Chapter 9 Advanced Class Design

OCP EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- **Reusing Implementations Through Inheritance**
  - Create and extend abstract classes
- **Programming Abstractly Through Interfaces**
  - Create and implement interfaces
  - Distinguish class inheritance from interface inheritance including abstract classes

In Chapter 8, "Class Design," we showed you how to create classes utilizing inheritance and polymorphism. In this chapter, we will continue our discussion of class design starting with abstract classes. By creating abstract class definitions, you're defining a platform that other developers can extend and build on top of. We'll then move on to interfaces and show how to use them to design a standard set of methods across classes with varying implementations. Finally, we'll conclude this chapter with a brief presentation of inner classes.

Creating Abstract Classes

We start our discussion of advanced class design with abstract classes. As you will see, abstract classes have important uses in defining a framework that other developers can use.

Introducing Abstract Classes

In Chapter 8, you learned that a subclass can override an inherited method defined in a parent class. Overriding a method potentially changes the behavior of a method in the parent class. For example, take a look at the following `Bird` class and its `Stork` subclass:

```
class Bird {
   public String getName() { return null; }
   public void printName() {
      System.out.print(getName());
   }
}

public class Stork extends Bird {
   public String getName() { return "Stork!"; }
   public static void main(String[] args) {
      new Stork().printName();
   }
}
```

This program prints `Stork!` at runtime. Notice that the `getName()` method is overridden in the subclass. Even though the implementation of `printName()` is defined in the `Bird` class, the fact that `getName()` is overridden in the subclass means it is replaced everywhere, even in the parent class.

Let's take this one step further. Suppose you want to define a `Bird` class that other developers can extend and use, but you want the developers to specify the particular type of `Bird`. Also, rather than having the `Bird` version of `getName()` return `null` (or throw an exception), you want to ensure every class that extends `Bird` is required to provide its own overridden version of the `getName()` method.

Enter abstract classes. An *abstract class* is a class that cannot be instantiated and may contain abstract methods. An *abstract method* is a method that does not define an implementation when it is declared. Both abstract classes and abstract methods are denoted with the `abstract` modifier. Compare our previous implementation with this new one using an abstract `Bird` class:

```
abstract class Bird {
   public abstract String getName();
   public void printName() {
      System.out.print(getName());
   }
}

public class Stork extends Bird {
   public String getName() { return "Stork!"; }
   public static void main(String[] args) {
      new Stork().printName();
   }
}
```

What's different? First, the `Bird` class is marked `abstract`. Next, the `getName()` method in `Bird` is also marked `abstract`. Finally, the implementation of `getName()`, including the braces (`{}`), have been replaced with a single semicolon (`;`).

What about the `Stork` class? It's exactly the same as before. While it may look the same, though, the rules around how the class must be implemented have changed. In particular, the `Stork` class *must* now override the abstract `getName()` method. For example, the following implementation does not compile because `Stork` does not override the required abstract `getName()` method:

```
public class Stork extends Bird {}  // DOES NOT COMPILE
```

While these differences may seem small, imagine the `Bird` and `Stork` class are each written by different people. By one person marking `getName()` as `abstract` in the `Bird` class, they are sending a message to the other developer writing the `Stork` class: “Hey, to use this class, you need to write a `getName()` method!”

An abstract class is most commonly used when you want another class to inherit properties of a particular class, but you want the subclass to fill in some of the implementation details. In our example, the author of the `Bird` class wrote the `printName()` method but did not know what it was going to do at runtime, since the `getName()` implementation had yet to be provided.

OVERRIDE VS. IMPLEMENT

Oftentimes, when an abstract method is overridden in a subclass, it is referred to as implementing the method. It is described this way because the subclass is providing an implementation for a method that does not yet have one. While we tend to use the terms *implement* and *override* interchangeably for abstract methods, the term *override* is more accurate.

When overriding an abstract method, all of the rules you learned about overriding methods in Chapter 8 are applicable. For example, you can override an abstract method with a covariant return type. Likewise, you can declare new unchecked exceptions but not checked exceptions in the overridden method. Furthermore, you can override an abstract method in one class and then override it again in a subclass of that class.

The method override rules apply whether the abstract method is declared in an abstract class or, as we shall see later in this chapter, an interface. We will continue to use override and implement interchangeably in this chapter, as this is common in software development. Just remember that providing an implementation for an abstract method is considered a method override and all of the associated rules for overriding methods apply.

Earlier, we said that an abstract class is one that cannot be instantiated. This means that if you attempt to instantiate it, the compiler will report an exception, as in this example:

```
abstract class Alligator {
   public static void main(String... food) {
      var a = new Alligator();  // DOES NOT COMPILE
   }
}
```

An abstract class can be initialized, but only as part of the instantiation of a nonabstract subclass.

Defining Abstract Methods

As you saw in the previous example, an abstract class may include nonabstract methods, in this case with the `printName()` method. In fact, an abstract class can include all of the same members as a nonabstract class, including variables, `static` and instance methods, and inner classes. As you will see in the next section, abstract classes can also include constructors.

One of the most important features of an abstract class is that it is not actually required to include any abstract methods. For example, the following code compiles even though it doesn't define any abstract methods:

```
public abstract class Llama {
   public void chew() {}
}
```

Although an abstract class doesn't have to declare any abstract methods, an abstract method can only be defined in an abstract class (or an interface, as you will see shortly). For example, the following code won't compile because the class is not marked `abstract`:

```
public class Egret {  // DOES NOT COMPILE
   public abstract void peck();
}
```

The exam creators like to include invalid class declarations like the `Egret` class, which mixes nonabstract classes with abstract methods. If you see a class that contains an abstract method, make sure the class is marked `abstract`.

Like the `final` modifier, the `abstract` modifier can be placed before or after the access modifier in class and method declarations, as shown in this `Tiger` class:

```
abstract public class Tiger {
   abstract public int claw();
}
```

There are some restrictions on the placement of the `abstract` modifier. The `abstract` modifier cannot be placed after the `class` keyword in a class declaration, nor after the return type in a method declaration. The following `Jackal` and `howl()` declarations do not compile for these reasons:

```
public class abstract Jackal {  // DOES NOT COMPILE
   public int abstract howl();  // DOES NOT COMPILE
}
```

NOTE

It is not possible to define an abstract method that has a body, or default implementation. You can still define a method with a body—you just can't mark it as `abstract`. As long as you do not mark the method as `final`, the subclass has the option to override an inherited method.

