**Lambdas and Predicates**

A functional interface is any interface that contains only one abstract method. (A functional interface may contain one or more default methods or static methods.) Because a functional interface contains only one abstract method, you can omit the name of that method when you implement it.

interface Predicate<T> {

boolean test(T t);

}

A lambda expression consists of the following

A comma-separated list of formal parameters enclosed in parentheses. The CheckPerson.test method contains one parameter, p, which represents an instance of the Person class.

interface CheckPerson {

boolean test(Person p);

}

printPersonsWithPredicate(

roster,

p -> p.getGender() == Person.Sex.MALE

&& p.getAge() >= 18

&& p.getAge() <= 25

);

public static void printPersonsWithPredicate(

List<Person> roster, CheckPerson tester) {

for (Person p: roster) {

if (tester.test(p)) {

// Do some Action

}

}

}

Note: You can omit the data type of the parameters in a lambda expression. In addition, you can omit the parentheses if there is only one parameter. For example, the following lambda expression is also valid:

The arrow token, ->

A body, which consists of a single expression or a statement block. This example uses the following expression:

p.getGender() == Person.Sex.MALE

&& p.getAge() >= 18

&& p.getAge() <= 25

If you specify a single expression, then the Java runtime evaluates the expression and then returns its value. Alternatively, you can use a return statement:

p -> {

return p.getGender() == Person.Sex.MALE

&& p.getAge() >= 18

&& p.getAge() <= 25;

}

Valid usage of Lambdas

print(() -> true)  
print(a-> a.startsWith(“test”));  
print((String a) -> a.startsWith(“test”));  
print((a,b) -> a.startsWith(“test”));  
print((String a,String b) -> a.startsWith(“test”));

This is valid, you can still use parenthesis to surround a single parameter   
print((i) -> I ==5));

Invalid   
print(a,b -> a.startsWith(“test”)); // invalid because it needs parenthesis around a,b  
print(a-> {a.startsWith(“test”);}); // invalid because missing return keyword  
print(a->{return a.startsWith(“test”)}); // invalid because missing semicolon after (“test”)

The type in the lambda must match the generic declared on the Predicate.

I.e. Predicate<String> s = (Object o) -> **true**;

Will generate a compile error because the complier expects a String and not an object.

A predicate with no type defined will treat the parameter line an Object.

The following line will fail because Predicate interface has a method that can only accept one parameter.

Predicate<String> str = (s1,s2) -> s1.isEmpty(); // COMPILE FAIL , wrong number of parameters.