JAVA 101: LEARN JAVA
By Jeff Friesen, JavaWorld
NOV 7, 2019 10:46 AM PST

ADVANCED JAVA LANGUAGE FEATURES

Get started with lambda expressions in Java

Learn how to use lambda expressions and functional programming techniques in your Java programs

Listing 10. LambdaDemo.java (version 10)

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Predicate;
class Account
{
   private int id, balance;
   Account(int id, int balance)
   {
      this.balance = balance;
      this.id = id;
   }
   void deposit(int amount)
   {
      balance += amount;
   }
   int getBalance()
   {
      return balance;
   }
   int getID()
   {
      return id;
   }
   void print()
   {
      System.out.printf("Account: [%d], Balance: [%d]%n", id, balance);
   }
}
public class LambdaDemo
{
   static List<Account> accounts;
   public static void main(String[] args)
   {
      accounts = new ArrayList<>();
      accounts.add(new Account(1000, 200));
      accounts.add(new Account(2000, -500));
      accounts.add(new Account(3000, 0));
      accounts.add(new Account(4000, -80));
      accounts.add(new Account(5000, 1000));
      // Deposit enough money in accounts with negative balances so that they
      // end up with zero balances (and are no longer overdrawn).
      adjustAccounts(account -> account.getBalance() < 0,
                     account -> account.deposit(-account.getBalance()));
   }
   static void adjustAccounts(Predicate<Account> tester,
                              Consumer<Account> adjuster)
```

```
   {
      for (Account account: accounts)
      {
         if (tester.test(account))
         {
            adjuster.accept(account);
            account.print();
         }
      }
   }
}
```

Listing 10 continues on from the previous example by introducing an `adjustAccounts()` method that addresses overdrawn accounts by depositing enough money to give them zero balances. `adjustAccounts()` takes two lambda arguments, which must conform to `Predicate<T>`'s and `Consumer<T>`'s abstract method parameter lists and return types.

The compiler determines that the lambda arguments passed to `adjustAccounts()` are correct. The `test()` method is implemented to take an `Account account` parameter and execute `return account.getBalance() < 0;`. Similarly, `accept()` is implemented to take the same parameter and execute `account.deposit(-account.getBalance());`.

Compile Listing 10 and run the application. You should observe the following output:

```
Account: [2000], Balance: [0]
Account: [4000], Balance: [0]
```

Primitive specializations of predefined functional interfaces

`java.util.function` includes primitive specializations of various functional interfaces. For example, `DoubleConsumer` is a primitive specialization of `Consumer`. Each primitive specialization functional interface exists for performance reasons, to avoid unnecessary object creation and method calls when the inputs or outputs are primitive type-based values.

In conclusion

In this tutorial I've introduced you to programming with lambda expressions. I started with a high-level overview, then offered in-depth introductions to the core features and techniques associated with lambdas: target types, scopes, local variables, the `this` and `super` keywords,

and exceptions.

While lambdas have done much to simplify and modernize Java programming, in some cases their usage still results in unnecessary clutter. The next tutorial in the Java 101 series will introduce method references, which you can combine with lambda expressions to write even more concise, readable Java code.

Jeff Friesen teaches Java technology (including Android) to everyone.

Follow