

Functional Interfaces and Lambda Expressions

6



ORACLE



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Adolfo De-la-Rosa (adolfoelarosaz@gmail.com) has a non-transferable license to use this Student Guide.

Objectives

After completing this lesson, you should be able to do the following:

- Explain functional programming concepts
- Define a functional interface
- Define a predicate
- Describe how to pass a function as a parameter
- Define a lambda expression
- Define a statement lambda
- Describe lambda parameters
- Describe local-variable syntax for lambda parameters



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Problem Statement

- Given a list of `Person` objects, perform operations like making an automated call to selected `Persons` by filtering the `List` based on selection criteria like age or gender.
- The `Person` class has the following properties:

```
public class Person {  
    private String givenName;  
    private String surName;  
    private int age;  
    private Gender ;  
    private String eMail;  
    private String phone;  
    private String address;  
    private String city;  
    private String state;  
    private String code;  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The `Person` class also includes getters and setters for each field and code that implements the Builder pattern. The code in the slide only shows the fields.

RoboCall Class

- The RoboCall class has the following properties:

```
public class RoboCall {  
  
    public static robocall(String number){  
  
        /*Code to place an automated call.  
        This code will connect to the phone system  
        using the supplied number and place the call.  
        */  
    }  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

RoboCall is a class that has a static method that can place an automated phone call to a supplied phone number. The program that uses this class needs to filter the list of `Person` objects based on the selection criteria and then pass the filtered `Persons`' phone number to the `RoboCall.robocall()` method which will place the automated call.

RoboCall Every Person

```
void robocallEveryPerson() {  
    List<Person> list=gatherPersons();  
    for(Person p:list){  
        String num=p.getPhoneNumber();  
        RoboCall.robocall(num); }}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

RoboCall Use Case: Eligible Drivers

```
void robocallEligibleDrivers() {  
    List<Person> list=gatherPersons();  
    for(Person p:list){  
        if(p.getAge() >= 16) {  
            String num=p.getPhoneNumber();  
            RoboCall.robocall(num);    }    }}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

For every Person in the list, you retrieve the phone number by invoking the `getPhoneNumber` method and calling `RoboCall.robocall()`.

RoboCall Use Case: Eligible Voters

```
void robocallEligibleVoters() {  
    List<Person> list=gatherPersons();  
    for(Person p:list){  
        if(p.getAge() >= 18) {  
            String num=p.getPhoneNumber();  
            RoboCall.robocall(num);    } }}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Call only persons over 18.

RoboCall Use Case: Legal Drinking Age

```
void robocallLegalDrinkers() {  
    List<Person> list=gatherPersons();  
    for(Person p:list){  
        if(p.getAge() >= 21) {  
            String num=p.getPhoneNumber();  
            RoboCall.robocall(num);    }    }}
```

- Problem: Notice the code in these three use cases; there is duplicated code, i.e. same boiler plate code with some tweaks inside!



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Call only persons over 21.

Solution: Parameterization of Values

Add a parameter, age.

```
public static void robocallPersonOlderThan(int age) {  
    for(Person p: pl) {  
        if(p.getAge() >= age) {  
            String num=p.getPhoneNumber();  
            RoboCall.robocall(num); } } }
```

Age is
parameterized
and the
constant value
for age is
replaced.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

age is passed as a parameter instead of a constant; you are passing a value.

Solution: Parameterized Methods

Add a parameter, age.

```
void robocallEligibleDrivers() {  
    robocallPersonOlderThan(16);  
  
    void robocallEligibleVoters() {  
        robocallPersonOlderThan(18);  
  
        void robocallLegalDrinkers() {  
            robocallPersonOlderThan(21);  
        }  
    }  
}
```

Removes the
duplication seen
earlier.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

This is a rewrite of the previous three methods; each of these methods call a single method, `robocallPersonOlderThan()`, and pass different values for age.

Parameters for Age Range

Additional Use Case: To enroll into the National Service Program, the age range is 18 to 25 (inclusive).