Constructors in Abstract Classes

Even though abstract classes cannot be instantiated, they are still initialized through constructors by their subclasses. For example, does the following program compile?

```
abstract class Bear {
   abstract CharSequence chew();
   public Bear() {
      System.out.println(chew());  // Does this compile?
   }
}

public class Panda extends Bear {
   String chew() { return "yummy!"; }
   public static void main(String[] args) {
      new Panda();
   }
}
```

Using the constructor rules you learned in Chapter 8, the compiler inserts a default no-argument constructor into the `Panda` class, which first calls `super()` in the `Bear` class. The `Bear` constructor is only called when the abstract class is being initialized through a subclass; therefore, there is an implementation of `chew()` at the time the constructor is called. This code compiles and prints `yummy!` at runtime.

For the exam, remember that abstract classes are initialized with constructors in the same way as nonabstract classes. For example, if an abstract class does not provide a constructor, the compiler will automatically insert a default no-argument constructor.

The primary difference between a constructor in an abstract class and a nonabstract class is that a constructor in abstract class can be called only when it is being initialized by a nonabstract subclass. This makes sense, as abstract classes cannot be instantiated.

Invalid Abstract Method Declarations

The exam writers are also fond of questions with methods marked as `abstract` for which an implementation is also defined. For example, can you see why each of the following methods does not compile?

```
public abstract class Turtle {
   public abstract long eat()      // DOES NOT COMPILE
   public abstract void swim() {}; // DOES NOT COMPILE
   public abstract int getAge() {  // DOES NOT COMPILE
      return 10;
   }
   public void sleep;              // DOES NOT COMPILE
   public void goInShell();        // DOES NOT COMPILE
}
```

The first method, `eat()`, does not compile because it is marked `abstract` but does not end with as semicolon (`;`). The next two methods, `swim()` and `getAge()`, do not compile because they are marked `abstract`, but they provide an implementation block enclosed in braces (`{}`). For the exam, remember that an abstract method declaration must end in a semicolon without any braces. The next method, `sleep`, does not compile because it is missing parentheses, `()`, for method arguments. The last method, `goInShell()`, does not compile because it is not marked `abstract` and therefore must provide a body enclosed in braces.

Make sure you understand why each of the previous methods does not compile and that you can spot errors like these on the exam. If you come across a question on the exam in which a class or method is marked `abstract`, make sure the class is properly implemented before attempting to solve the problem.

Invalid Modifiers

In Chapter 7, "Methods and Encapsulation," you learned about various modifiers for methods and classes. In this section, we review the `abstract` modifier and which modifiers it is not compatible with.

abstract and ***final*** Modifiers

What would happen if you marked a class or method both `abstract` and `final`? If you mark something `abstract`, you are intending for someone else to extend or implement it. But, if you mark something `final`, you are preventing anyone from extending or implementing it. These concepts are in direct conflict with each other.

Due to this incompatibility, Java does not permit a class or method to be marked both `abstract` and `final`. For example, the following code snippet will not compile:

```
public abstract final class Tortoise {  // DOES NOT COMPILE
   public abstract final void walk();   // DOES NOT COMPILE
}
```

In this example, neither the class or method declarations will compile because they are marked both `abstract` and `final`. The exam doesn't tend to use `final` modifiers on classes or methods often, so if you see them, make sure they aren't used with the `abstract` modifier.

abstract and ***private*** Modifiers

A method cannot be marked as both `abstract` and `private`. This rule makes sense if you think about it. How would you define a subclass that implements a required method if the method is not inherited by the subclass? The answer is you can't, which is why the compiler will complain if you try to do the following:

```
public abstract class Whale {
   private abstract void sing();  // DOES NOT COMPILE
}

public class HumpbackWhale extends Whale {
   private void sing() {
      System.out.println("Humpback whale is singing");
   }
}
```

In this example, the abstract method `sing()` defined in the parent class `Whale` is not visible to the subclass `HumpbackWhale`. Even though `HumpbackWhale` does provide an implementation, it is not considered an override of the abstract method since the abstract method is not inherited. The compiler recognizes this in the parent class and reports an error as soon as `private` and `abstract` are applied to the same method.

NOTE

While it is not possible to declare a method `abstract` and `private`, it is possible (albeit redundant) to declare a method `final` and `private`.

If we changed the access modifier from `private` to `protected` in the parent class `Whale`, would the code compile? Let's take a look:

```
public abstract class Whale {
   protected abstract void sing();
}

public class HumpbackWhale extends Whale {
   private void sing() {  // DOES NOT COMPILE
      System.out.println("Humpback whale is singing");
   }
}
```

In this modified example, the code will still not compile, but for a completely different reason. If you remember the rules for overriding a method, the subclass cannot reduce the visibility of the parent method, `sing()`. Because the method is declared `protected` in the parent class, it must be marked as `protected` or `public` in the child class. Even with abstract methods, the rules for overriding methods must be followed.

***abstract* and *static* Modifiers**

As you saw in Chapter 8, a `static` method cannot be overridden. It is defined as belonging to the class, not an instance of the class. If a `static` method cannot be overridden, then it follows that it also cannot be marked `abstract` since it can never be implemented. For example, the following class does not compile:

```
abstract class Hippopotamus {
   abstract static void swim();  // DOES NOT COMPILE
}
```

For the exam, make sure you know which modifiers can and cannot be used with one another, especially for abstract classes and interfaces.

Creating a Concrete Class

An abstract class becomes usable when it is extended by a concrete subclass. A *concrete class* is a nonabstract class. The first concrete subclass that extends an abstract class is required to implement all inherited abstract methods. This includes implementing any inherited abstract methods from inherited interfaces, as we will see later in this chapter.

When you see a concrete class extending an abstract class on the exam, check to make sure that it implements all of the required abstract methods. Can you see why the following `Walrus` class does not compile?

```
public abstract class Animal {
   public abstract String getName();
}

public class Walrus extends Animal { // DOES NOT COMPILE
}
```

In this example, we see that `Animal` is marked as `abstract` and `Walrus` is not, making `Walrus` a concrete subclass of `Animal`. Since `Walrus` is the first concrete subclass, it must implement all inherited abstract methods— `getName()` in this example. Because it doesn't, the compiler reports an error with the declaration of `Walrus`.

We highlight the *first* concrete subclass for a reason. An abstract class can extend a nonabstract class, and vice versa. Any time a concrete class is extending an abstract class, it must implement all of the methods that are inherited as abstract. Let's illustrate this with a set of inherited classes:

```
abstract class Mammal {
   abstract void showHorn();
   abstract void eatLeaf();
}

abstract class Rhino extends Mammal {
   void showHorn() {}
}
```

```
public class BlackRhino extends Rhino {
   void eatLeaf() {}
}
```

In this example, the `BlackRhino` class is the first concrete subclass, while the `Mammal` and `Rhino` classes are abstract. The `BlackRhino` class inherits the `eatLeaf()` method as abstract and is therefore required to provide an implementation, which it does.

