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

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

Working with Enums

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

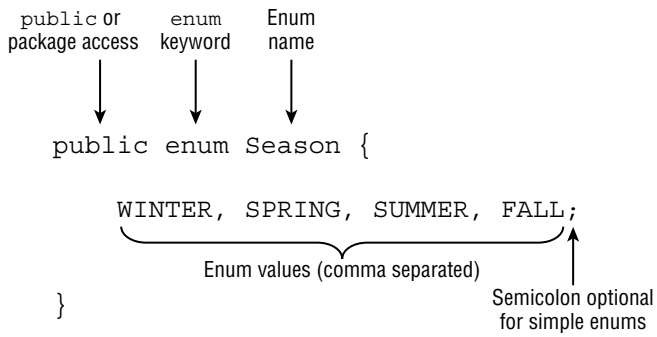
Using an enum is much better than using a bunch of constants because it provides type-safe checking. With numeric or `String` constants, you can pass an invalid value and not find out until runtime. With enums, it is impossible to create an invalid enum value without introducing a compiler error.

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

Creating Simple Enums

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

FIGURE 7.4 Defining a simple enum



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

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

Using an enum is super easy.

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

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

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

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

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

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

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

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

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

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

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

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

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

Calling the *valueOf*() Method

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

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

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

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

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

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

Using Enums in *switch* Statements

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

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

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

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

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

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

Adding Constructors, Fields, and Methods

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

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

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

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

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

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

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

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

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

This class prints the following:

```
begin,constructing,end
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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