EXERCISE 1-1

Creating an Abstract Superclass and Concrete Subclass

The following exercise will test your knowledge of `public`, default, `final`, and `abstract` classes. Create an `abstract` superclass named `Fruit` and a concrete subclass named `Apple`. The superclass should belong to a package called `food` and the subclass can belong to the default package (meaning it isn't put into a package explicitly). Make the superclass `public` and give the subclass default access.

1. Create the superclass as follows:

```
package food;
public abstract class Fruit{ /* any code you want */}
```

2. Create the subclass in a separate file as follows:

```
import food.Fruit;
class Apple extends Fruit{ /* any code you want */}
```

3. Create a directory called `food` off the directory in your class path setting.
4. Attempt to compile the two files. If you want to use the `Apple` class, make sure you place the `Fruit.class` file in the `food` subdirectory.

CERTIFICATION OBJECTIVE

Use Interfaces (OCA Objective 7.5)

7.6 Use abstract classes and interfaces.

Declaring an Interface

In general, when you create an interface, you're defining a contract for *what* a class can do, without saying anything about *how* the class will do it.

Note: As of Java 8, you can now also describe the *how*, but you usually won't. Until we get to the new interface-related features of Java 8—`default` and `static` methods—we will discuss interfaces from a traditional perspective, which is again, defining a contract for *what* a class can do.

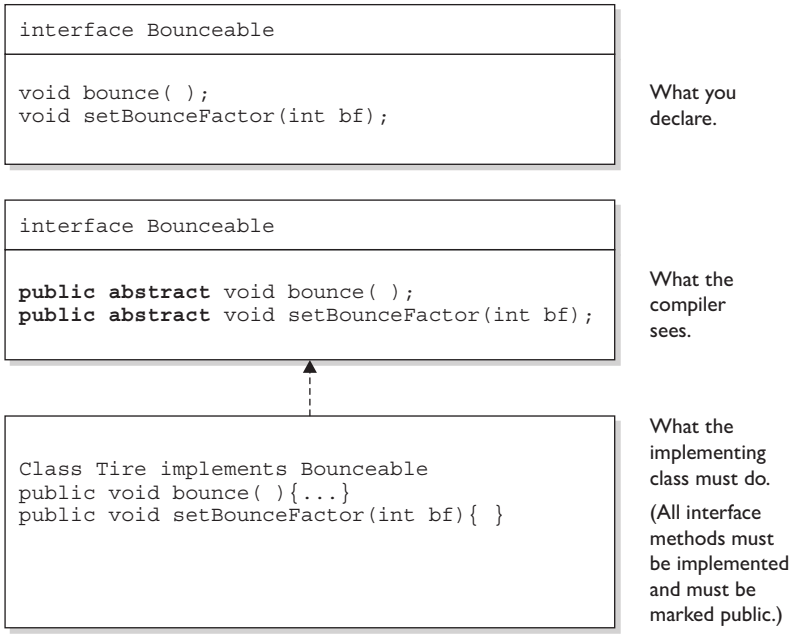
An interface is a contract. You could write an interface `Bounceable`, for example, that says in effect, "This is the `Bounceable` interface. Any concrete class type that implements this interface must agree to write the code for the `bounce()` and `setBounceFactor()` methods."

By defining an interface for `Bounceable`, any class that wants to be treated as a `Bounceable` thing can simply implement the `Bounceable` interface and provide code for the interface's two methods.

Interfaces can be implemented by any class, from any inheritance tree. This lets you take radically different classes and give them a common characteristic. For example, you might want both a `Ball` and a `Tire` to have bounce behavior, but `Ball` and `Tire` don't share any inheritance relationship; `Ball` extends `Toy` while `Tire` extends only `java.lang.Object`. But by making both `Ball` and `Tire` implement `Bounceable`, you're saying that `Ball` and `Tire` can be treated as "Things that can bounce," which in Java translates to, "Things on which you can invoke the `bounce()` and `setBounceFactor()` methods." Figure 1-1 illustrates the relationship between interfaces and classes.

FIGURE 1-1

The relationship between interfaces and classes



Think of a traditional interface as a 100 percent `abstract` class. Like an `abstract` class, an interface defines abstract methods that take the following form:

```
abstract void bounce();  // Ends with a semicolon rather than
                         // curly braces
```

But although an `abstract` class can define both `abstract` and nonabstract methods, an interface *generally* has only `abstract` methods. Another way interfaces differ from `abstract` classes is that interfaces have very little flexibility in how the methods and variables defined in the interface are declared. These rules are strict:

- Interface methods are implicitly `public` and `abstract`, unless declared as `default` or `static`. In other words, you do not need to actually type the `public` or `abstract` modifiers in the method declaration, but the method is still always `public` and `abstract`.
- All variables defined in an interface must be `public`, `static`, and `final`—in other words, interfaces can declare only constants, not instance variables.
- Interface methods cannot be marked `final`, `strictfp`, or `native`. (More on these modifiers later in the chapter.)
- An interface can *extend* one or more other interfaces.
- An interface cannot extend anything but another interface.
- An interface cannot implement another interface or class.
- An interface must be declared with the keyword `interface`.
- Interface types can be used polymorphically (see Chapter 2 for more details).

The following is a legal interface declaration:

```
public abstract interface Rollable { }
```

Typing in the `abstract` modifier is considered redundant; interfaces are implicitly abstract whether you type `abstract` or not. You just need to know that both of these declarations are legal and functionally identical:

```
public abstract interface Rollable { }
public interface Rollable { }
```

The `public` modifier is required if you want the interface to have `public` rather than default access.