What about the `showHorn()` method? Since the parent class, `Rhino`, provides an implementation of `showHorn()`, the method is inherited in the `BlackRhino` as a nonabstract method. For this reason, the `BlackRhino` class is permitted but not required to override the `showHorn()` method. The three classes in this example are correctly defined and compile.

What if we changed the `Rhino` declaration to remove the abstract modifier?

```
class Rhino extends Mammal {  // DOES NOT COMPILE
   void showHorn() {}
}
```

By changing `Rhino` to a concrete class, it becomes the first nonabstract class to extend the abstract `Mammal` class. Therefore, it must provide an implementation of both the `showHorn()` and `eatLeaf()` methods. Since it only provides one of these methods, the modified `Rhino` declaration does not compile.

Let's try one more example. The following concrete class `Lion` inherits two abstract methods, `getName()` and `roar()`:

```
public abstract class Animal {
   abstract String getName();
}

public abstract class BigCat extends Animal {
   protected abstract void roar();
}

public class Lion extends BigCat {
   public String getName() {
      return "Lion";
   }
   public void roar() {
      System.out.println("The Lion lets out a loud ROAR!");
   }
}
```

In this sample code, `BigCat` extends `Animal` but is marked as `abstract`; therefore, it is not required to provide an implementation for the `getName()` method. The class `Lion` is not marked as `abstract`, and as the first concrete subclass, it must implement all of the inherited abstract methods not defined in a parent class. All three of these classes compile successfully.

Reviewing Abstract Class Rules

For the exam, you should know the following rules about abstract classes and abstract methods. While it may seem like a lot to remember, most of these rules are pretty straightforward. For example, marking a class or method `abstract` and `final` makes it unusable. Be sure you can spot contradictions such as these if you come across them on the exam.

Abstract Class Definition Rules

1. Abstract classes cannot be instantiated.
2. All top-level types, including abstract classes, cannot be marked `protected` or `private`.
3. Abstract classes cannot be marked `final`.

4. Abstract classes may include zero or more abstract and nonabstract methods.
5. An abstract class that `extends` another abstract class inherits all of its abstract methods.
6. The first concrete class that `extends` an abstract class must provide an implementation for all of the inherited abstract methods.
7. Abstract class constructors follow the same rules for initialization as regular constructors, except they can be called only as part of the initialization of a subclass.

These rules for abstract methods apply regardless of whether the abstract method is defined in an abstract class or interface.

Abstract Method Definition Rules

1. Abstract methods can be defined only in abstract classes or interfaces.
2. Abstract methods cannot be declared `private` or `final`.
3. Abstract methods must not provide a method body/implementation in the abstract class in which they are declared.
4. Implementing an abstract method in a subclass follows the same rules for overriding a method, including covariant return types, exception declarations, etc.

Implementing Interfaces

Although Java doesn't allow multiple inheritance of state, it does allow a class to implement any number of interfaces. An *interface* is an abstract data type are that declares a list of abstract methods that any class implementing the interface must provide. An interface can also include constant variables. Both abstract methods and constant variables included with an interface are implicitly assumed to be `public`.

INTERFACES AND NONABSTRACT METHODS

For the 1Z0-815 exam, you only need to know about two members for interfaces: abstract methods and constant variables. With Java 8, interfaces were updated to include `static` and `default` methods. A `default` method is one in which the interface method has a body and is not marked `abstract`. It was added for backward compatibility, allowing an older class to use a new version of an interface that contains a new method, without having to modify the existing class.

In Java 9, interfaces were updated to support `private` and `private static` methods. Both of these types were added for code reusability within an interface declaration and cannot be called outside the interface definition.

When you study for the 1Z0-816 exam, you will need to know about other kinds of interface members. For the 1Z0-815 exam, you only need to know about abstract methods and constant variables.

Defining 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 9.1 for a proper interface declaration.

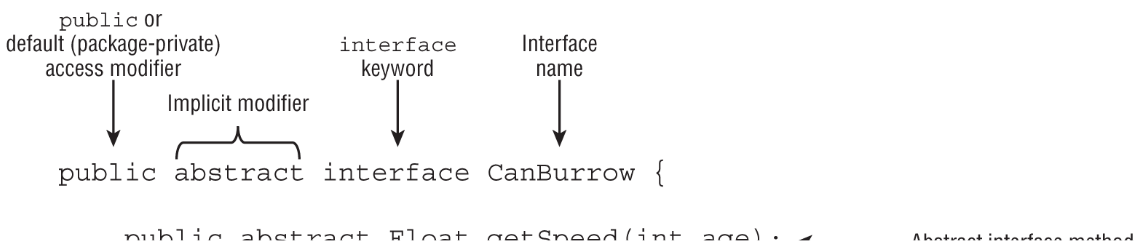

public or default (package-private) access modifier
Implicit modifier
interface keyword
Interface name
public abstract interface CanBurrow {
public abstract Float getSpeed(int age);
Abstract interface method

FIGURE 9.1 Defining an interface

In Figure 9.1, our interface declaration includes a constant variable and an abstract method. 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 9.1 also includes an abstract method that, like an interface variable, is assumed to be `public`.

NOTE

For brevity, we sometimes say "an instance of an interface" 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 adds to a class, interface, method, or variable declaration. For example, an interface is always considered to be `abstract`, even if it is not marked so. We'll cover rules and examples for implicit modifiers in more detail later in the chapter.

Let's start with an example. Imagine 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:

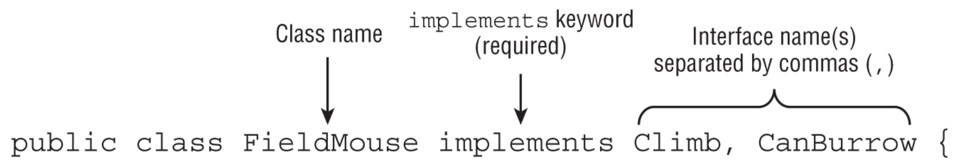
```
public class Biped {
   public static void main(String[] args) {
      var e = new WalksOnTwoLegs();         // DOES NOT COMPILE
   }
}

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:

```
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 9.2.


Class name
implements keyword (required)
Interface name(s) separated by commas (,)

```
public class FieldMouse implements Climb, CanBurrow {
```

FIGURE 9.2 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. The access modifier of the interface method is assumed to be `public` in `Climb`, although the concrete class `FieldMouse` must explicitly declare it.

As shown in Figure 9.2, 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` also implements the `CanBurrow` interface that we saw in Figure 9.1. In this manner, the class overrides two abstract methods at the same time with one method declaration. You'll learn more about duplicate and compatible interface methods shortly.

