Lambda Expressions

A *lambda expression* is an unnamed function, together with its scope (called a *closure*). To use lambdas in Java, you need at least Java version 8. The syntax of a lambda expression is:

```
(Datatype variable[,...]) -> { statement block; }
```

for example, a lambda that prints its argument to System.out is:

```
(Object x) -> { System.out.println(x); }
```

in cases where there is only one variable and the data type can be inferred from context, you can omit the data type and parenthesis:

```
x -> { System.out.println( x );  }
```

if the lambda definition is only a single statement you can omit the brackets { } and semi-colon, too.

```
x -> System.out.println( x )
```

Use of Lambda Expressions

The most common use is to define an implementation of an interface that has only one required method. These are called *functional interfaces.* The examples below illustrate this use.

Examples

In a Swing GUI, we can add an ActionListener to a button using an anonymous class like this:

```
button.addActionListener( new ActionListener( ) {
      @Override
      public void actionPerformed( ActionEvent evt ) {
            System.out.println( evt.getSource() + " pressed" );
      }
      } );
```

Usiing a lambda expression we could write this as:

```
button.addActionListener(
      ( ActionEvent evt ) -> {
            System.out.println( evt.getSource() + " pressed" ); } );
```

Java knows from context that an ActionListener is required as argument to addActionListener( ) and it knows that ActionListener has only one method, with a parameter of type ActionEvent. Hence, the meaning of the Lambda is clear from context. Thus, we can further simplify this to:

```
button.addActionListener(
       evt -> System.out.println( evt.getSource() + " pressed" ) );
```

Suppose we want a Comparator to preform a *case insensitive* sorting of an array of Strings.

```
String [] array = { "Jack fruit", "durian", "Apple", "fig", "banana"};
Comparator<String> comp = new Comparator<String>( ) {
     public int compare(String a, String b) {
         return a.compareToIgnoreCase(b);
     }
};
Arrays.sort( array, comp );
```

Using a lambda expression, with 2 parameters, we would write this as:

```
Comparator<String> comp = (a,b) -> a.compareToIgnoreCase(b) ;

Arrays.sort( array, comp );
```

Or we could define the comparator inline:

```
Arrays.sort( array, (a,b) -> a.compareToIgnoreCase(b) );
```

The Java compiler knows that the second argument to Arrays.sort must be a Comparaotr, and can infer that the type must be String.

Lambdas for Functions without Arguments

To write a lambda expression for a method without parameters, use () for the lambda params, as down in method declarations. For example, the Runnable interface has a single method run( ). To write a lambda as Runnable:

```
Runnable task = () -> System.out.println("running") ;
```

Method References

Sometimes the body of a Lambda expression simply passes parameters to another method.

Java defines *method references* of the form: "`Classname::methodName`" for static methods, and "`objectReference::methodName`" for instance methods (note the double colon ::).

As a simple example, Java has a java.util.functionConsumer interface with a single method accept that returns void. Its called a consumer because it "consumes" a value and doesn't return anything. We could define a consumer to print its argument:

```
Consumer print = (x) -> System.out.println(x);
print.accept("Hello nerd");
```

This lambda just passes the parameter (x) to another function, so we could rewrite it as a method reference:

```
Consumer print = System.out::println;
print.accept("Goodbye, nerd");
```

In the case insensitive sorting example above:

```
Comparator<String> ignoreCase = (a,b) -> a.compareToIgnoreCase(b);
Arrays.sort( array, ignoreCase );
```

can also be written as a method reference, using:

```
Arrays.sort( array, String::compareToIgnoreCase );
```

From context, Java knows that the second parameter to `sort()` must be a Comparator<String>, and Comparator<String> has a single required method with 2 String parameters. Since string.compareToIgnoreCase is not static, Java uses the first parameter (a) as the object reference, and the second parameter (b) as the parameter to compareToIgnoreCase, so it becomes: a.compareToIgnoreCase(b).

Lambda as Commands

Suppose we have a list of Students with a name, id, and birthday.

Student
name: String id: String birthday: LocalDate

We want to print all the students born this month (so we can send them a birthday greeting.

A simple code for this is:

```
public void filterAndPrint( List<Student> students, int month ) {
      for(Student s : students ) {
            if (s.getBirthday().getMonthValue() == month)
System.out.println( s );
      }
}
```

Code improvement:

1) define a Filter interface with a single method boolean test(Student s).

2) use anonymous class to define Filter for birthday.

3) use lambda instead of anonymous class.

4) don't need Filter: java.util.function.Predicate does the same thing.

5) add a Consumer to generalize "print".

References

- Oracle's *Java Tutorial* has a section on Lambda expressions. It also has a section on the new properties of interfaces in Java 8.1