**Anonymous Inner Classes**

The rule of Abstraction asserts that the type of a variable should be an interface when possible. In such a case the name of the class that implements the interface is relatively unimportant, as it will only be used when the class constructor is called. This section examines how to create unnamed classes, called anonymous inner classes, and the convenience that they provide.

An anonymous inner class defines a class without giving it a name.

Suppose that T is an interface. The general syntax is:

T v = new T() { ... };

This statement causes the compiler to do three things:

* It creates a new class that implements T and has the code appearing within the braces.
* It creates a new object of that class by calling the class’s default constructor.
* It saves a reference to that object in variable v.

Note that the client will never need to know the class of the new object,

because it interacts with the object only via the variable of type T.

The code for the method innerClassComp in ComparatorBankAccounts

appears in Listing 1. The bold code highlights the anonymous inner

class syntax. The code within the braces implements the compare method.

*Listing 1. The innerClassComp Method*

private static Comparator<BankAccount> innerClassComp() {

Comparator<BankAccount> result =

**new Comparator<BankAccount>() {**

**public int compare(BankAccount ba1,**

**BankAccount ba2) {**

**int bal1 = ba1.getBalance();**

**int bal2 = ba2.getBalance();**

**if (bal1 == bal2)**

**return ba1.getAcctNum() - ba2.getAcctNum();**

**else**

**return bal1 - bal2;**

**}**

};

return result;

}

**Lambda Expressions**

An anonymous inner class provides a convenient way to define a class and

create a single instance of it, as both the class and its instance can be created inline. This section shows how it is often possible to shorten the definitions of anonymous inner classes, making them even more convenient.

An interface is said to be ***functional*** if it has only one method, not

counting any default or static methods. The interface Comparator<T>

is an example of a functional interface. An anonymous inner class for a

functional interface can be written very compactly. Since there is only

one method to define, its name and return type are determined by the

interface, so you don’t need to write them; you only need to write the code for the method. This notation is called a lambda expression. Its syntax is:

(T1 t1, ..., Tn tn) -> {...}

The method’s parameter list is to the left of the “arrow” and its code is

to its right, within braces. The method lambdaExpComp1 in Comparator

BankAccounts uses this syntax; see the bold portion of Listing 2. Its

compare method compares accounts by their account numbers, from high

to low.

*Listing 2. The lambdaExpComp1 Method*

private static Comparator<BankAccount> lambdaExpComp1() {

Comparator<BankAccount> result =

**(BankAccount ba1, BankAccount ba2) -> {**

**return ba2.getAcctNum() - ba1.getAcctNum();**

**};**

return result;

}

Although lambda expressions can be written reasonably compactly,

Java lets you abbreviate them even further.

* You don’t have to specify the types of the parameters.
* If there is only one parameter then you can omit the parentheses around it.
* If the body of the method consists of a single statement then you can omit the braces; if a single-statement method also returns something then you also omit the “return” keyword.

private static Comparator<BankAccount> lambdaExpComp2() {

Comparator<BankAccount> result =

**(ba1, ba2) -> ba1.getAcctNum() - ba2.getAcctNum();**

return result;

}