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

```
interface Nocturnal {}

public interface HasBigEyes extends Nocturnal {}
```

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

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

interface CanFly {
   public void flap();
}

interface HasBigEyes extends Nocturnal, CanFly {}

public class Owl implements Nocturnal, CanFly {
   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 that a class implementing them must follow. They also include various `static` members, including constants that do not require an instance of the class to use.

Many of the rules for class declarations also apply to interfaces including the following:

- A Java file may have at most one `public` top-level class or interface, and it must match the name of the file.
- A top-level class or interface can only be declared with `public` or package-private access.

It may help to think of an interface as a specialized abstract class, as many of the rules carry over. Just remember that an interface does not follow the same rules for single inheritance and instance initialization with constructors, as a class does.

WHAT ABOUT ENUMS?

In this section, we described how a Java class can have at most one `public` top-level element, a class or interface. This `public` top-level element could also be an enumeration, or *enum* for short. An enum is a specialized type that defines a set of fixed values. It is declared with the `enum` keyword. The following demonstrates a simple example of an enum for `Color`:

```
public enum Color {
   RED, YELLOW, BLUE, GREEN, ORANGE, PURPLE
}
```

Like classes and interfaces, enums can have more complex formations including methods, `private` constructors, and instance variables.

Luckily for you, enums are out of scope for the 1Z0-815 exam. Like some of the more advanced interface members we described earlier, you will need to study enums when preparing for the 1Z0-816 exam.

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 8. 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 assumed to be `abstract`.
- Interface variables are assumed to be `public`, `static`, and `final`.

- Interface methods without a body are assumed to be `abstract` and `public`.

For 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 they do 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 assumed to be `public`, then 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.

While issues with `private` and `protected` access modifiers in interfaces are easy to spot, what about the package-private access? For example, what is the access level of the following two elements `volume` and `start()` ?

```
public interface Sing {
   float volume = 10;
   abstract void start();
}
```

If you said `public` , then you are correct! When working with class members, omitting the access modifier indicates default (package-private) access. When working with interface members, though, the lack of access modifier always indicates `public` access.

Let's try another one. Which line or lines of this top-level interface declaration do not compile?

```
1: private final interface Crawl {
2:    String distance;
3:    private int MAXIMUM_DEPTH = 100;
4:    protected abstract boolean UNDERWATER = false;
5:    private void dig(int depth);
6:    protected abstract double depth();
7:    public final void surface(); }
```

Every single line of this example, including the interface declaration, does not compile! Line 1 does not compile for two reasons. First, it is marked as `final`, which cannot be applied to an interface since it conflicts with the implicit `abstract` keyword. Next, it is marked as `private`, which conflicts with the `public` or package-private access for top-level interfaces.

Line 2 does not compile because the `distance` variable is not initialized. Remember that interface variables are assumed to be `static final` constants and initialized when they are declared. Lines 3 and 4 do not compile because interface variables are also assumed to be `public`, and the access modifiers on these lines conflict with this. Line 4 also does not compile because variables cannot be marked `abstract`.

Next, lines 5 and 6 do not compile because all interface abstract methods are assumed to be `public` and marking them as `private` or `protected` is not permitted. Finally, the last line doesn't compile because the method is marked as `final`, and since interface methods without a body are assumed to be `abstract`, the compiler throws an exception for using both `abstract` and `final` keywords on a method.

Study these examples with conflicting modifiers carefully and make sure you know why they fail to compile. On the exam, you are likely to get at least one question in which an interface includes a member that contains an invalid modifier.

Differences between Interfaces and Abstract Classes

Even though abstract classes and interfaces are both considered abstract types, only interfaces make use of implicit modifiers. This means that an abstract class and interface with similar declarations may have very different properties. For example, how do the `play()` methods differ in the following two definitions?

```
abstract class Husky {
   abstract void play();
}

interface Poodle {
   void play();
}
```

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? Even though neither has an access modifier, they do not have the same access level. The `play()` method in `Husky` class is considered default (package-private), whereas the method in the `Poodle` interface is assumed to be `public`. This is especially important when you create classes that inherit these definitions. For example, can you spot anything wrong with the following class definitions that use our abstract types?

```
class Webby extends Husky {
   void play() {}
}

class Georgette implements Poodle {
   void play() {}
}
```

The `Webby` class compiles, but the `Georgette` class does not. Even though the two method implementations are identical, the method in the `Georgette` class breaks the rules of method overriding. From the `Poodle` interface, the inherited abstract method is assumed to be `public`. The definition of `play()` in the `Georgette` class therefore reduces the visibility of a method from `public` to package-private, resulting in a compiler error. The following is the correct implementation of the `Georgette` class:

```
class Georgette implements Poodle {
   public void play() {}
}
```

Inheriting an Interface

An interface can be inherited in one of three ways.

- An interface can extend another interface.
- A class can implement an interface.
- A class can extend another class whose ancestor implements an interface.

When an interface is inherited, all of the abstract methods are inherited. Like we saw with abstract classes, if the type inheriting the interface is also abstract, such as an interface or abstract class, it is not required to implement the interface methods. On the other hand, the first concrete subclass that inherits the interface must implement all of the inherited abstract methods.

We illustrate this principle in Figure 9.3. How many abstract methods does the concrete `Swan` class inherit?