We've looked at the interface declaration, but now we'll look closely at the methods within an interface:

```
public interface Bounceable {
    public abstract void bounce();
    public abstract void setBounceFactor(int bf);
}
```

Typing in the `public` and `abstract` modifiers on the methods is redundant, though, since all interface methods are implicitly `public` and `abstract`. Given that rule, you can see that the following code is exactly equivalent to the preceding interface:

```
public interface Bounceable {
      void bounce();                // No modifiers
      void setBounceFactor(int bf); // No modifiers
}
```

You must remember that all interface methods not declared `default` or `static` are `public` and `abstract` regardless of what you see in the interface definition.

Look for interface methods declared with any combination of `public`, `abstract`, or no modifiers. For example, the following five method declarations, if declared within their own interfaces, are legal and identical!

```
void bounce();
public void bounce();
abstract void bounce();
public abstract void bounce();
abstract public void bounce();
```

The following interface method declarations won't compile:

```
final void bounce();        // final and abstract can never be used
                            // together, and abstract is implied
private void bounce();      // interface methods are always public
protected void bounce();    // (same as above)
```

Declaring Interface Constants

You're allowed to put constants in an interface. By doing so, you guarantee that any class implementing the interface will have access to the same constant. By placing the constants right in the interface, any class that implements the interface has direct access to the constants, just as if the class had inherited them.

You need to remember one key rule for interface constants. They must always be

```
public static final
```

So that sounds simple, right? After all, interface constants are no different from any other publicly accessible constants, so they obviously must be declared `public`,

`static`, and `final`. But before you breeze past the rest of this discussion, think about the implications: **Because interface constants are defined in an interface, they don't have to be *declared* as `public`, `static`, or `final`. They must be `public`, `static`, and `final`, but you don't actually have to declare them that way.** Just as interface methods are always `public` and `abstract` whether you say so in the code or not, any variable defined in an interface must be—and implicitly is—a `public` constant. See if you can spot the problem with the following code (assume two separate files):

```
interface Foo {
  int BAR = 42;
  void go();
}

class Zap implements Foo {
  public void go() {
     BAR = 27;
  }
}
```

You can't change the value of a constant! Once the value has been assigned, the value can never be modified. The assignment happens in the interface itself (where the constant is declared), so the implementing class can access it and use it, but as a read-only value. So the `BAR = 27` assignment will not compile.

exam Watch

Look for interface definitions that define constants, but without explicitly using the required modifiers. For example, the following are all identical:

```
public int x = 1;               // Looks non-static and non-final,
                                // but isn't!
int x = 1;                      // Looks default, non-final,
                                // non-static, but isn't!
static int x = 1;               // Doesn't show final or public
final int x = 1;                // Doesn't show static or public
public static int x = 1;        // Doesn't show final
public final int x = 1;         // Doesn't show static
static final int x = 1;         // Doesn't show public
public static final int x = 1;  // what you get implicitly
```

Any combination of the required (but implicit) modifiers is legal, as is using no modifiers at all! On the exam, you can expect to see questions you won't be able to answer correctly unless you know, for example, that an interface variable is `final` and can never be given a value by the implementing (or any other) class.

Declaring `default` Interface Methods

As of Java 8, interfaces can include inheritable* methods with concrete implementations. (*The strict definition of "inheritance" has gotten a little fuzzy with Java 8; we'll talk more about inheritance in Chapter 2.) These concrete methods are called `default` methods. In the next chapter we'll talk a lot about the various OO-related rules that are impacted because of `default` methods. For now we'll just cover the simple declaration rules:

- `default` methods are declared by using the `default` keyword. The `default` keyword can be used only with interface method signatures, not class method signatures.
- `default` methods are `public` by definition, and the `public` modifier is optional.
- `default` methods **cannot** be marked as `private`, `protected`, `static`, `final`, or `abstract`.
- `default` methods must have a concrete method body.

Here are some examples of legal and illegal `default` methods:

```
interface TestDefault {
  default int m1(){return 1;} // legal
  public default void m2(){;} // legal
  static default void m3(){;} // illegal: default cannot be marked static
  default void m4();          // illegal: default must have a method body
}
```

Declaring `static` Interface Methods

As of Java 8, interfaces can include `static` methods with concrete implementations. As with interface `default` methods, there are OO implications that we'll discuss in Chapter 2. For now, we'll focus on the basics of declaring and using `static` interface methods:

- `static` interface methods are declared by using the `static` keyword.
- `static` interface methods are public by default, and the `public` modifier is optional.
- `static` interface methods cannot be marked as `private`, `protected`, `final`, or `abstract`.

- `static` interface methods must have a concrete method body.
- When invoking a `static` interface method, the method's type (interface name) MUST be included in the invocation.

Here are some examples of legal and illegal static interface methods and their use:

```
interface StaticIface {
  static int m1(){ return 42; }      // legal
  public static void m2(){ ; }       // legal
  // final static void m3(){ ; }     // illegal: final not allowed
  // abstract static void m4(){ ; }  // illegal: abstract not allowed
  // static void m5();               // illegal: needs a method body
}

public class TestSIF implements StaticIface {
  public static void main(String[] args) {
    System.out.println(StaticIface.m1());   // legal: m1()'s type
                                            // must be included
    new TestSIF().go();
    // System.out.println(m1());     // illegal: reference to interface
                                     // is  required
  }
  void go() {
    System.out.println(StaticIface.m1());  // also legal from an instance
  }
}
```

which produces this output:

```
42
42
```

As we said earlier, we'll return to our discussion of `default` methods and `static` methods for interfaces in Chapter 2.

CERTIFICATION OBJECTIVE

Declare Class Members (OCA Objectives 2.1, 2.2, 2.3, 4.1, 4.2, 6.2, 6.3, and 6.4)

2.1 Declare and initialize variables (including casting of primitive data types).
2.2 Differentiate between object reference variables and primitive variables.
2.3 Know how to read or write to object fields.