```
public static void robocallPersonsInAgeRange(int low, int high) {  
    for (Person p : pl) {  
        if (low <= p.getAge() && high < p.getAge()) {  
            String num = p.getPhoneNumber();  
            RoboCall.robocall(num);    }  
        }  
    }  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The code has been made more flexible by now looking at an age range.

Using Parameters for Age Range

```
void robocallEligibleDrivers() {  
    robocallPersonsInAgeRange (16, MAX);  
}  
  
void robocallEligibleVoters() {  
    robocallPersonsInAgeRange (18, MAX);  
}  
  
void robocallLegalDrinkers () {  
    robocallPersonsInAgeRange (21, MAX);  
}  
  
void robocallSelectiveService() {  
    robocallPersonsInAgeRange (18, 26);  
}
```

INTEGER.MAX_VALUE



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

This is a good example of *delegation*. A method is called and that method determines what other method to call and with what parameters. The caller is only aware of the methods on this page.

Corrected Use Case

To enroll into the National Service Program, age range is 18 to 25 (inclusive) and only **men** can enroll!

```
enum GENDER{MALE,FEMALE}

void robocallSelectiveService (){
    robocallPersonInRange(18,25,MALE);
}
```

This works, but what about queries that don't care about gender?



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

As the selection criteria become more complex and varied, it becomes more cumbersome to implement this with separate methods.

To retrieve all the eligible drivers or voters that don't care about gender of a person, you can either:

Add `DON'T_CARE` as a value to `enum GENDER`; this mixes data with values, or you can pass `NULL` as values to the `enum`. But it's wrong to pass `NULL` as value to the `gender` of an object `Person`, so it becomes more complex to handle all the use cases.

Parameterized Computation

- Value parameterization is good, but only up to a point.
- Problems:
 - Values must be in the given range
 - Sometimes some special values work: 0, -1, null, INTEGER.MAX_VALUE
 - Add more boolean or enum parameters
- Paradigm Shift: Parameterize a method's **behavior** instead of its values:
 - Make a method's parameter a **function** instead of a value



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Based on all the use cases, it's clear parameterizing values has drawbacks. Instead, parameterizing behavior, that is, passing behavior as a function to a method, may be a superior approach. This will be explored next.

How To Pass a Function in Java?

Ideally pass the function - the actual behavior itself - `p.getAge() >= someValue`
Instead of parametrizing values, you parametrize the behavior.

```
/* For each Person in the list, perform a test: are they within an age
range? Are they Male or Female? Are they in a certain city. And there could
be many, currently unidentified, tests needed in the future as well.
If they pass the test, get their phone number and Robocall their number. */

// How would it look to pass the function as an argument?
void robocallEligibleDrivers(){
    robocallPersonOlderThan(<a function to test if a person is 16 or over>;)

void robocallEligibleVoters(){
    robocallPersonOlderThan(<a function to test if a person is 18 or over>;)}}

// How can this be done in Java?
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

A better solution is to use the Functional Programming paradigm to handle situations where you find yourself adding more methods to cover more and more cases.

Prior to Java SE 8: Pass a Function Wrapped in an Object

```
public class DriverEligibilityTester {
    public boolean isEligible(Person p) {
        return p.getAge() >= 16;    }}

// Pass isEligible method wrapped within an object
DriverEligibilityTester eTester = new DriverEligibilityTester();
robocallEligible(eTester);

public static void robocallEligible(DriverEligibilityTester tester) {
    for (Person p : pl) {
        if (tester.isEligible(p)) { // call the wrapped method
            String num=p.getPhoneNumber();
            RoboCall.robocall(num);    }}}

// Notice you can use a more generic method to make the calls. By passing
// the test condition as an argument instead of passing literal values,
// the same method can be used for many different eligibility selections.
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Prior to SE 8, methods were not “first class citizens” and could not be referenced or passed as parameters. Therefore, you need to “wrap” the method in a object in order to pass it.

While this approach works, and has been available since JDK 1.2, it is verbose and requires a lot of extra code to make it work.

