers to an instance variable and a `final` local variable, so both variable references are allowed from within the local class.

Earlier, we made the statement that local variable references are allowed if they are `final` or effectively final. As an illustrative example, consider the following:

```
public void processData() {
   final int length = 5;
   int width = 10;
   int height = 2;
   class VolumeCalculator {
      public int multiply() {
         return length * width * height; // DOES NOT COMPILE
      }
   }
   width = 2;
}
```

The `length` and `height` variables are `final` and effectively final, respectively, so neither causes a compilation issue. On the other hand, the `width` variable is reassigned during the method, so it cannot be effectively final. For this reason, the local class declaration does not compile.

Why Can Local Classes Only Access *final* or Effectively Final Variables?

Earlier, we mentioned that the compiler generates a separate `.class` file for each inner class. A separate class has no way to refer to a local variable. However, if the local variable is `final` or effectively final, Java can handle it by passing *a copy of the value or reference variable* to the constructor of the local class. If it weren't `final` or effectively final, these tricks wouldn't work because the value could change after the copy was made.

Defining an Anonymous Class

An *anonymous class* is a specialized form of a local class that does not have a name. It is declared and instantiated all in one statement using the `new` keyword, a type name with parentheses, and a set of braces `{}`. Anonymous classes must extend an existing class or implement an existing interface. They are useful when you have a short implementation that will not be used anywhere else. Here's an example:

```
1:  public class ZooGiftShop {
2:     abstract class SaleTodayOnly {
3:        abstract int dollarsOff();
4:     }
5:     public int admission(int basePrice) {
6:        SaleTodayOnly sale = new SaleTodayOnly() {
7:           int dollarsOff() { return 3; }
8:        };  // Don't forget the semicolon!
9:        return basePrice - sale.dollarsOff();
10: } }
```

Lines 2–4 define an `abstract` class. Lines 6–8 define the anonymous class. Notice how this anonymous class does not have a name. The code says to instantiate a new `SaleTodayOnly` object. But wait: `SaleTodayOnly` is `abstract`. This is okay because we provide the class body right there—anonymously. In this example, writing an anonymous class is equivalent to writing a local class with an unspecified name that extends `SaleTodayOnly` and immediately uses it.

Pay special attention to the semicolon on line 8. We are declaring a local variable on these lines. Local variable declarations are required to end with semicolons, just like other Java statements—even if they are long and happen to contain an anonymous class.

Now we convert this same example to implement an `interface` instead of extending an `abstract` class:

```
1:  public class ZooGiftShop {
2:     interface SaleTodayOnly {
3:        int dollarsOff();
4:     }
5:     public int admission(int basePrice) {
6:        SaleTodayOnly sale = new SaleTodayOnly() {
7:           public int dollarsOff() { return 3; }
8:        };
9:        return basePrice - sale.dollarsOff();
10: } }
```

The most interesting thing here is how little has changed. Lines 2–4 declare an `interface` instead of an `abstract` class. Line 7 is `public` instead of using default access since interfaces require `public` methods. And that is it. The anonymous class is the same whether you implement an interface or extend a class! Java figures out which one you want automatically. Just remember that in this second example, an instance of a class is created on line 6, not an interface.

But what if we want to both implement an `interface` and extend a class? You can't do so with an anonymous class unless the class to extend is `java.lang.Object`. The `Object` class doesn't count in the rule. Remember that an anonymous class is just an unnamed local class. You can write a local class and give it a name if you have this problem. Then you can extend a class and implement as many interfaces as you like. If your code is this complex, a local class probably isn't the most readable option anyway.

You can even define anonymous classes outside a method body. The following may look like we are instantiating an interface as an instance variable, but the `{}` after the interface name indicates that this is an anonymous class implementing the interface:

```
public class Gorilla {
   interface Climb {}
   Climb climbing = new Climb() {};
}
```

Real World Scenario

Anonymous Classes and Lambda Expressions

Prior to Java 8, anonymous classes were frequently used for asynchronous tasks and event handlers. For example, the following shows an anonymous class used as an event handler in a JavaFX application:

```
var redButton = new Button();
redButton.setOnAction(new EventHandler<ActionEvent>() {
   public void handle(ActionEvent e) {
      System.out.println("Red button pressed!");
   }
});
```

Since the introduction of lambda expressions, anonymous classes are now often replaced with much shorter implementations:

```
Button redButton = new Button();
redButton.setOnAction(e -> System.out.println("Red button pressed!"));
```

We cover lambda expressions in detail in the next chapter.

