Item 21: Use function objects to represent strategies

Some languages support *function pointers, delegates, lambda expressions*, or similar facilities that allow programs to store and transmit the ability to invoke a particular function. Such facilities are typically used to allow the caller of a function to specialize its behavior by passing in a second function. For example, the `qsort` function in C's standard library takes a pointer to a *comparator* function, which `qsort` uses to compare the elements to be sorted. The comparator function takes two parameters, each of which is a pointer to an element. It returns a negative integer if the element indicated by the first parameter is less than the one indicated by the second, zero if the two elements are equal, and a positive integer if the element indicated by the first parameter is greater than the one indicated by the second. Different sort orders can be obtained by passing in different comparator functions. This is an example of the *Strategy* pattern [Gamma95, p. 315]; the comparator function represents a strategy for sorting elements.

Java does not provide function pointers, but object references can be used to achieve a similar effect. Invoking a method on an object typically performs some operation on *that object*. However, it is possible to define an object whose methods perform operations on *other objects*, passed explicitly to the methods. An instance of a class that exports exactly one such method is effectively a pointer to that method. Such instances are known as *function objects*. For example, consider the following class:

```
class StringLengthComparator {
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}
```

This class exports a single method that takes two strings and returns a negative integer if the first string is shorter than the second, zero if the two strings are of equal length, and a positive integer if the first string is longer. This method is a comparator that orders strings based on their length instead of the more typical lexicographic ordering. A reference to a `StringLengthComparator` object serves as a "function pointer" to this comparator, allowing it to be invoked on arbitrary pairs of strings. In other words, a `StringLengthComparator` instance is a *concrete strategy* for string comparison.

As is typical for concrete strategy classes, the `StringLengthComparator` class is *stateless*: it has no fields, hence all instances of the class are functionally

equivalent. Thus it should be a singleton to save on unnecessary object creation costs (Item 3, Item 5):

```
class StringLengthComparator {
    private StringLengthComparator() { }
    public static final StringLengthComparator
        INSTANCE = new StringLengthComparator();
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}
```

To pass a `StringLengthComparator` instance to a method, we need an appropriate type for the parameter. It would do no good to use `StringLengthComparator` because clients would be unable to pass any other comparison strategy. Instead, we need to define a `Comparator` interface and modify `StringLengthComparator` to implement this interface. In other words, we need to define a *strategy interface* to go with the concrete strategy class. Here it is:

```
// Strategy interface
public interface Comparator<T> {
    public int compare(T t1, T t2);
}
```

This definition of the `Comparator` interface happens to come from the `java.util` package, but there's nothing magic about it: you could just as well have written it yourself. The `Comparator` interface is *generic* (Item 26) so that it is applicable to comparators for objects other than strings. Its `compare` method takes two parameters of type `T` (its *formal type parameter*) rather than `String`. The `StringLengthComparator` class shown above can be made to implement `Comparator<String>` merely by declaring it to do so:

```
class StringLengthComparator implements Comparator<String> {
    ... // class body is identical to the one shown above
}
```

Concrete strategy classes are often declared using anonymous classes (Item 22). The following statement sorts an array of strings according to length:

```
Arrays.sort(stringArray, new Comparator<String>() {
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
});
```

But note that using an anonymous class in this way will create a new instance each time the call is executed. If it is to be executed repeatedly, consider storing the function object in a private static final field and reusing it. Another advantage of doing this is that you can give the field a descriptive name for the function object.

Because the strategy interface serves as a type for all of its concrete strategy instances, a concrete strategy class needn't be made public to export a concrete strategy. Instead, a "host class" can export a public static field (or static factory method) whose type is the strategy interface, and the concrete strategy class can be a private nested class of the host. In the example that follows, a static member class is used in preference to an anonymous class to allow the concrete strategy class to implement a second interface, `Serializable`:

```
// Exporting a concrete strategy
class Host {
    private static class StrLenCmp
            implements Comparator<String>, Serializable {
        public int compare(String s1, String s2) {
            return s1.length() - s2.length();
        }
    }

    // Returned comparator is serializable
    public static final Comparator<String>
        STRING_LENGTH_COMPARATOR = new StrLenCmp();

    ...  // Bulk of class omitted
}
```

The `String` class uses this pattern to export a case-independent string comparator via its `CASE_INSENSITIVE_ORDER` field.

To summarize, a primary use of function pointers is to implement the Strategy pattern. To implement this pattern in Java, declare an interface to represent the strategy, and a class that implements this interface for each concrete strategy. When a concrete strategy is used only once, it is typically declared and instantiated as an anonymous class. When a concrete strategy is designed for repeated use, it is generally implemented as a private static member class and exported in a public static final field whose type is the strategy interface.