Prior to SE 8: Abstract Behavior With an Interface

While using a concrete class will work, it is more flexible and a better design to use an interface:

```
public interface EligiblePerson {
    boolean isEligible(Person p);    }

public class DriverEligibilityTester implements EligiblePerson{
    @Override
    public boolean isEligible(Person p) {
        return p.getAge() >= 16;    }}

// main()...
EligiblePerson eTester = new DriverEligibilityTester();
robocallEligible(eTester);

public static void robocallEligible(EligiblePerson tester) {
    for (Person p : pl) {
        if (tester.isEligible(p)) {
            String num=p.getPhoneNumber();
            RoboCall.robocall(num);    }    } }
```

roboCallEligible() can accept any class that implements the EligiblePerson interface



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Substituting an interface as the formal parameter allows greater flexibility in what types you can pass.

Prior to SE 8: Replace Implementation Class with Anonymous Inner Class

```
//old way
//EligiblePerson eTester = new EligibilityTester();
//robocallEligible(eTester);

// New way:
robocallEligible(new EligiblePerson() {
    public boolean isEligible(Person p) {
        return p.getAge() >= 16;    }    });

public static void robocallEligible(EligiblePerson tester) {
    for (Person p : pl) {
        if (tester.isEligible(p)) {
            String num=p.getPhoneNumber();
            RoboCall.robocall(num);    }    }
```

Note what's different. The object being passed is now instantiated within the method call. It is an implementation of EligiblePerson but has no class name. It is an anonymous inner class.

There is a lot of “boilerplate” code – is it all needed? In SE 8 and later you can replace anonymous inner classes with a Lambda expression.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The interface and concrete class can be replaced with an anonymous inner class. While this is an improvement over coding both the functional interface and concrete class, it still inserts verbose, boilerplate code. The anonymous inner class compiles into separate classes that increase the size of the code base for the application, as well.

Lambda Solution: Replace Anonymous Inner Class with Lambda Expression

Java SE 8 introduced the Lambda syntax. A functional interface can now be expressed more simply and clearly.

This:

```
robocallEligibile( new EligiblePerson () {  
    public boolean isEligible(Person p) {  
        return p.getAge() >= 16; } })
```

Can be replaced with this syntax:

```
robocallEligibile( (Person p) -> p.getAge() >= 16 );
```

```
public static void robocallEligibile(EligiblePerson tester) {  
    for (Person p : pl) {  
        if (tester.isEligible(p)) {...
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

If the intent is to pass a method as a parameter, before SE 8 you would have needed to create an interface and the use an anonymous inner class to create the object to be passed at runtime. This anonymous class can be replaced with a lambda expression in SE 8.

The Java SE 8 JDK (compiler and runtime) understands and accepts a lambda expression for an anonymous inner class whose interface has one abstract method.

It's now easy to use a lambda expression to create much more complex filters for the person to be robocalled. What if you needed to call everyone who is 18-25 and lives in Denver?

```
(Person p) p-> getAge() >=18 && p.getAge() <=25 && p.getCity.equals("Denver");
```

As long as the return type is Boolean and takes a Person as an argument, any expression will work:

```
(Person p) p-> p.getSurName().length() > 10 characters // works
```

Rewriting the Use Cases Using Lambda

```
void robocallEligibleDrivers() {  
    robocallEligible(p -> p.getAge() >= 16);  
}  
  
void robocallEligibleVoters() {  
    robocallEligible(p -> p.getAge() >= 18);  
}  
  
void robocallLegalDrinkers () {  
    robocallEligible(p -> p.getAge() >= 21);  
}  
  
void robocallEligibleForService() {  
    robocallEligible(p -> p.getGender() == MALE &&  
                    p.getAge() >= 18 &&  
                    p.getAge() <= 25);  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The method that calls those eligible to be robocalled is now generic and uses a lambda expression for the test of eligibility.

The specific methods encapsulate the “rules” for that type of eligible driver. Example: age >= 16 or even something more complex.

What Is a Lambda?

