Chapter 9 - Advanced class design

Creating Abstract Classes

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.

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() {}
}
```

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
}
```

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.

- Java does not permit a class or method to be marked both `abstract` and `final`
- A method cannot be marked as both `abstract` and `private`
- If a `static` method cannot be overridden, then it follows that it also cannot be marked `abstract` since it can never be implemented

Abstract Class Definition Rules

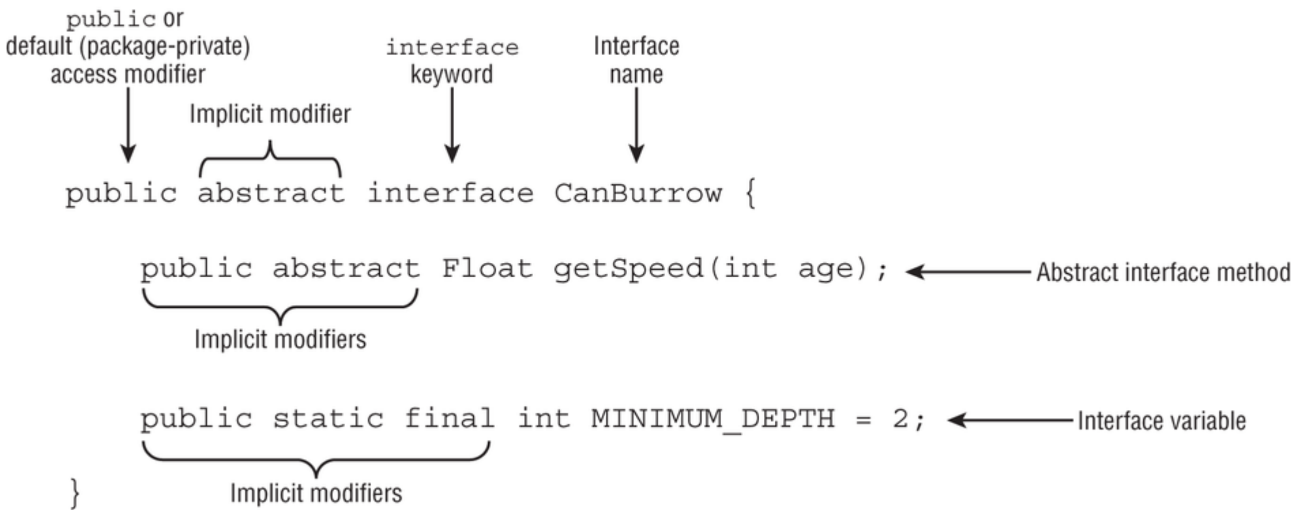
1. Abstract classes cannot be instantiated.
2. All top-level types, including abstract classes, cannot be marked `protected` or `private`.
3. Abstract classes cannot be marked `final`.
4. Abstract classes may include zero or more abstract and nonabstract methods.
5. An abstract class that `extends` another 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.

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.

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
Implicit modifiers
public static final int MINIMUM_DEPTH = 2;
Interface variable
}
Implicit modifiers

- Unlike abstract classes, interfaces 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.
- 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.

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`.

Which line or lines of this top-level interface declaration do not compile?

```java
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!

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.

Duplicate Interface Method Declarations

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

```java
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.

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.

Casting Interfaces

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.

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.

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.

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`.

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