JAVAWORLD JAVA

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

The first example demonstrates a lambda in a variable declaration context. It assigns lambda `() -> { System.out.println("running"); }` to variable `r` of `Runnable` interface type. The second example is similar, but demonstrates a lambda in an assignment context (to previously declared variable `r`).

The third example demonstrates a lambda in a return statement context. It invokes the `getFilter()` method with a specified file extension argument to return a `java.io.FileFilter` object. This object is passed to `java.io.File`'s `listFiles()` method, which invokes the filter for each file, ignoring files that don't match the extension.

The `getFilter()` method returns a `FileFilter` object expressed via a lambda. The compiler notes that the lambda satisfies this functional interface's `boolean accept(File pathname)` method (both have a single parameter and the lambda body returns a Boolean value) and binds the lambda to `FileFilter`.

The fourth example demonstrates lambda usage in an array initializer context. Two `java.nio.file.PathMatcher` objects are created based on lambdas. Each `PathMatcher` object matches files based on criteria specified by its lambda's body. Here is the relevant code:

```
final PathMatcher matchers[] =
{
  (path) -> path.toString().endsWith("txt"),
  (path) -> path.toString().endsWith("java")
};
```

The `PathMatcher` functional interface provides a `boolean matches(Path path)` method that agrees with the lambda's parameter list and its body's Boolean return type. This method is subsequently called to determine a match (based on file extension) for each encountered file during a visit of the current directory and subdirectories.

The fifth example demonstrates a lambda in a `Thread` constructor context. The sixth example demonstrates a lambda in a lambda context, which shows that lambdas can be nested. The seventh example demonstrates a lambda in a ternary conditional expression (`?:`) context: one of two lambdas is selected based on an ascending or descending sort.

The eighth (and final) example demonstrates a lambda in a cast expression context. The `() -> System.getProperty("user.name")` lambda is cast to `PrivilegedAction<String>` functional interface type. This cast addresses an ambiguity in the `java.security.AccessController` class, which declares the following methods:

```
static <T> T doPrivileged(PrivilegedAction<T> action)
static <T> T doPrivileged(PrivilegedExceptionAction<T> action)
```

The problem is that each of interfaces `PrivilegedAction` and `PrivilegedExceptionAction` declares an identical `T run()` method. Because the compiler cannot figure out which interface is the target type, it reports an error in the absence of the cast.

Compile Listing 4 and run the application. You should observe the following output, which assumes that `LambdaDemo.java` is the only `.java` file in the current directory and that this directory contains no `.txt` files:

```
running
running
Found matched file: '.\LambdaDemo.java'.
running
called
Washington
Sydney
Rome
Ottawa
Moscow
London
Jerusalem
Berlin
jeffrey
```

Lambdas and scopes

The term *scope* refers to that part of a program where a name is bound to a particular entity (e.g., a variable). In another part of the program, the name may be bound to another entity. A lambda body doesn't introduce a new scope. Instead, its scope is the enclosing scope.

Lambdas and local variables

A lambda body can define local variables. Because these variables are considered part of the enclosing scope, the compiler will report an error when it detects that the lambda body is redefining a local variable. Listing 5 demonstrates this problem.

Listing 5. LambdaDemo.java (version 5)

```
public class LambdaDemo
{
   public static void main(String[] args)
   {
      int limit = 10;
      Runnable r = () -> {
                           int limit = 5;
                           for (int i = 0; i < limit; i++)
                              System.out.println(i);
                         };
   }
}
```

Because `limit` is already present in the enclosing scope (the `main()` method), the lambda body's redefinition of `limit` (`int limit = 5;`) causes the compiler to report the following error message: `error: variable limit is already defined in method main(String[])`.

Lambda bodies and local variables
Whether originating in a lambda body or in the enclosing scope, a local variable must be initialized before being used. Otherwise, the compiler will report an error.

A local variable or parameter that's defined outside a lambda body and referenced from the body must be marked `final` or considered *effectively final* (the variable cannot be assigned to after initialization). Attempting to modify an effectively final variable causes the compiler to report an error, as demonstrated in Listing 6.

Listing 6. LambdaDemo.java (version 6)

```
public class LambdaDemo
{
   public static void main(String[] args)
   {
      int limit = 10;
      Runnable r = () -> {
                           limit = 5;
                           for (int i = 0; i < limit; i++)
                              System.out.println(i);
                         };
   }
}
```

`limit` is effectively final. The lambda body's attempt to modify this variable causes the compiler to report an error. It does so because a final/effectively final variable will need to hang around until the lambda executes, which may not happen until long after the code in which the variable was defined returns. Non-final/non-effectively final variables no longer exist.

Lambdas and the 'this' and 'super' keywords

Any this or super reference that is used in a lambda body is regarded as being equivalent to its usage in the enclosing scope (because a lambda doesn't introduce a new scope). However, this isn't the case with anonymous classes, which Listing 7 demonstrates.