Reviewing Nested Classes

For the exam, make sure that you know the information in Table 7.4 about which syntax rules are permitted in Java.

TABLE 7.4 Modifiers in nested classes

Permitted modifiers	Inner class	`static` nested class	Local class	Anonymous class
Access modifiers	All	All	None	None
`abstract`	Yes	Yes	Yes	No
`final`	Yes	Yes	Yes	No

You should also know the information in Table 7.5 about types of access. For example, the exam might try to trick you by having a `static` class access an outer class instance variable without a reference to the outer class.

TABLE 7.5 Nested class access rules

	Inner class	`static` nested class	Local class	Anonymous class
Can extend a class or implement any number of interfaces?	Yes	Yes	Yes	No—must have exactly one superclass or one interface
Can access instance members of enclosing class?	Yes	No	Yes (if declared in an instance method)	Yes (if declared in an instance method)
Can access local variables of enclosing method?	N/A	N/A	Yes (if `final` or effectively final)	Yes (if `final` or effectively final)

Understanding Polymorphism

We conclude this chapter with a discussion of polymorphism, the property of an object to take on many different forms. To put this more precisely, a Java object may be accessed using:

- A reference with the same type as the object
- A reference that is a superclass of the object
- A reference that defines an interface the object implements or inherits

Furthermore, a cast is not required if the object is being reassigned to a supertype or interface of the object. Phew, that's a lot! Don't worry; it'll make sense shortly.

Let's illustrate this polymorphism property with the following example:

```
public class Primate {
   public boolean hasHair() {
      return true;
   }
}

public interface HasTail {
   public abstract boolean isTailStriped();
}

public class Lemur extends Primate implements HasTail {
   public boolean isTailStriped() {
      return false;
   }
   public int age = 10;
   public static void main(String[] args) {
      Lemur lemur = new Lemur();
      System.out.println(lemur.age);

      HasTail hasTail = lemur;
      System.out.println(hasTail.isTailStriped());

      Primate primate = lemur;
      System.out.println(primate.hasHair());
   } }
```

This code compiles and prints the following output:

```
10
false
true
```

The most important thing to note about this example is that only one object, `Lemur`, is created. Polymorphism enables an instance of `Lemur` to be reassigned or passed to a method using one of its supertypes, such as `Primate` or `HasTail`.

Once the object has been assigned to a new reference type, only the methods and variables available to that reference type are callable on the object without an explicit cast. For example, the following snippets of code will not compile:

```
HasTail hasTail = new Lemur();
System.out.println(hasTail.age);             // DOES NOT COMPILE

Primate primate = new Lemur();
System.out.println(primate.isTailStriped()); // DOES NOT COMPILE
```

In this example, the reference `hasTail` has direct access only to methods defined with the `HasTail` interface; therefore, it doesn't know that the variable `age` is part of the object. Likewise, the reference `primate` has access only to methods defined in the `Primate` class, and it doesn't have direct access to the `isTailStriped()` method.

Object vs. Reference

In Java, all objects are accessed by reference, so as a developer you never have direct access to the object itself. Conceptually, though, you should consider the object as the entity that exists in memory, allocated by the Java Runtime Environment. Regardless of the type of the reference you have for the object in memory, the object itself doesn't change. For example, since all objects inherit `java.lang.Object`, they can all be reassigned to `java.lang.Object`, as shown in the following example:

```
Lemur lemur = new Lemur();
Object lemurAsObject = lemur;
```

Even though the `Lemur` object has been assigned to a reference with a different type, the object itself has not changed and still exists as a `Lemur` object in memory. What has changed, then, is our ability to access methods within the `Lemur` class with the `lemurAsObject` reference. Without an explicit cast back to `Lemur`, as you see in the next section, we no longer have access to the `Lemur` properties of the object.

