***Chapter NINE*  
Lambda Expressions**

**What is a lambda expression?**

* The term lambda expression comes from lambda calculus, written as “λ-calculus”
* λ is the Greek letter lambda. This form of calculus deals with defining and applying functions.

In JAVA

Lambda expressions, are a short-form replacement for anonymous classes. Lambda expressions simplify the use of interfaces that declare a single abstract method, which are also called functional interfaces., a single method interface can be implemented with one of the following options.

A lambda expression can be used to implement a functional interface without creating a class or an anonymous class. Lambda expressions can be used only with interfaces that declare a single method.

**Benefits of lambda expressions**

* Concise syntax
* Method references and constructor references
* Convenient for new streams library
* Reduced runtime overhead compared to anonymous classes

Understanding Lambda Syntax

Figure 9.1

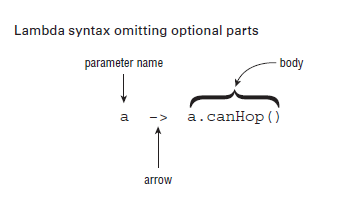
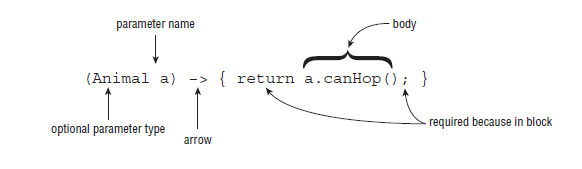


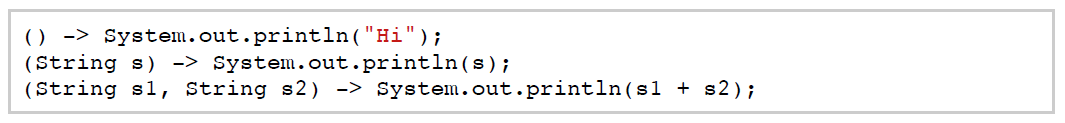
Figure 9.2



A lambda expression has three parts:

**A list of parameters**

A lambda expression can have zero (represented by empty parentheses), one or more parameters:



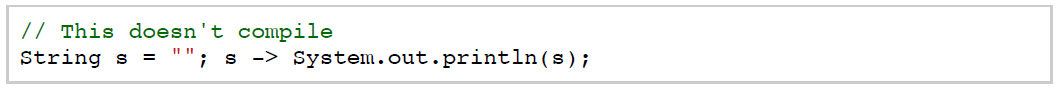
The type of the parameters can be declared explicitly, or it can be inferred from the context:



If there is a single parameter, the type is inferred and it is not mandatory to use parentheses:



If the lambda expression uses a parameter name which is the same as a variable name of the enclosing context, a compile error is generated:



**An arrow**

Formed by the characters - and > to separate the parameters and the body.

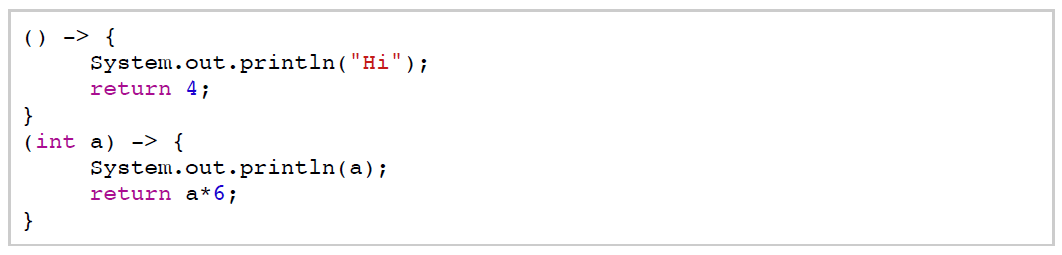
**A body**

The body of the lambda expressions can contain one or more statements.

If the body has one statement, curly brackets are not required and the value of the expression (if any) is returned:



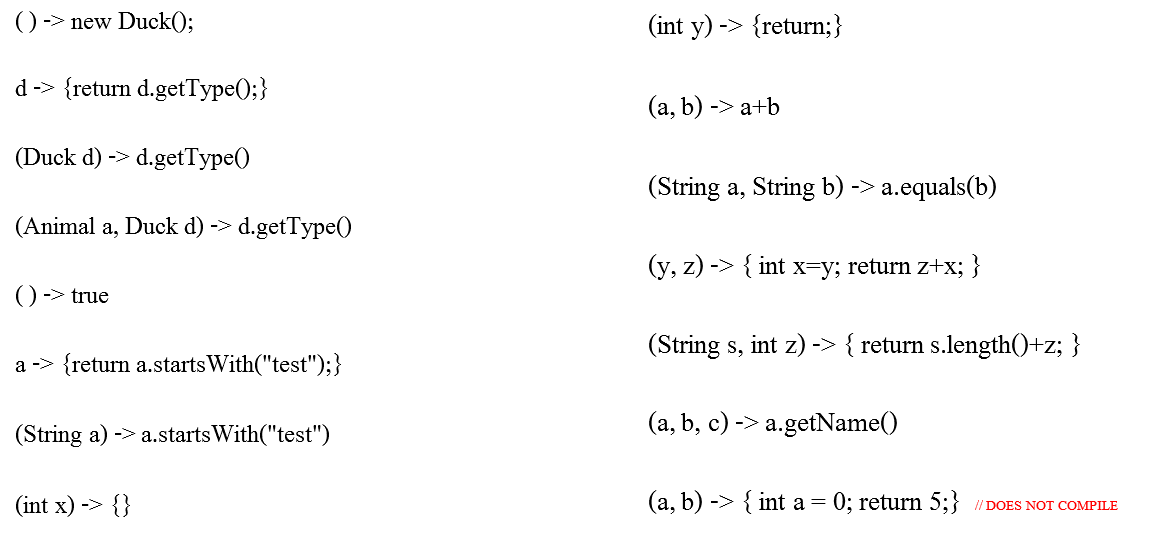
If the body has more than one statement, curly brackets are required, and if the expression returns a value, it must be returned with a return statement:

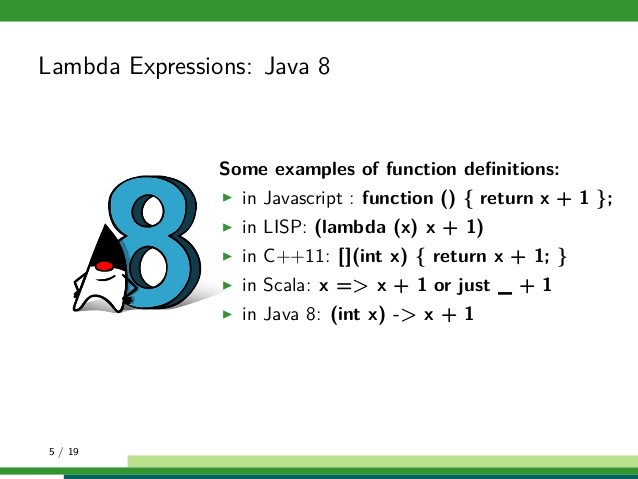


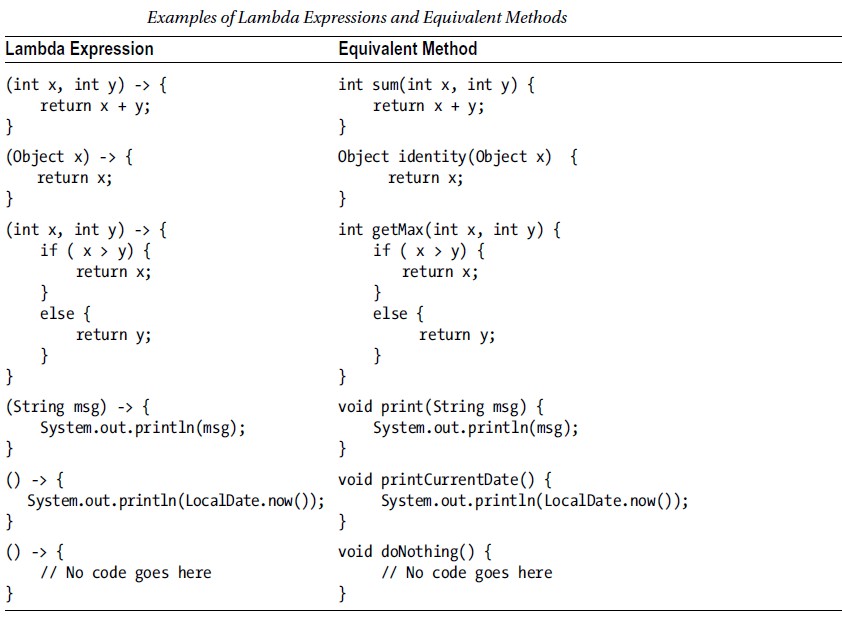
If the lambda expression doesn't return a result, a return statement is optional. For example, the following expressions are equivalent:



For example, the following are all valid lambda expressions, assuming that there are valid functional interfaces that can consume them:







**How are functional interfaces related to all this?**

The signature of the abstract method of a functional interface provides the signature of a lambda expression (this signature is called a ***functional descriptor***).