- Essentially an anonymous function:
 - Allows one to treat data as a function
 - Provides parameterization of **behavior** as opposed to **values** or **type**
- Combined with JVM and class library changes, it supports:
 - Explicit but unobtrusive parallelism
 - High productivity
- Suitable for simple everyday programming and also heavy-lifting (parallel)



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Adolfo De-la-Rosa (adolfoelarosa2012@gmail.com) + Rosa
non-transferable license to use this Student Guide

What is a Functional Interface

- A functional interface in Java is one that has only **one** method.
- For example:
 - Some widely used functional interfaces
 - `java.util.Comparator` has only `compare()` method
 - `java.awt.event.ActionListener` has only `actionPerformed(ActionEvent)` method
- `Arrays.sort()` will take an array of objects and sort them, providing the object implements the `Comparable` interface – has a method that compares elements to determine order.
- But, what if the object array **does not** implement `Comparable`?
 - Another `Arrays.sort()` method takes the object array and the sorting behavior as a parameter (a `Comparator`):

```
Arrays.sort( (p1,p2) -> p1.getSurName().compareTo(p2.getSurName()) )
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Sorting collections is interesting to examine.

The default approach is that the object being sorted must implement `Comparable`. `Comparable` tells the sort method what the sorting behavior should be. This is quite an elegant approach, but doesn't handle objects that haven't implemented `Comparable`, and perhaps cannot be refactored to do so.

But there's another `sort()` method that can sort objects that *don't* implement `Comparable`. If the object doesn't implement `Comparable`, how can the sorting behavior be established? The answer is by passing in this *behavior* as a parameter.

This is done with the `Comparator` function. It's interesting that this bit of functional style programming has been part of Java since 1.2! It's just so useful that even though you have to use an anonymous class or concrete class to express the functionality in the `Comparator`, it's been the way to do it.

Then lambda really just makes this passing of functionality easy enough that it becomes useful, readable, and elegant for all kinds of designs. Streams would probably be pretty much unusable without lambda - certainly they'd lose their "fluent" description.

`Arrays.sort()` and `Collections.sort()` are part of the API, but any design situation where you find yourself adding more methods to cover more and more cases is a good example of where passing functionality is useful. Adding new methods is ugly and error-prone; at the very least adding a new method means changing more than one class. Using the functional approach instead is much better, the method doesn't need to change, only the caller of the method.

Which of These Interfaces Are Functional Interfaces?

```
public interface DoAddition{
    int add(int a, int b);
}

public interface SmartSubtraction{
    int subtract(int a, int b);
    int subtract(double a, double b);
}

public interface SmartAddition extends DoAddition{
    int add(double a, double b);
}

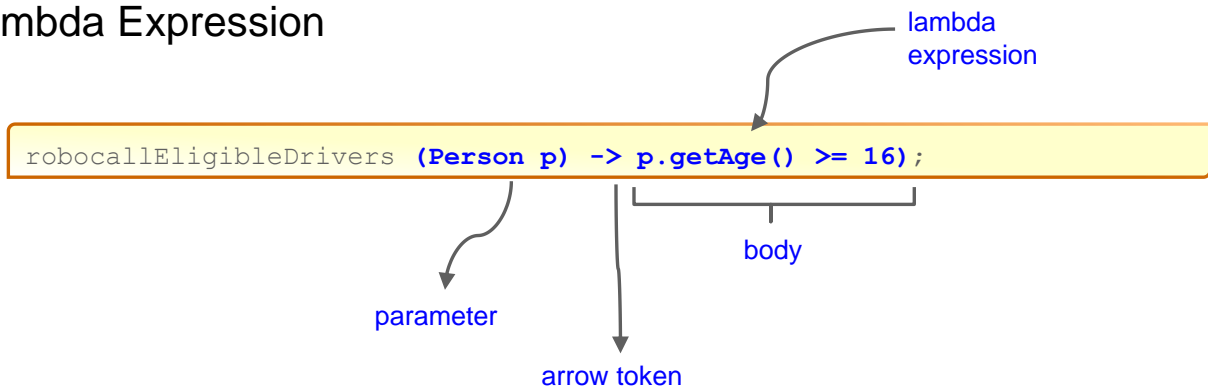
public interface NothingToSeeHere{
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The second is not a functional interface, as there are two abstract methods. The third is not as it adds a second method. The fourth is not because it has no methods.