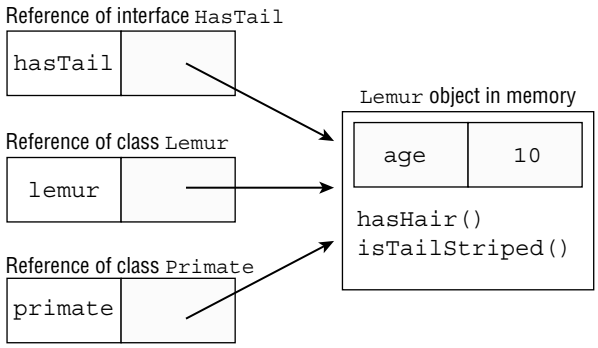
We can summarize this principle with the following two rules:

1. The type of the object determines which properties exist within the object in memory.
2. The type of the reference to the object determines which methods and variables are accessible to the Java program.

It therefore follows that successfully changing a reference of an object to a new reference type may give you access to new properties of the object; but remember, those properties existed before the reference change occurred.

Using the `Lemur` example, we illustrate this property in Figure 7.8.

FIGURE 7.8 Object vs. reference



As you can see in the figure, the same object exists in memory regardless of which reference is pointing to it. Depending on the type of the reference, we may only have access to certain methods. For example, the `hasTail` reference has access to the method `isTailStriped()` but doesn't have access to the variable `age` defined in the `Lemur` class. As you learn in the next section, it is possible to reclaim access to the variable `age` by explicitly casting the `hasTail` reference to a reference of type `Lemur`.

Real World Scenario

Using Interface References

When working with a group of objects that implement a common interface, it is considered a good coding practice to use an interface as the reference type. This is especially common with collections that you learn about in Chapter 9, "Collections and Generics." Consider the following method:

```
public void sortAndPrintZooAnimals(List<String> animals) {
   Collections.sort(animals);
   for(String a : animals) System.out.println(a);
}
```

This method sorts and prints `animals` in alphabetical order. At no point is this class interested in what the actual underlying object for `animals` is. It might be an `ArrayList` or another type. The point is, our code works on any of these types because we used the interface reference type rather than a class type.

Casting Objects

In the previous example, we created a single instance of a `Lemur` object and accessed it via superclass and interface references. Once we changed the reference type, though, we lost access to more specific members defined in the subclass that still exist within the object. We can reclaim those references by casting the object back to the specific subclass it came from:

```
Lemur lemur = new Lemur();

Primate primate = lemur;         // Implicit Cast to supertype

Lemur lemur2 = (Lemur)primate;   // Explicit Cast to subtype

Lemur lemur3 = primate;          // DOES NOT COMPILE (missing cast)
```

In this example, we first create a `Lemur` object and implicitly cast it to a `Primate` reference. Since `Lemur` is a subtype of `Primate`, this can be done without a cast operator. We then cast it back to a `Lemur` object using an explicit cast, gaining access to all of the methods and fields in the `Lemur` class. The last line does not compile because an explicit cast is required. Even though the object is stored in memory as a `Lemur` object, we need an explicit cast to assign it to `Lemur`.

Casting objects is similar to casting primitives, as you saw in Chapter 2, "Operators." When casting objects, you do not need a cast operator if casting to an inherited supertype. This is referred to as an *implicit cast* and applies to classes or interfaces the object inherits. Alternatively, if you want to access a subtype of the current reference, you need to perform an explicit cast with a compatible type. If the underlying object is not compatible with the type, then a `ClassCastException` will be thrown at runtime.

When reviewing a question on the exam that involves casting and polymorphism, be sure to remember what the instance of the object actually is. Then, focus on whether the compiler will allow the object to be referenced with or without explicit casts.

We summarize these concepts into a set of rules for you to memorize for the exam:

1. Casting a reference from a subtype to a supertype doesn't require an explicit cast.
2. Casting a reference from a supertype to a subtype requires an explicit cast.
3. At runtime, an invalid cast of a reference to an incompatible type results in a `ClassCastException` being thrown.
4. The compiler disallows casts to unrelated types.

Disallowed Casts

The first three rules are just a review of what we've said so far. The last rule is a bit more complicated. The exam may try to trick you with a cast that the compiler knows is not permitted (aka impossible). In the previous example, we were able to cast a `Primate` reference to a `Lemur` reference because `Lemur` is a subclass of `Primate` and therefore related. Consider this example instead:

```
public class Bird {}
```

```
public class Fish {
   public static void main(String[] args) {
      Fish fish = new Fish();
      Bird bird = (Bird)fish;  // DOES NOT COMPILE
   }
}
```

In this example, the classes `Fish` and `Bird` are not related through any class hierarchy that the compiler is aware of; therefore, the code will not compile. While they both extend `Object` implicitly, they are considered unrelated types since one cannot be a subtype of the other.