This means that to use a lambda expression, you first need a functional interface. For example, using the interface of the previous chapter:



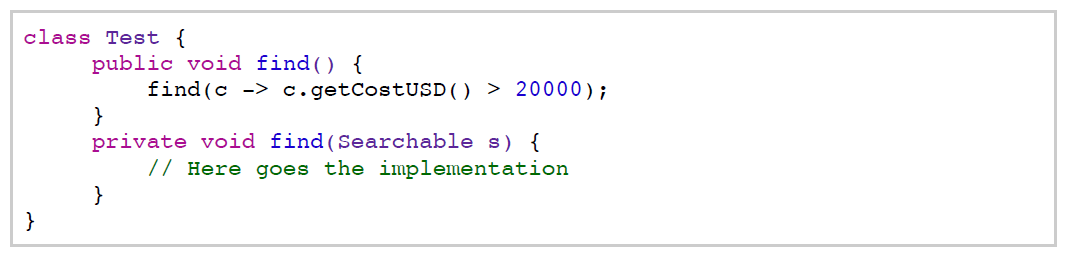
We can create a lambda expression that takes a Car object as argument and returns a boolean:



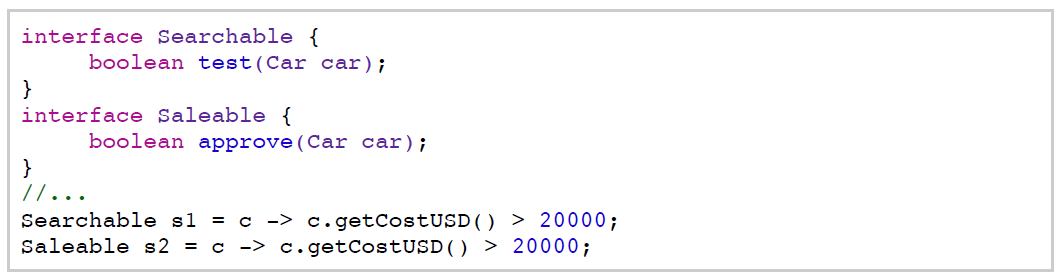
In this case, the compiler inferred that the lambda expression can be assigned to a Searchable interface, just by its signature.

In fact, lambda expressions don't contain information about which functional interface they are implementing. The type of the expression is deduced from the context in which the lambda is used. This type is called ***target type*.**

If we were using the lambda as an argument to a method, the compiler would use the definition of the method to infer the type of the expression:

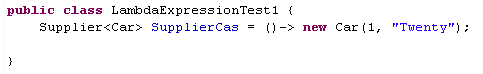


Because of this, the same lambda expression can be associated with different functional interfaces if they have a compatible abstract method signature. For example:



For reference, the contexts where the target type (the functional interface) of a lambda expression can be inferred are:

* A variable declaration



* An assignment



* A return statement

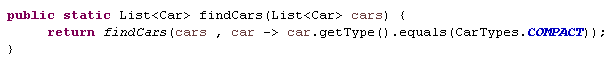


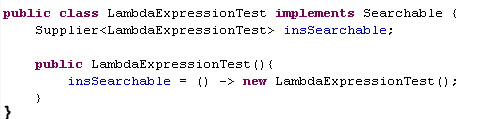
* An array initializer





* Method or constructor arguments



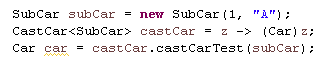


* A ternary conditional expression



* A cast expression

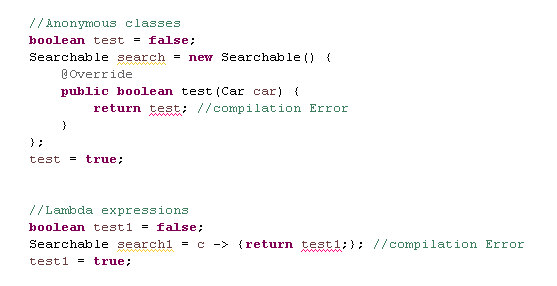




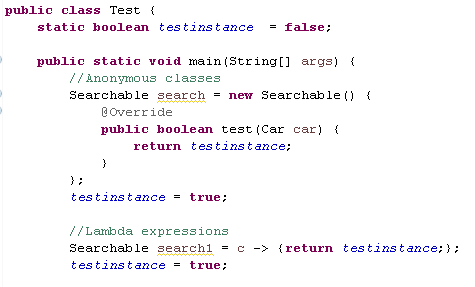
\*\*\*However, if you understand the concept, you don't need to memorize this list.