Listing 7. LambdaDemo.java (version 7)

```
public class LambdaDemo
{
   public static void main(String[] args)
   {
      new LambdaDemo().doWork();
   }
   public void doWork()
   {
      System.out.printf("this = %s%n", this);
      Runnable r = new Runnable()
                       {
                          @Override
                          public void run()
                          {
                             System.out.printf("this = %s%n", this);
                          }
                       };
      new Thread(r).start();
      new Thread(() -> System.out.printf("this = %s%n", this)).start();
   }
}
```

Listing 7's main() method instantiates LambdaDemo and invokes the object's doWork() method to output the object's this reference, instantiate an anonymous class that implements Runnable, create a Thread object that executes this runnable when its thread is started, and create another Thread object whose thread executes a lambda when started.

Compile Listing 7 and run the application. You should observe something similar to the following output:

```
this = LambdaDemo@776ec8df
this = LambdaDemo$1@48766bb
this = LambdaDemo@776ec8df
```

The first line shows LambdaDemo's this reference, the second line shows a different this reference in the new Runnable scope, and the third output line shows the this reference in a lambda context. The third and first lines match because the lambda's scope is nested inside the doWork() method; this has the same meaning throughout this method.

Lambdas and exceptions

A lambda body is not allowed to throw more exceptions than are specified in the throws clause of the functional interface method. If a lambda body throws an exception, the functional interface method's throws clause must declare the same exception type or its supertype. Consider Listing 8.

Listing 8. LambdaDemo.java (version 8)

```
import java.awt.AWTException;
import java.io.IOException;
@FunctionalInterface
interface Work
{
   void doSomething() throws IOException;
}
public class LambdaDemo
{
   public static void main(String[] args) throws AWTException, IOException
   {
      Work work = () -> { throw new IOException(); };
      work.doSomething();
      work = () -> { throw new AWTException(""); };
   }
}
```

Listing 8 declares a Work functional interface whose doSomething() method is declared to throw java.io.IOException. The main() method assigns a lambda that throws IOException to work, which is okay because IOException is listed in doSomething()'s throws clause.

main() next assigns a lambda that throws java.awt.AWTException to work. However, the compiler doesn't allow this assignment because AWTException isn't part of doSomething()'s throws clause (and is certainly not a subtype of IOException).

Predefined functional interfaces

You might find yourself repeatedly creating similar functional interfaces. For example, you might create a `CheckConnection` functional interface with a `boolean isConnected(Connection c)` method and a `CheckAccount` functional interface with a `boolean isPositiveBalance(Account acct)` method. This is wasteful.

The previous examples expose the abstract concept of a *predicate* (a Boolean-valued function). Anticipating such patterns, Oracle provides the `java.util.function` package of commonly-used functional interfaces. For example, this package's `Predicate<T>` functional interface can be used in place of `CheckConnection` and `CheckAccount`.

`Predicate<T>` provides a `boolean test(T t)` method that evaluates this predicate on its argument (`t`), returning true when `t` matches the predicate, and returning false otherwise. Notice that `test()` provides the same kind of parameter list as `isConnected()` and `isPositiveBalance()`. Also, notice that they all have the same return type (`boolean`).

The application source code in Listing 9 demonstrates `Predicate<T>`.

Listing 9. LambdaDemo.java (version 9)

```java
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;
class Account
{
   private int id, balance;
   Account(int id, int balance)
   {
      this.balance = balance;
      this.id = id;
   }
   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));
      // Print all accounts
      printAccounts(account -> true);
      System.out.println();
      // Print all accounts with negative balances.
      printAccounts(account -> account.getBalance() < 0);
      System.out.println();
      // Print all accounts whose id is greater than 2000 and less than 5000.
      printAccounts(account -> account.getID() > 2000 &&
                         account.getID() < 5000);
   }
   static void printAccounts(Predicate<Account> tester)
   {
```

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

Listing 9 creates an array-based list of accounts with positive, zero, and negative balances. It then demonstrates `Predicate<T>` by invoking `printAccounts()` with lambdas for printing out all accounts, only those accounts with negative balances, and only those accounts whose IDs are greater than 2000 and less than 5000.

Consider lambda expression `account -> true`. The compiler verifies that the lambda matches `Predicate<T>`'s `boolean test(T)` method, which it does--the lambda presents a single parameter (`account`) and its body always returns a Boolean value (`true`). For this lambda, `test()` is implemented to execute `return true;`.

Compile Listing 9 and run the application. You should observe the following output:

```
Account: [1000], Balance: [200]
Account: [2000], Balance: [-500]
Account: [3000], Balance: [0]
Account: [4000], Balance: [-80]
Account: [5000], Balance: [1000]
Account: [2000], Balance: [-500]
Account: [4000], Balance: [-80]
Account: [3000], Balance: [0]
Account: [4000], Balance: [-80]
```

`Predicate<T>` is just one of `java.util.function`'s various predefined functional interfaces. Another example is `Consumer<T>`, which represents an operation that accepts a single argument and returns no result. Unlike `Predicate<T>`, `Consumer<T>` is expected to operate via side-effects. In other words, it modifies its argument in some way.

`Consumer<T>`'s `void accept(T t)` method executes an operation on its argument (`t`). When appearing in the context of this functional interface, a lambda must conform to the `accept()` method's solitary parameter and return type. Listing 10 presents an example that demonstrates `Consumer<T>` along with `Predicate<T>`.