Casting Interfaces

While the compiler can enforce rules about casting to unrelated types for classes, it cannot always do the same for interfaces. Remember, instances support multiple inheritance, which limits what the compiler can reason about them. While a given class may not implement an interface, it's possible that some subclass may implement the interface. When holding a reference to a particular class, the compiler doesn't know which specific subtype it is holding.

Let's try an example. Do you think the following program compiles?

```
1: interface Canine {}
2: interface Dog {}
3: class Wolf implements Canine {}
4:
5: public class BadCasts {
6:    public static void main(String[] args) {
7:       Wolf wolfy = new Wolf();
8:       Dog badWolf = (Dog)wolfy;
9:    } }
```

In this program, a `Wolf` object is created and then assigned to a `Wolf` reference type on line 7. With interfaces, the compiler has limited ability to enforce many rules because even though a reference type may not implement an interface, one of its subclasses could. Therefore, it allows the invalid cast to the `Dog` reference type on line 8, even though `Dog` and `Wolf` are not related. Fear not, even though the code compiles, it still throws a `ClassCastException` at runtime.

This limitation aside, the compiler can enforce one rule around interface casting. The compiler does not allow a cast from an interface reference to an object reference if the object type cannot possibly implement the interface, such as if the class is marked `final`. For example, if the `Wolf` interface is marked `final` on line 3, then line 8 no longer compiles. The compiler recognizes that there are no possible subclasses of `Wolf` capable of implementing the `Dog` interface.

The *instanceof* Operator

In Chapter 3, we presented the `instanceof` operator with pattern matching. The `instanceof` operator can be used to check whether an object belongs to a particular class or interface and to prevent a `ClassCastException` at runtime. Consider the following example:

```
1: class Rodent {}
2:
3: public class Capybara extends Rodent {
4:    public static void main(String[] args) {
5:       Rodent rodent = new Rodent();
6:       var capybara = (Capybara)rodent;  // ClassCastException
7:    }
8: }
```

This program throws an exception on line 6. We can replace line 6 with the following.

```
6:       if(rodent instanceof Capybara c) {
7:          // Do stuff
8:       }
```

Now the code snippet doesn't throw an exception at runtime and performs the cast only if the `instanceof` operator is successful.

Just as the compiler does not allow casting an object to unrelated types, it also does not allow `instanceof` to be used with unrelated types. We can demonstrate this with our unrelated `Bird` and `Fish` classes:

```
public class Bird {}

public class Fish {
   public static void main(String[] args) {
      Fish fish = new Fish();
      if (fish instanceof Bird b) {  // DOES NOT COMPILE
         // Do stuff
      }
   }
}
```

Polymorphism and Method Overriding

In Java, polymorphism states that when you override a method, you replace all calls to it, even those defined in the parent class. As an example, what do you think the following code snippet outputs?

```
class Penguin {
   public int getHeight() { return 3; }
```

```
   public void printInfo() {
      System.out.print(this.getHeight());
   }
}

public class EmperorPenguin extends Penguin {
   public int getHeight() { return 8; }
   public static void main(String []fish) {
      new EmperorPenguin().printInfo();
   }
}
```

If you said 8, then you are well on your way to understanding polymorphism. In this example, the object being operated on in memory is an `EmperorPenguin`. The `getHeight()` method is overridden in the subclass, meaning all calls to it are replaced at runtime. Despite `printInfo()` being defined in the `Penguin` class, calling `getHeight()` on the object calls the method associated with the precise object in memory, not the current reference type where it is called. Even using the `this` reference, which is optional in this example, does not call the parent version because the method has been replaced.

Polymorphism's ability to replace methods at runtime via overriding is one of the most important properties of Java. It allows you to create complex inheritance models with subclasses that have their own custom implementation of overridden methods. It also means the parent class does not need to be updated to use the custom or overridden method. If the method is properly overridden, then the overridden version will be used in all places that it is called.

Remember, you can choose to limit polymorphic behavior by marking methods `final`, which prevents them from being overridden by a subclass.