Lambda Expression



- A lambda expression is like a method; it provides a list of formal parameters and a body.
- A lambda body is either a single expression or a block, expressed in terms of those parameters.
 - Like a method body, a lambda body describes code that will be executed whenever an invocation occurs.

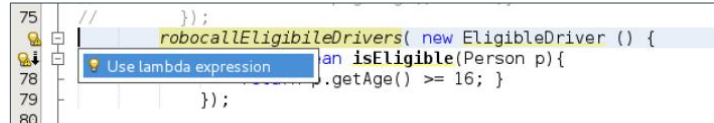


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The syntax of the lambda is based on the functional notation discussed earlier. When you write a lambda expression in Java SE 8 or later, it's converted into an instance of a functional interface.

Using Lambdas

- Lambdas work just like an interface or class and can be used anywhere you would use a functional interface.
- This syntax change can be detected by IDEs like NetBeans, and they can suggest it or an anonymous class if you prefer.

A screenshot of a code editor showing a Java method call. The code is:

```
robocallEligibleDrivers( new EligibleDriver () {  
    an isEligible(Person p){  
        .getAge() >= 16; }  
});
```

 A blue tooltip box with a lightbulb icon and the text "Use lambda expression" is overlaid on the anonymous class definition. The line numbers 75, 76, 77, 78, 79, and 80 are visible on the left margin.

- Using Lambda expressions you can simplify your code and improve readability.
- They can even make frameworks and advanced and sophisticated functionality like streams feasible.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Methods ("functions") can now be passed as parameters and called, referenced and passed by object references and "stored" within an object reference. How? The method (lambda expression) is referenced like an object.

Which of The Following Are Valid Lambda Expressions?

1. `() -> {}`
2. `() -> "Duke"`
3. `() -> {return "Kenny";}`
4. `(Integer i) -> return "Larry" + i;`
5. `(String s) -> {"IronMan";}`

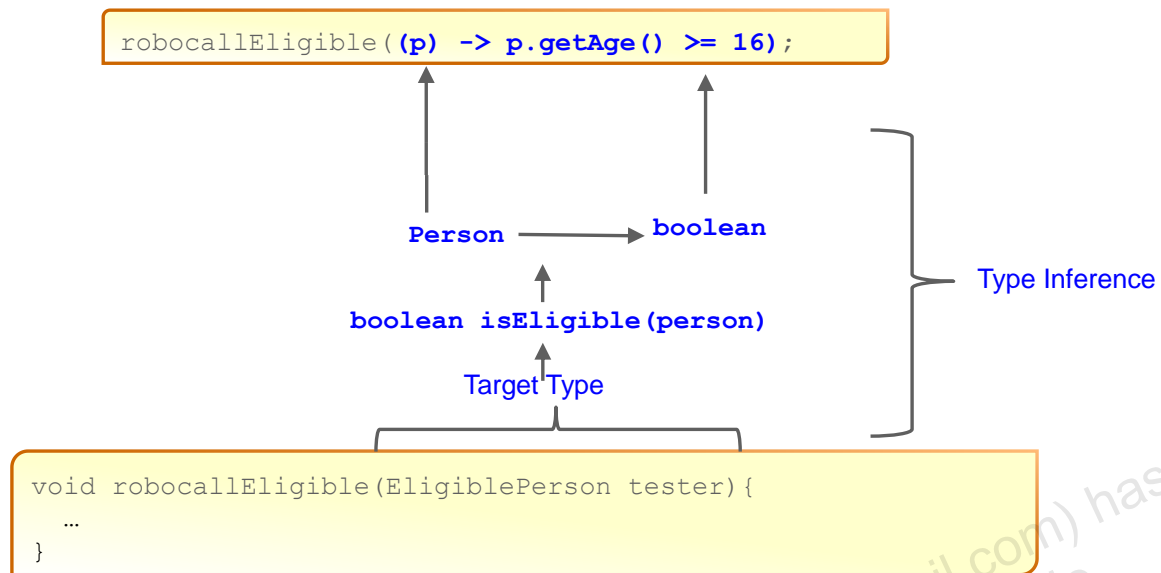


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Answer: Only 4 and 5 are invalid lambdas.

