17.8 Functional Interfaces

[This section requires the interface concepts introduced in Sections 10.9–10.10.]

Section 10.10 introduced Java SE 8's enhanced interface features—`default` methods and `static` methods—and discussed the concept of a *functional interface*—an interface that contains exactly one `abstract` method (and may also contain `default` and `static` methods). Such interfaces are also known as *single abstract method* (SAM) interfaces. Functional interfaces are used extensively in functional-style Java programming. Functional programmers work with so-called *pure functions* that have *referential transparency*—that is, they:

- depend only on their parameters
- have no side-effects and
- do not maintain any state.

In Java, pure functions are methods that implement functional interfaces—typically defined as lambdas, like those you've seen so far in this chapter's examples. State changes occur by passing data from method to method. No data is shared.

 Software Engineering Observation 17.5

Pure functions are safer because they do not modify a program's state (variables). This also makes them less error prone and thus easier to test, modify and debug.

Functional Interfaces in Package `java.util.function`

Package `java.util.function` contains several functional interfaces. Figure 17.10 shows the six basic generic functional interfaces, several of which you've already used in this chapter's examples. Throughout the table, `T` and `R` are generic type names that represent the type of the object on which the functional interface operates and the return type of a method, respectively. Many other functional interfaces in package `java.util.function` are specialized versions of those in Fig. 17.10. Most are for use with `int`, `long` and `double` primitive values. There are also generic customizations of `Consumer`, `Function` and `Predicate` for binary operations—that is, methods that take two arguments. For each `IntStream` method we've shown that receives a lambda, the method's parameter is actually an `int`-specialized version of one of these interfaces.

Interface	Description
`BinaryOperator<T>`	Represents a method that takes two parameters of the same type and returns a value of that type. Performs a task using the parameters (such as a calculation) and returns the result. The lambdas you passed to `IntStream` method `reduce` (Section 17.7) implemented `IntBinaryOperator`—an `int` specific version of `BinaryOperator`.
`Consumer<T>`	Represents a one-parameter method that returns `void`. Performs a task using its parameter, such as outputting the object, invoking a method of the object, etc. The lambda you passed to `IntStream` method `forEach` (Section 17.6) implemented interface `IntConsumer`—an `int`-specialized version of `Consumer`. Later sections present several more examples of `Consumer`s.
`Function<T, R>`	Represents a one-parameter method that performs a task on the parameter and returns a result—possibly of a different type than the parameter. The lambda you passed to `IntStream` method `mapToObj` (Section 17.6) implemented interface `IntFunction`—an `int`-specialized version of `Function`. Later sections present several more examples of `Function`s.
`Predicate<T>`	Represents a one-parameter method that returns a `boolean` result. Determines whether the parameter satisfies a condition. The lambda you passed to `IntStream` method `filter` (Section 17.4) implemented interface `Int-Predicate`—an `int`-specialized version of `Predicate`. Later sections present several more examples of `Predicate`s.
`Supplier<T>`	Represents a no-parameter method that returns a result. Often used to create a collection object in which a stream operation's results are placed. You'll see several examples of `Supplier`s starting in Section 17.13.
	Represents a one-parameter method that returns a result of the same type as its parameter. The lambdas you passed in Section 17.3 to

`UnaryOperator<T>`	`IntStream` method `map` implemented `IntUnaryOperator`—an `int`-specialized version of `Unary-Operator`. Later sections present several more examples of `UnaryOperator`s.

Fig. 17.10

The six basic generic functional interfaces in package `java.util.function`.