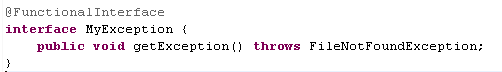
**Lambda expressions VS Anonymous classes**

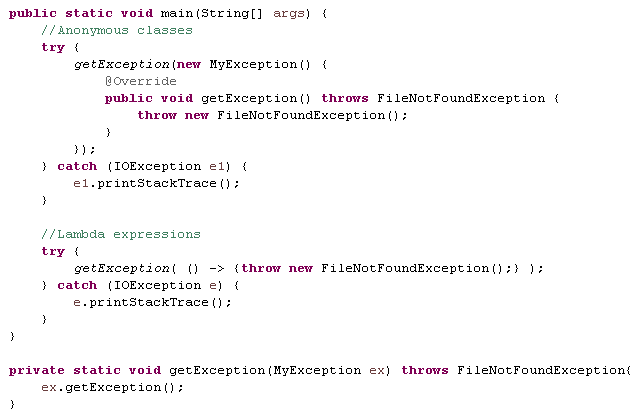
* **They have some similarities:**
  + Local variables (variables or parameters defined in a method) can only be used if they are declared final or are effectively final.



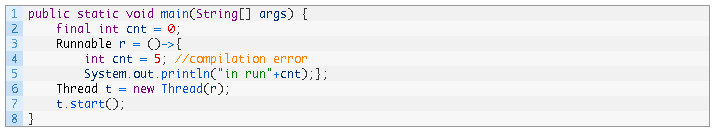
* + You can access instance or static variables of the enclosing class.



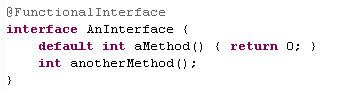
* + They must not throw more exceptions than specified in the throws clause of the functional interface method. Only the same type or a supertype.

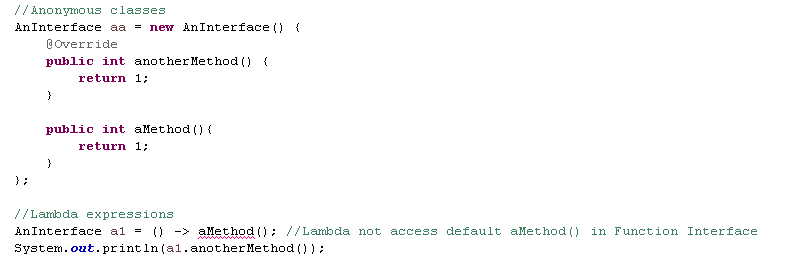


* **And some significant differences:**
  +  Scoping



* + Default methods of a functional interface cannot be accessed from within lambda expressions. Anonymous classes can.





* + Performance

At runtime anonymous inner classes require class loading, memory allocation and object initialization and invocation of a non-static method while lambda expression is pure compile time activity and don’t incur extra cost during runtime. So performance of lambda expression is better as compare to anonymous inner classes.

**Key Points**

* Lambda expressions have three parts: a list of parameters, and arrow, and a body:  
  
* You can think of lambda expressions as anonymous methods (or functions) as they don't have a name.
* A lambda expression can have zero (represented by empty parentheses), one or more parameters.
* The type of the parameters can be declared explicitly, or it can be inferred from the context.
* If there is a single parameter, the type is inferred and is not mandatory to use parentheses.
* If the lambda expression uses as a parameter name which is the same as a variable name of the enclosing context, a compile error is generated.
* If the body has one statement, curly brackets are not required, and the value of the expression (if any) is returned.
* If the body has more than one statement, curly brackets are required, and if the expression returns a value, it must return with a return statement.
* If the lambda expression doesn't return a result, a return statement is optional.
* The signature of the abstract method of a functional interface provides the signature of a lambda expression (this signature is called a *functional descriptor*).
* This means that to use a lambda expression, you first need a functional interface.
* The type of the expression is deduced from the context in which the lambda is used. This type is called *target type*.
* The contexts where the target type of a lambda expression can be inferred include an assignment, method or constructor arguments, and a cast expression.
* Like anonymous classes, lambda expressions can access instance and static variables, but only final or effectively final local variables.
* Also, they cannot throw exceptions that are not defined in the throws clause of the function interface method.
* Default methods of a functional interface cannot be accessed from within lambda expressions.