17.9 Lambdas: A Deeper Look

Type Inference and a Lambda's Target Type

Lambda expressions can be used anywhere functional interfaces are expected. The Java compiler can usually *infer* the types of a lambda's parameters and the type returned by a lambda from the context in which the lambda is used. This is determined by the lambda's **target type**—the functional-interface type that's expected where the lambda appears in the code. For example, in the call to `IntStream` method `map` from stream pipeline in Fig. 17.4

```
IntStream.rangeClosed(1, 10)
         .map((int x) -> {return x * 2;})
         .sum()
```

the target type is `IntUnaryOperator`, which represents a method that takes one `int` parameter and returns an `int` result. In this case, the lambda parameter's type is explicitly declared to be `int` and the compiler *infers* the lambda's return type as int, because that's what an `IntUnaryOperator`

requires.

The compiler also can *infer* a lambda parameter's type. For example, in the call to `IntStream` method `filter` from stream pipeline in Fig. 17.7

```
IntStream.rangeClosed(1, 10)
         .filter(x -> x % 2 == 0)
         .map(x -> x * 3)
         .sum()
```

the target type is `IntPredicate`, which represents a method that takes one `int` parameter and returns a `boolean` result. In this case, the compiler *infers* the lambda parameter `x`'s type as `int`, because that's what an `IntPredicate` requires. We generally let the compiler *infer* the lambda parameter's type in our examples.

Scope and Lambdas

Unlike methods, lambdas do not have their own scope. So, for example, you cannot shadow an enclosing method's local variables with lambda parameters that have the same names. A compilaton error occurs in this case, because the method's local variables and the lambda parameters are in the *same* scope.

Capturing Lambdas and `final` Local Variables

A lambda that refers to a local variable from the enclosing method (known as the lambda's *lexical scope*) is a **capturing lambda**. For such a lambda, the compiler captures the local variable's value and stores it with the lambda to ensure that the lambda can use the value when the lambda *eventually* executes. This is important, because you can pass a lambda to another method that executes the lambda *after* its lexical scope *no longer exists*.

8

Any local variable that a lambda references in its lexical scope must be `final`. Such a variable either can be explicitly declared `final` or it can be *effectively* `final` (Java SE 8). For an effectively `final` variable, the compiler *infers* that the local variable could have been declared `final`, because its enclosing method never modifies the variable after it's declared and initialized.