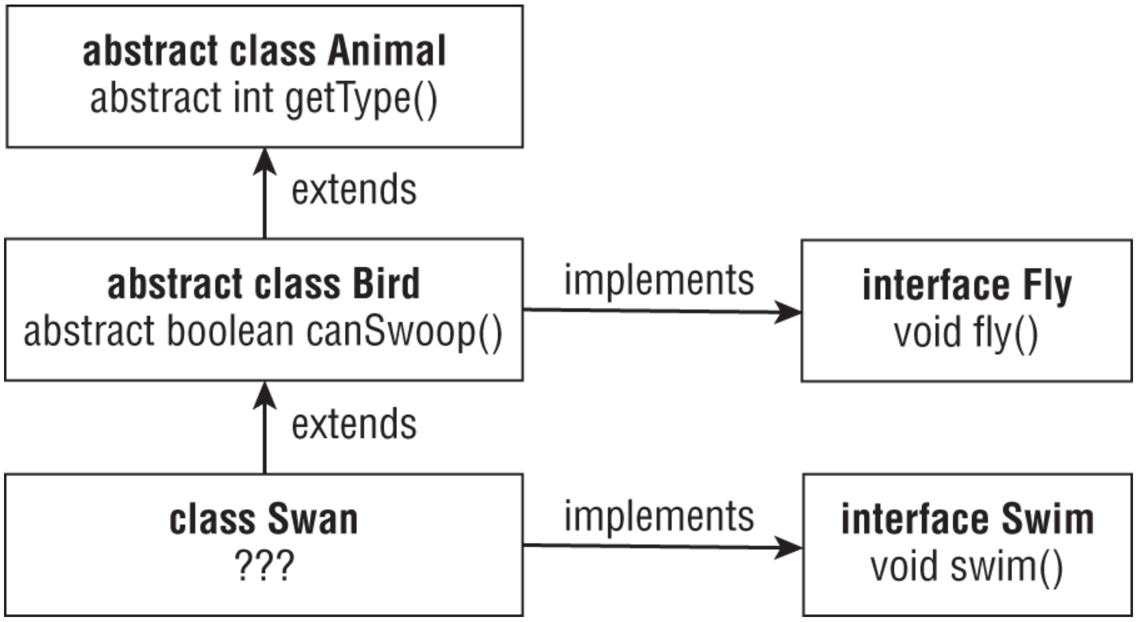


FIGURE 9.3 **Interface Inheritance**

The `Swan` class inherits four methods: the `public fly()` and `swim()` methods, along with the package-private `getType()` and `canSwoop()` methods.

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 is not required to implement any of the abstract methods it inherits from the `HasTail` and `HasWhiskers` because it is marked `abstract`. The concrete class `CommonSeal`, which extends `HarborSeal`, is required to implement all inherited abstract methods. In this example, `CommonSeal` doesn't provide an implementation for the inherited abstract interface methods, so `CommonSeal` doesn't compile.

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 that 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 definitions. The following is the only valid syntax for relating classes and interfaces in their declarations:

```
class1 extends class2
interface1 extends interface2, interface3, ...
class1 implements interface2, interface3, ...
```

Duplicate Interface Method Declarations

Since Java allows for multiple inheritance via interfaces, you might be wondering what will happen if you define a class that inherits from two interfaces that contain the same `abstract` method.

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

public interface Omnivore {
   public void eatPlants();
   public void eatMeat();
}
```

In this scenario, the signatures for the two interface methods `eatPlants()` are duplicates. As they have identical method declarations, they are also considered *compatible*. By compatibility, we mean that the compiler can resolve the differences between the two declarations without finding any conflicts. You can define a class that fulfills both interfaces simultaneously.

```
public class Bear implements Herbivore, Omnivore {
   public void eatMeat() {
      System.out.println("Eating meat");
   }
   public void eatPlants() {
      System.out.println("Eating plants");
   }
}
```

As we said earlier, interfaces simply define a set of rules that a class implementing them must follow. If two `abstract` interface methods have identical behaviors—or in this case the same method declaration—you just need to be able to create a single method that overrides both inherited abstract methods at the same time.

What if the duplicate methods have different signatures? If the method name is the same but the input parameters are different, there is no conflict because this is considered a method overload. We demonstrate this principle in the following example:

```
public interface Herbivore {
   public int eatPlants(int quantity);
}

public interface Omnivore {
   public void eatPlants();
}

public class Bear implements Herbivore, Omnivore {
   public int eatPlants(int quantity) {
      System.out.println("Eating plants: "+quantity);
      return quantity;
   }
   public void eatPlants() {
      System.out.println("Eating plants");
   }
}
```

In this example, we see that the class that implements both interfaces must provide implementations of both versions of `eatPlants()` , since they are considered separate methods.

What if the duplicate methods have the same signature but different return types? In that case, you need to review the rules for overriding methods. Let's try an example:

```
interface Dances {
   String swingArms();
}
interface EatsFish {
   CharSequence swingArms();
}

public class Penguin implements Dances, EatsFish {
   public String swingArms() {
      return "swing!";
```

```
    }
}
```

In this example, the `Penguin` class compiles. The `Dances` version of the `swingArms()` method is trivially overridden in the `Penguin` class, as the declaration in `Dances` and `Penguin` have the same method declarations. The `EatsFish` version of `swingArms()` is also overridden as `String` and `CharSequence` are covariant return types.

Let's take a look at a sample where the return types are not covariant:

```
interface Dances {
   int countMoves();
}
interface EatsFish {
   boolean countMoves();
}

public class Penguin implements Dances, EatsFish { // DOES NOT COMPILE
   ...
}
```

Since it is not possible to define a version of `countMoves()` that returns both `int` and `boolean`, there is no implementation of the `Penguin` that will allow this declaration to compile. It is the equivalent of trying to define two methods in the same class with the same signature and different return types.

The compiler would also throw an exception if you define an abstract class or interface that inherits from two conflicting abstract types, as shown here:

```
interface LongEars {
   int softSkin();
}
interface LongNose {
   void softSkin();
}

interface Donkey extends LongEars, LongNose {}  // DOES NOT COMPILE

abstract class Aardvark implements LongEars, LongNose {}
                                                // DOES NOT COMPILE
```

All of the types in this example are abstract, with none being concrete. Despite the fact they are all abstract, the compiler detects that `Donkey` and `Aardvark` contain incompatible methods and prevents them from compiling.

Polymorphism and Interfaces

In Chapter 8, we introduced polymorphism and showed how an object in Java can take on many forms through references. While many of the same rules apply, the fact that a class can inherit multiple interfaces limits some of the checks the compiler can perform.

Abstract Reference Types

When working with abstract types, you may prefer to work with the abstract reference types, rather than the concrete class. This is especially common when defining method parameters. Consider the following implementation:

```
import java.util.*;
public class Zoo {
   public void sortAndPrintZooAnimals(List<String> animals) {
      Collections.sort(animals);
      for(String a : animals) {
         System.out.println(a);
      }
   }
}
```

This class defines a method that 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`, which you have seen before, but it may also be a `LinkedList` or a `Vector` (neither of which you need to know for the exam).

Casting Interfaces

Let's say you have an abstract reference type variable, which has been instantiated by a concrete subclass. If you need access to a method that is only declared in the concrete subclass, then you will need to cast the interface reference to that type, assuming the cast is supported at runtime. That brings us back to a rule we discussed in Chapter 8, namely, that the compiler does not allow casts to unrelated types. For example, the following is not permitted as the compiler detects that the `String` and `Long` class cannot be related:

```
String lion = "Bert";
Long tiger = (Long)lion;
```

With interfaces, there are limitations to what the compiler can validate. For example, does the following program compile?

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

In this program, a `Wolf` object is created and then assigned to a `Canine` reference type on line 7. Because of polymorphism, Java cannot be sure which specific class type the `canine` instance on line 8 is. Therefore, it allows the invalid cast to the `Dog` reference type, even though `Dog` and `Wolf` are not related. The code compiles but 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 does not implement the interface. For example, the following change to line 8 causes the program to fail to compile:

```
8:       Object badDog = (String)canine;  // DOES NOT COMPILE
```

Since `String` does not implement `Canine`, the compiler recognizes that this cast is not possible.

Interfaces and the *instanceof* Operator

In Chapter 3, “Operators,” we showed that the compiler will report an error if you attempt to use the `instanceof` operator with two unrelated classes, as follows:

```
Number tickets = 4;
if(tickets instanceof String) {}  // DOES NOT COMPILE
```

With interfaces, the compiler has limited ability to enforce this rule because even though a reference type may not implement an interface, one of its subclasses could. For example, the following does compile:

```
Number tickets = 5;
if(tickets instanceof List) {}
```

Even though `Number` does not inherit `List`, it’s possible the `tickets` variable may be a reference to a subclass of `Number` that does inherit

`List`. As an example, the `tickets` variable could be assigned to an instance of the following `MyNumber` class (assuming all inherited methods were implemented):

```
public class MyNumber extends Number implements List
```

That said, the compiler can check for unrelated interfaces if the reference is a class that is marked `final`.

```
Integer tickets = 6;
if(tickets instanceof List) {}  // DOES NOT COMPILE
```

The compiler rejects this code because the `Integer` class is marked `final` and does not inherit `List`. Therefore, it is not possible to create a subclass of `Integer` that inherits the `List` interface.

Reviewing Interface Rules

We summarize the interface rules in this part of the chapter in the following list. If you compare the list to our list of rules for an abstract class definition, the first four rules are similar.

1. **Interface Definition Rules** Interfaces cannot be instantiated.
2. All top-level types, including interfaces, cannot be marked `protected` or `private`.
3. Interfaces are assumed to be `abstract` and cannot be marked `final`.
4. Interfaces may include zero or more abstract methods.
5. An interface can extend any number of interfaces.
6. An interface reference may be cast to any reference that inherits the interface, although this may produce an exception at runtime if the classes aren't related.
7. The compiler will only report an unrelated type error for an `instanceof` operation with an interface on the right side if the reference on the left side is a `final` class that does not inherit the interface.

8. An interface method with a body must be marked `default`, `private`, `static`, or `private static` (covered when studying for the 1Z0-816 exam).

The following are the five rules for abstract methods defined in interfaces.

Abstract Interface Method Rules

1. Abstract methods can be defined only in abstract classes or interfaces.
2. Abstract methods cannot be declared `private` or `final`.
3. Abstract methods must not provide a method body/implementation in the abstract class in which is it declared.
4. Implementing an abstract method in a subclass follows the same rules for overriding a method, including covariant return types, exception declarations, etc.
5. Interface methods without a body are assumed to be `abstract` and `public`.

Notice anything? The first four rules for abstract methods, whether they be defined in abstract classes or interfaces, are exactly the same! The only new rule you need to learn for interfaces is the last one.

Finally, there are two rules to remember for interface variables.

Interface Variables Rules

1. Interface variables are assumed to be `public`, `static`, and `final`.
2. Because interface variables are marked `final`, they must be initialized with a value when they are declared.

It may be helpful to think of an interface as a specialized kind of abstract class, since it shares many of the same properties and rules as an abstract class. The primary differences between the two are that interfaces include implicit modifiers, do not contain constructors, do not participate in the instance initialization process, and support multiple inheritance.

Real World Scenario

USING AN INTERFACE VS. IMPLEMENTING AN INTERFACE

An interface provides a way for one individual to develop code that uses another individual's code, without having access to the other individual's underlying implementation. Interfaces can facilitate rapid application development by enabling development teams to create applications in parallel, rather than being directly dependent on each other.

For example, two teams can work together to develop a one-page standard interface at the start of a project. One team then develops code that *uses* the interface, while the other team develops code that *implements* the interface. The development teams can then combine their implementations toward the end of the project, and as long as both teams developed with the same interface, they will be compatible. Of course, testing will still be required to make sure that the class implementing the interface behaves as expected.

Introducing Inner Classes

We conclude this chapter with a brief discussion of inner classes. For the 1Z0-815 exam, you only need to know the basics of inner classes. In particular, you should know the difference between a top-level class and an inner class, permitted access modifiers for an inner class, and how to define a member inner class.

NOTE

For simplicity, we will often refer to inner or nested interfaces as *inner classes*, as the rules described in this chapter for inner classes apply to both `class` and `interface` types.

Defining a Member Inner Class

A *member inner class* is a class defined at the member level of a class (the same level as the methods, instance variables, and constructors). It is the opposite of a top-level class, in that it cannot be declared unless it is inside another class.

Developers often define a member inner class inside another class if the relationship between the two classes is very close. For example, a `Zoo` sells tickets for its patrons; therefore, it may want to manage the lifecycle of the `Ticket` object.

NOTE

For the 1Z0-816 exam, there are four types of nested classes you will need to know about: member inner classes, local classes, anonymous classes, and `static` nested classes. You'll also need to know more detail about member inner classes. For this chapter, we limit our discussion to just the basics of member inner classes, as this is all you need to know on the 1Z0-815 exam.

The following is an example of an outer class `Zoo` with an inner class `Ticket`:

```
public class Zoo {
   public class Ticket {}
}
```

We can expand this to include an interface.

```
public class Zoo {
   private interface Paper {}
```

```
      public class Ticket implements Paper {}
   }
```

While top-level classes and interfaces can only be set with `public` or package-private access, member inner classes do not have the same restriction. A member inner class can be declared with all of the same access modifiers as a class member, such as `public` , `protected` , default (package-private), or `private` .

A member inner class can contain many of the same methods and variables as a top-level class. Some members are disallowed in member inner classes, such as `static` members, although you don't need to know that for the 1Z0-815 exam. Let's update our example with some instance members.

```
   public class Zoo {
      private interface Paper {
         public String getId();
      }
      public class Ticket implements Paper {
         private String serialNumber;
         public String getId() { return serialNumber;}
      }
   }
```

Our `Zoo` and `Ticket` examples are starting to become more interesting. In the next section, we will show you how to use them.

Using a Member Inner Class

One of the ways a member inner class can be used is by calling it in the outer class. Continuing with our previous example, let's define a method in `Zoo` that makes use of the member inner class with a new `sell-Ticket()` method.

```
public class Zoo {
   private interface Paper {
      public String getId();
   }
   public class Ticket implements Paper {
      private String serialNumber;
      public String getId() { return serialNumber; }
   }
   public Ticket sellTicket(String serialNumber) {
      var t = new Ticket();
      t.serialNumber = serialNumber;
      return t;
   }
}
```

The advantage of using a member inner class in this example is that the `Zoo` class completely manages the lifecycle of the `Ticket` class.

Let’s add an entry point to this example.