1. This lambda has no parameters and returns void. It's similar to a method with an empty body. For example:
`public void run() { }.`
2. This lambda has no parameters and returns a `String` as an expression.
3. This lambda has no parameters and returns a `String` (using an explicit return statement).
4. `return` is a control-flow statement. To make this lambda valid, curly braces are required as follows:
`(Integer i) -> {return "Larry" + i;}.`
5. `"IronMan"` is an expression, not a statement. To make this lambda valid, you can remove the curly braces and semicolon as follows:
`(String s) -> "IronMan".`
Or if you prefer, you can use an explicit return statement as follows:
`(String s) -> {return "IronMan";}.`

Lambda Expression: Type Inference



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

What is the type of p?

The compiler performs the type inference as illustrated in the slide to determine the type of p. The compiler verifies that in the lambda expression the parameter to the left of the arrow: p is of the type and the expression to the right of the arrow returns a boolean value. You don't have to explicitly specify the type.

To Create a Lambda Expression

It's essentially an anonymous inner class without the boilerplate code.

Include the essentials: parameter list, block of code to execute and match the return type of the method.

```
// pass as a parameter to a method:
robocallEligible((p, age) -> p.getAge() >= 16);

// store in variable and pass as a variable to the method as a parameter.
EligiblePerson eTest = (p) -> p.getAge() >= 16;
robocallEligible(eTest);

// store in an array
EligiblePerson[] eTests = {(Person p) -> p.getAge() >= 16,
                           (Person p) -> p.getAge() >= 18};
robocallEligible(eTests[0]);
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Getting started creating lambdas can be tricky. One technique that can help is to get your code working with an anonymous inner class first, then replace the boilerplate code with the lambda.

Examples of Lambdas

A boolean expression	<code>(List<String> list) -> list.isEmpty() // Predicate</code>
Creating objects	<code>() -> new Person(10) // Supplier</code>
Consuming from an object	<code>(Person p) -> {System.out.println(p.getSurName());} // Consumer</code>
Select/extract from an object	<code>(String s) -> s.length()</code>
Combine two values	<code>(int a, int b) -> a * b //Function</code>
Compare two objects	<code>(Person p1, Person p2) -> p1.getAge().compareTo(p2.getAge())</code>



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Lambdas really just make this passing of functionality easy enough that it becomes useful, readable, and elegant for all kinds of designs.

Statement Lambdas: Lambda with Body

```
robocallMatchingPersons (p -> p.getAge() >=16,  
                        num -> {System.out.println("Calling");  
                               robocall(num); });  
  
robocallMatchingPersons (p -> p.getAge() >=18,  
                        num -> {System.out.println("Calling");  
                               txtMsg(num); });
```

These are called **statement** lambdas instead of **expression** lambdas as they contain a block; it's similar to executing an ordinary block of code in Java within braces.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Examples: Lambda Expression

```
() -> {} // No parameters; result is void  
  
() -> 42 // No parameters, expression body  
  
(int x) -> x+1 // Single declared-type parameter  
  
(x) -> x+1 // Single inferred-type parameter  
  
(int x, int y) -> x+y // Multiple declared-type parameters  
  
(x, y) -> x+y // Multiple inferred-type parameters
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Lambda Parameters

- The formal parameters of a lambda expression may have either declared types or inferred types.
 - It's not possible to declare the types of some of its parameters but leave others to be inferred.
 - Only parameters with declared types can have modifiers.

```
(x, int y) -> x+y      // Illegal: can't mix inferred and declared types  
(x, final y) -> x+y   // Illegal: no modifiers with inferred types
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Local-Variable Syntax for Lambda Parameters

- This is a new feature introduced in Java SE 11.
- Allows `var` keyword to be used when declaring the formal parameters of implicitly typed lambda expressions.
- A lambda expression may be implicitly typed, where the types of all its formal parameters are inferred:
`(x, y) -> x.process(y) // implicitly typed lambda expression`
- Java SE 10 makes implicit typing available for local variables:

```
var x = new Foo();  
for (var x : xs) { ... }  
try (var x = ...) { ... } catch ...
```

- For uniformity with local variables, in Java SE 11 you can add `var` for the formal parameters of an implicitly typed lambda expression:
`(var x, var y) -> x.process(y) // implicit typed lambda expression`



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Local-Variable Syntax for Lambda Parameters

- An implicitly typed lambda expression must use `var` for all its formal parameters or for none of them.
 - In addition, `var` is permitted only for the formal parameters of implicitly typed lambda expressions.
 - Explicitly typed lambda expressions continue to specify manifest types for all their formal parameters.
 - The following examples are illegal:

```
(var x, y) -> x.process(y)    // Cannot mix 'var' and 'no var' in implicitly typed lambda  
                             expression  
  
(var x, int y) -> x.process(y) // Cannot mix 'var' and manifest types in explicitly typed  
                             lambda expression
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

JEP 323: Local-Variable Syntax for Lambda Parameters:

<http://openjdk.java.net/jeps/323>

JEP 286: Local-Variable Type Inference

<http://openjdk.java.net/jeps/286>

Functional Interfaces: Predicate

You don't have to create a new interface each time you wish to use a lambda expression.

```
public interface Predicate<T> {  
    public boolean test(T t);  
  
    robocallEligible( (Person p) -> p.getAge() >= 16 );  
  
    public static void robocallEligible(Predicate pred) {  
        for (Person p : pl) {  
            if (pred.test(p))  
                RoboCall.robocall(p.getPhoneNumber()); } }  
}
```

The method signature of the lambda expression matches the interface, so it compiles.

- Lambda method signature: boolean methodName (Person p)
- Required method signature: boolean methodName (Type t)



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Functional interfaces are covered in detail in the lesson "Built-in Functional Interfaces".

Using Functional Interfaces

```
// DriverEligible interface is not needed!

robocallEligibile((Person p) -> p.getAge() >= 16); // Drivers
robocallEligibile((Person p) -> p.getAge() >= 18); //Voters
robocallEligibile((Person p) -> p.getCity() >= "Denver"); // Residents
robocallEligibile((Person p) -> p.getAge()>=18 && p.getAge()<=25); //age range

public static void robocallEligibile(Predicate pred) {
    for (Person p : pl) {
        if (pred.test(p)) {
            RoboCall.roboCall(p.getPhoneNumber()); } } }
// compiler needs a method signature that matches our lambda (and vice versa)
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Using existing functional interfaces makes your code more robust and predictable. The method calls will always be passing an existing type. Also as you'll see in the lesson "Built-in Functional Interfaces", functional interfaces often have useful functionality expressed in their default methods.

Quiz

Q

When a developer creates an anonymous inner class, the new class is typically based on which one of the following?

- A. enums
- B. Executors
- C. Functional interfaces
- D. Static variables



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Quiz

Q

Which is true about the parameters passed into this lambda expression:

`(t, s) -> t.contains(s)`

- A. Their type is inferred from the context.
- B. Their type is executed.
- C. Their type must be explicitly defined.
- D. Their type is undetermined.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Summary

In this lesson, you should have learned how to:

- Explain functional programming concepts
- Define a functional interface
- Define a predicate
- Describe how to pass a function as a parameter
- Define a lambda expression
- Define a statement lambda
- Describe lambda parameters
- Describe local-variable syntax for lambda parameters

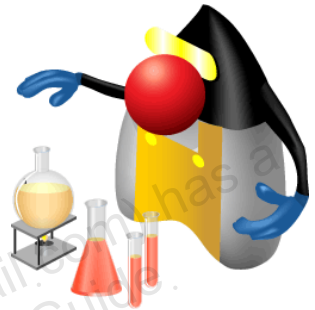


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Practice 6: Overview

This practice covers creating lambda expressions.

- Practice 6-1: Refactor Code to Use Lambda Expressions
- Practice 6-2: Refactor Code to Reuse Lambda Expressions



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a
non-transferable license to use this Student Guide.