Calling the Parent Version of an Overridden Method

Just because a method is overridden doesn't mean the parent method is completely inaccessible. We can use the `super` reference that you learned about in Chapter 6 to access it. How can you modify our previous example to print 3 instead of 8? You could try calling `super.getHeight()` in the parent `Penguin` class:

```
class Penguin {
   public int getHeight() { return 3; }
   public void printInfo() {
      System.out.print(super.getHeight());  // DOES NOT COMPILE
   }
}
```

Unfortunately, this does not compile, as `super` refers to the superclass of `Penguin`; in this case, `Object`. The solution is to override `printInfo()` in the child `EmperorPenguin` class and use `super` there.

```
public class EmperorPenguin extends Penguin {
   public int getHeight() { return 8; }
   public void printInfo() {
      System.out.print(super.getHeight());
   }
   public static void main(String []fish) {
      new EmperorPenguin().printInfo();  // 3
   }
}
```

Overriding vs. Hiding Members

While method overriding replaces the method everywhere it is called, `static` method and variable hiding do not. Strictly speaking, hiding members is not a form of polymorphism since the methods and variables maintain their individual properties. Unlike method overriding, hiding members is very sensitive to the reference type and location where the member is being used.

Let's take a look at an example:

```
class Penguin {
   public static int getHeight() { return 3; }
   public void printInfo() {
      System.out.println(this.getHeight());
   }
}

public class CrestedPenguin extends Penguin {
   public static int getHeight() { return 8; }
   public static void main(String... fish) {
      new CrestedPenguin().printInfo();
   }
}
```

The `CrestedPenguin` example is nearly identical to our previous `EmperorPenguin` example, although as you probably already guessed, it prints `3` instead of `8`. The `getHeight()` method is `static` and is therefore hidden, not overridden. The result is that calling `getHeight()` in `CrestedPenguin` returns a different value than calling it in

`Penguin`, even if the underlying object is the same. Contrast this with overriding a method, where it returns the same value for an object regardless of which class it is called in.

What about the fact that we used `this` to access a `static` method in `this.getHeight()`? As discussed in Chapter 5, while you are permitted to use an instance reference to access a `static` variable or method, doing so is often discouraged. The compiler will warn you when you access `static` members in a non-`static` way. In this case, the `this` reference had no impact on the program output.

Besides the location, the reference type can also determine the value you get when you are working with hidden members. Ready? Let's try a more complex example:

```
class Marsupial {
   protected int age = 2;
   public static boolean isBiped() {
      return false;
   } }

public class Kangaroo extends Marsupial {
   protected int age = 6;
   public static boolean isBiped() {
      return true;
   }

   public static void main(String[] args) {
      Kangaroo joey = new Kangaroo();
      Marsupial moey = joey;
      System.out.println(joey.isBiped());
      System.out.println(moey.isBiped());
      System.out.println(joey.age);
      System.out.println(moey.age);
   } }
```

The program prints the following:

```
true
false
6
2
```

In this example, only *one object* (of type `Kangaroo`) is created and stored in memory! Since `static` methods can only be hidden, not overridden, Java uses the reference type to determine which version of `isBiped()` should be called, resulting in `joey.isBiped()` printing `true` and `moey.isBiped()` printing `false`.

Likewise, the `age` variable is hidden, not overridden, so the reference type is used to determine which value to output. This results in `joey.age` returning `6` and `moey.age` returning `2`.

For the exam, make sure you understand these examples, as they show how hidden and overridden methods are fundamentally different. In practice, overriding methods is the cornerstone of polymorphism and an extremely powerful feature.

Real World Scenario

Don't Hide Members in Practice

Although Java allows you to hide variables and `static` methods, it is considered an extremely poor coding practice. As you saw in the previous example, the value of the variable or method can change depending on what reference is used, making your code very confusing, difficult to follow, and challenging for others to maintain. This is further compounded when you start modifying the value of the variable in both the parent and child methods, since it may not be clear which variable you're updating.

When you're defining a new variable or `static` method in a child class, it is considered good coding practice to select a name that is not already used by an inherited member. Redeclaring `private` methods and variables is considered less problematic, though, because the child class does not have access to the variable in the parent class to begin with.

Summary

In this chapter, we presented numerous topics in advanced object-oriented design, covering many top-level types beyond classes. We started with interfaces and described how they can support multiple inheritance. Remember, interfaces and their members can include a number of implicit modifiers inserted by the compiler automatically. We then covered all six types of interface members you need to know for the exam: `abstract` methods, `static` constants, `default` methods, `static` methods, `private` methods, and `private static` methods.