```
public class Zoo {
   ...
   public static void main(String... unused) {
      var z = new Zoo();
      var t = z.sellTicket("12345");
      System.out.println(t.getId()+" Ticket sold!");
   }
}
```

This compiles and prints `12345 Ticket sold!` at runtime.

For the 1Z0-815 exam, this is the extent of what you need to know about inner classes. As discussed, when you study for the 1Z0-816 exam, there is a lot more you will need to know.

Summary

In this chapter, we presented advanced topics in class design, starting with abstract classes. An abstract class is just like a regular class except that it cannot be instantiated and may contain abstract methods. An abstract class can extend a nonabstract class, and vice versa. Abstract classes can be used to define a framework that other developers write subclasses against.

An abstract method is one that does not include a body when it is declared. An abstract method may be placed inside an abstract class or interface. Next, an abstract method can be overridden with another abstract declaration or a concrete implementation, provided the rules for overriding methods are followed. The first concrete class must implement all of the inherited abstract methods, whether they are inherited from an abstract class or interface.

An interface is a special type of abstract structure that primarily contains abstract methods and constant variables. Interfaces include implicit modifiers, which are modifiers that the compiler will automatically apply to the interface declaration. For the 1Z0-815 exam, you should know which modifiers are assumed in interfaces and be able to spot potential conflicts. When you prepare for the 1Z0-816 exam, you will study the four additional nonabstract methods that interfaces now support. Finally, while the compiler can often prevent casting to unrelated types, it has limited ability to prevent invalid casts when working with interfaces.

We concluded this chapter with a brief presentation of member inner classes. For the exam, you should be able to recognize member inner classes and know which access modifiers are allowed. Member inner classes, along with the other types of nested classes, will be covered in much more detail when you study for the 1Z0-816 exam.

Exam Essentials

Be able to write code that creates and extends abstract classes. In Java, classes and methods can be declared as `abstract`. An abstract class cannot be instantiated. An instance of an abstract class can be obtained only through a concrete subclass. Abstract classes can include any number, including zero, of abstract and nonabstract methods. Abstract methods follow all the method override rules and may be defined only within abstract classes. The first concrete subclass of an abstract class must implement all the inherited methods. Abstract classes and methods may not be marked as `final`.

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 the implicit modifiers that the compiler will automatically apply to an interface. All interfaces are assumed to be `abstract`. An interface method without a body is assumed to be `public` and `abstract`. An interface variable is assumed to be `public`, `static`, and `final` and initialized with a value when it is declared. Using a modifier that conflicts with one of these implicit modifiers will result in a compiler error.

Distinguish between top-level and inner classes/interfaces and know which access modifiers are allowed. A top-level class or interface is one that is not defined within another class declaration, while an inner class or interface is one defined within another class. Inner classes can be marked `public`, `protected`, package-private, or `private`.

Review Questions

The answers to the chapter review questions can be found in the Appendix.

1. What modifiers are implicitly applied to all interface methods that do not declare a body? (Choose all that apply.)

   1. `protected`
   2. `public`
   3. `static`
   4. `void`
   5. `abstract`
   6. `default`
2. Which of the following statements can be inserted in the blank line so that the code will compile successfully? (Choose all that apply.)

```
interface CanHop {}
public class Frog implements CanHop {
   public static void main(String[] args) {
      ___________ frog = new TurtleFrog();
   }
}
class BrazilianHornedFrog extends Frog {}
class TurtleFrog extends Frog {}
```

   1. `Frog`
   2. `TurtleFrog`
   3. `BrazilianHornedFrog`
   4. `CanHop`
   5. `Object`
   6. `Long`
   7. None of the above; the code contains a compilation error.
3. Which of the following is true about a concrete class? (Choose all that apply.)
   1. A concrete class can be declared as `abstract`.
   2. A concrete class must implement all inherited abstract methods.
   3. A concrete class can be marked as `final`.
   4. If a concrete class inherits an interface from one of its superclasses, then it must declare an implementation for all methods defined in that interface.
   5. A concrete method that implements an abstract method must match the method declaration of the abstract method exactly.

4. Which statements about the following program are correct? (Choose all that apply.)

    ```
    1:  interface HasExoskeleton {
    2:     double size = 2.0f;
    3:     abstract int getNumberOfSections();
    4:  }
    5:  abstract class Insect implements HasExoskeleton {
    6:     abstract int getNumberOfLegs();
    7:  }
    8:  public class Beetle extends Insect {
    9:     int getNumberOfLegs() { return 6; }
    10:    int getNumberOfSections(int count) { return 1; }
    11: }
    ```

    1. It compiles without issue.
    2. The code will produce a `ClassCastException` if called at runtime.
    3. The code will not compile because of line 2.
    4. The code will not compile because of line 5.
    5. The code will not compile because of line 8.
    6. The code will not compile because of line 10.
5. What modifiers are implicitly applied to all interface variables? (Choose all that apply.)
    1. `private`
    2. `nonstatic`
    3. `final`
    4. `const`
    5. `abstract`
    6. `public`
    7. default (package-private)
6. Which statements about the following program are correct? (Choose all that apply.)

    ```
    1: public abstract interface Herbivore {
    2:    int amount = 10;
    3:    public void eatGrass();
    ```

```
4:    public abstract int chew() { return 13; }
5: }
6:
7: abstract class IsAPlant extends Herbivore {
8:    Object eatGrass(int season) { return null; }
9: }
```

1. It compiles and runs without issue.
2. The code will not compile because of line 1.
3. The code will not compile because of line 2.
4. The code will not compile because of line 4.
5. The code will not compile because of line 7.
6. The code will not compile because line 8 contains an invalid method override.

7. Which statements about the following program are correct? (Choose all that apply.)

```
1: abstract class Nocturnal {
2:    boolean isBlind();
3: }
4: public class Owl extends Nocturnal {
5:    public boolean isBlind() { return false; }
6:    public static void main(String[] args) {
7:       var nocturnal = (Nocturnal)new Owl();
8:       System.out.println(nocturnal.isBlind());
9: } }
```

1. It compiles and prints `true`.
2. It compiles and prints `false`.
3. The code will not compile because of line 2.
4. The code will not compile because of line 5.
5. The code will not compile because of line 7.
6. The code will not compile because of line 8.
7. None of the above

8. Which statements are true about the following code? (Choose all that apply.)