We next moved on to enums, which are compile-time constant properties. Simple enums are composed of a list of values, while complex enums can include constructors, methods, and fields. Enums can also be used in `switch` statements and expressions. When an enum method is marked `abstract`, each enum value must provide an implementation.

Moving on to new topics in Java, we covered sealed classes and how they allow classes to function like enumerated types in which only certain subclasses are permitted. For the exam,

it's important to remember that the subclasses of a sealed class must be marked `final`, `sealed`, or `non-sealed`. If the subclasses of the sealed class are defined in the same file, then the `permits` clause may be omitted in the sealed class declaration. Finally, sealed interfaces may be used to limit which classes can implement an interface, which interfaces may extend an interface, or both.

Records are another new feature available in Java. Records are a compact way of declaring an immutable and encapsulated POJO in which the compiler adds a lot of the boilerplate code for you. Remember, encapsulation is the practice of preventing external callers from accessing the internal components of an object. Records include automatic creation of the accessor methods, a long constructor, and useful implementations of `equals()`, `hashCode()`, and `toString()`. Records can include overloaded and compact constructors to support data validation and transformation. Records do not permit instance variables, since this could break immutability, but they do allow methods, `static` members, and nested types.

We then moved on to nested types. For simplicity, we focused on nested classes and covered each of the four types. An inner class requires an instance of the outer class to use, while a `static` nested class does not. A local class is commonly defined within a method or block. Local classes can only access local variables that are `final` and effectively final. Anonymous classes are a special type of local class that does not have a name. Anonymous classes are required to extend exactly one class or implement one interface. Inner, local, and anonymous classes can access `private` members of the class in which they are defined, provided the latter two are used inside an instance method.

We concluded this chapter with a discussion of polymorphism, which is central to the Java language, and showed how objects can be accessed in a variety of forms. Make sure you understand when casts are needed for accessing objects, and be able to spot the difference between compile-time and runtime cast problems.

Exam Essentials

Be able to write code that creates, extends, and implements interfaces. Interfaces are specialized abstract types that focus on abstract methods and constant variables. An interface may extend any number of interfaces and, in doing so, inherits their abstract methods. An interface cannot extend a class, nor can a class extend an interface. A class may implement any number of interfaces.

Know which interface methods an interface method can reference. Non-`static` `private`, `default`, and `abstract` interface methods are associated with an instance of an interface. Non-`static` `private` and `default` interface methods may reference any method within the interface declaration. Alternatively, `static` interface methods are associated with class membership and can only reference other `static` members. Finally, `private` methods can only be referenced within the interface declaration.

Be able to create and use enum types. An enum is a data structure that defines a list of values. If the enum does not contain any other elements, the semicolon (`;`) after the values is optional. An enum can be used in `switch` statements and contain instance variables, constructors, and methods. Enum constructors are implicitly `private`. Enums can include methods, both as members or within individual enum values. If the enum declares an `abstract` method, each enum value must implement it.

Be able to recognize when sealed classes are being correctly used. A sealed class is one that defines a list of permitted subclasses that extend it. Be able to use the correct modifier (`final`, `sealed`, or `non-sealed`) with sealed classes. Understand when the `permits` clause may be excluded.

Identify properly encapsulated classes. Instance variables in encapsulated classes are `private`. All code that retrieves the value or updates it uses methods. Encapsulated classes may include accessor (getter) or mutator (setter) methods, although this is not required.

Understand records and know which members the compiler is adding automatically. Records are encapsulated and immutable types in which the compiler inserts a long constructor, accessor methods, and useful implementations of `equals()`, `hashCode()`, and `toString()`. Each of these elements may be overridden. Be able to recognize compact constructors and know that they are used only for validation and transformation of constructor parameters, not for accessing fields. Recognize that when a record is declared with an instance member, it does not compile.

Be able to declare and use nested classes. There are four types of nested types: inner classes, `static` classes, local classes, and anonymous classes. Instantiating an inner class requires an instance of the outer class. On the other hand, `static` nested classes can be created without a reference to the outer class. Local and anonymous classes cannot be declared with an access modifier. Anonymous classes are limited to extending a single class or implementing one interface.

Understand polymorphism. An object may take on a variety of forms, referred to as polymorphism. The object is viewed as existing in memory in one concrete form but is accessible in many forms through reference variables. Changing the reference type of an object may grant access to new members, but the members always exist in memory.