```
interface Dog extends CanBark, HasVocalCords {
   abstract int chew();
}

public interface CanBark extends HasVocalCords {
   public void bark();
}

interface HasVocalCords {
   public abstract void makeSound();
}
```

   1. The `CanBark` declaration doesn't compile.
   2. A class that implements `HasVocalCords` must override the `makeSound()` method.
   3. A class that implements `CanBark` inherits both the `makeSound()` and `bark()` methods.
   4. A class that implements `Dog` must be marked `final`.
   5. The `Dog` declaration does not compile because an interface cannot extend two interfaces.

9. Which access modifiers can be applied to member inner classes? (Choose all that apply.)
   1. `static`
   2. `public`
   3. default (package-private)
   4. `final`
   5. `protected`
   6. `private`
10. Which statements are true about the following code? (Choose all that apply.)

```
5:  public interface CanFly {
6:     int fly()
7:     String fly(int distance);
8:  }
9:  interface HasWings {
```

```
10:    abstract String fly();
11:    public abstract Object getWingSpan();
12: }
13: abstract class Falcon implements CanFly, HasWings {}
```

1. It compiles without issue.
2. The code will not compile because of line 5.
3. The code will not compile because of line 6.
4. The code will not compile because of line 7.
5. The code will not compile because of line 9.
6. The code will not compile because of line 10.
7. The code will not compile because of line 13.

11. Which modifier pairs can be used together in a method declaration? (Choose all that apply.)
    1. `static and final`
    2. `private and static`
    3. `static and abstract`
    4. `private and abstract`
    5. `abstract and final`
    6. `private and final`
12. Which of the following statements about the `FruitStand` program are correct? (Choose all that apply.)

```
1:  interface Apple {}
2:  interface Orange {}
3:  class Gala implements Apple {}
4:  class Tangerine implements Orange {}
5:  final class Citrus extends Tangerine {}
6:  public class FruitStand {
7:     public static void main(String... farm) {
8:        Gala g = new Gala();
9:        Tangerine t = new Tangerine();
10:       Citrus c = new Citrus();
11:       System.out.print(t instanceof Gala);
12:       System.out.print(c instanceof Tangerine);
13:       System.out.print(g instanceof Apple);
14:       System.out.print(t instanceof Apple);
15:       System.out.print(c instanceof Apple);
16: } }
```

1. Line 11 contains a compiler error.
2. Line 12 contains a compiler error.
3. Line 13 contains a compiler error.
4. Line 14 contains a compiler error.
5. Line 15 contains a compiler error.
6. None of the above

13. What is the output of the following code?

```
1:  interface Jump {
2:     static public int MAX = 3;
3:  }
4:  public abstract class Whale implements Jump {
5:     public abstract void dive();
6:     public static void main(String[] args) {
7:        Whale whale = new Orca();
8:        whale.dive(3);
9:     }
10: }
11: class Orca extends Whale {
12:    public void dive() {
13:       System.out.println("Orca diving");
14:    }
15:    public void dive(int... depth) {
16:       System.out.println("Orca diving deeper "+MAX);
17: } }
```

1. `Orca diving`
2. `Orca diving deeper 3`
3. The code will not compile because of line 2.
4. The code will not compile because of line 4.
5. The code will not compile because of line 11.
6. The code will not compile because of line 16.
7. None of the above

14. Which statements are true for both abstract classes and interfaces? (Choose all that apply.)
    1. Both can be extended using the `extends` keyword.
    2. All methods within them are assumed to be `abstract`.
    3. Both can contain `public static final` variables.
    4. The compiler will insert the implicit `abstract` modifier automatically on methods declared without a body, if they are not marked as such.
    5. Both interfaces and abstract classes can be declared with the `abstract` modifier.
    6. Both inherit `java.lang.Object`.
15. What is the result of the following code?

```
abstract class Bird {
   private final void fly() { System.out.println("Bird"); }
   protected Bird() { System.out.print("Wow-"); }
}
public class Pelican extends Bird {
   public Pelican() { System.out.print("Oh-"); }
   protected void fly() { System.out.println("Pelican"); }
   public static void main(String[] args) {
      var chirp = new Pelican();
      chirp.fly();
} }
```

    1. `Oh-Bird`
    2. `Oh-Pelican`
    3. `Wow-Oh-Bird`
    4. `Wow-Oh-Pelican`
    5. The code contains a compilation error.
    6. None of the above
16. Which of the following statements about this program is correct?

```
interface Aquatic {
   int getNumOfGills(int p);
}
```

```
4: public class ClownFish implements Aquatic {
5:    String getNumOfGills() { return "14"; }
6:    int getNumOfGills(int input) { return 15; }
7:    public static void main(String[] args) {
8:       System.out.println(new ClownFish().getNumOfGills(-1));
9: } }
```

   1. It compiles and prints `14`.
   2. It compiles and prints `15`.
   3. The code will not compile because of line 4.
   4. The code will not compile because of line 5.
   5. The code will not compile because of line 6.
   6. None of the above

17. Which statements about top-level types and member inner classes are correct? (Choose all that apply.)
    1. A member inner class can be marked `final`.
    2. A top-level type can be marked `protected`.
    3. A member inner class cannot be marked `public` since that would make it a top-level class.
    4. A top-level type must be stored in a `.java` file with a name that matches the class name.
    5. If a member inner class is marked `private`, then it can be referenced only in the outer class for which it is defined.
18. What types can be inserted in the blanks on the lines marked `X` and `Z` that allow the code to compile? (Choose all that apply.)

```
interface Walk { public List move(); }
interface Run extends Walk { public ArrayList move(); }
public class Leopard {
   public ______ move() { // X
      return null;
   }
}
public class Panther implements Run {
   public ______ move() { // Z
      return null;
      }
   }
```

1. `Integer` on the line marked `X`
2. `ArrayList` on the line marked `X`
3. `List` on the line marked `Z`
4. `ArrayList` on the line marked `Z`
5. None of the above, since the `Run` interface does not compile.
6. The code does not compile for a different reason.

19. Which statements about interfaces are correct? (Choose all that apply.)
    1. A class cannot extend multiple interfaces.
    2. Java enables true multiple inheritance via interfaces.
    3. Interfaces cannot be declared `abstract`.
    4. If an interface does not contain a constructor, the compiler will insert one automatically.
    5. An interface can extend multiple interfaces.
    6. An interface cannot be instantiated.

20. Which of the following classes and interfaces are correct and compile? (Choose all that apply.)

```
   abstract class Camel {
      void travel();
   }
   interface EatsGrass {
      protected int chew();
   }
   abstract class Elephant {
      abstract private class SleepsAlot {
         abstract int sleep();
      }
   }
   class Eagle {
      abstract soar();
   }
```

1. `SleepsAlot`

2. `Eagle`
3. `Camel`
4. `Elephant`
5. `EatsGrass`
6. None of the classes or interfaces compile.