Lambdas and Functional Interfaces

1. Writing Simple Lambdas

En Java 8, el lenguaje agregó la capacidad de escribir código usando otro estilo. La programación funcional es una forma de escribir código más declarativamente. Usted especifica lo que quiere hacer en lugar de tratar con el estado de los objetos. Te enfocas más en las expresiones que en los bucles.

La programación funcional usa expresiones lambda para escribir código. Una expresión lambda es un bloque de código que se pasa. Puedes pensar en una expresión lambda como método anónimo. Tiene parámetros y un cuerpo como lo hacen los métodos completos, pero no tiene un nombre como un método real. Solo las expresiones lambda más simples están en el examen OCA.

1.1. LAMBDA EXAMPLE

Nuestro objetivo es imprimir todos los animales en una lista de acuerdo con algunos criterios:

```
public class Animal {
  private String species;
  private boolean canHop;
  private boolean canSwim;
  public Animal(String speciesName, boolean hopper, boolean swimmer) {
    species = speciesName;
    canHop = hopper;
    canSwim = swimmer;
  }
   public boolean canHop() { return canHop; }
   public boolean canSwim() { return canSwim; }
   public String toString() { return species; }
}

public interface CheckTrait {
  boolean test(Animal a);
}

public class CheckIfHopper implements CheckTrait {
  public boolean test(Animal a) {
    return a.canHop();
  }
}
```

Ahora tenemos todo lo que necesitamos para escribir nuestro código para encontrar los animales que brincan (*hop*):

```
1: public class TraditionalSearch {
2:   public static void main(String[] args) {
3:     List<Animal> animals = new ArrayList<Animal>();  // list of animals
4:     animals.add(new Animal("fish", false, true));
5:     animals.add(new Animal("kangaroo", true, false));
6:     animals.add(new Animal("rabbit", true, false));
7:     animals.add(new Animal("turtle", false, true));
8:
```

```
9:     print(animals, new CheckIfHopper());      // pass class that does check
10:   }
11:  private static void print(List<Animal> animals, CheckTrait checker) {
12:     for (Animal animal : animals) {
13:       if (checker.test(animal))               // the general check
14:         System.out.print(animal + " ");
15:     }
16:     System.out.println();
17:   }
18: }
```

Podríamos reemplazar la línea 9 por la siguiente, que usa una lambda:

```
9:    print(animals, a -> a.canHop());
```

Solo tenemos que agregar una línea de código, no hay necesidad de una clase adicional para hacer algo simple. Aquí está esa otra línea:

```
print(animals, a -> a.canSwim());
```

¿Qué hay de los animales que no pueden nadar (*swim*)?

```
print(animals, a -> ! a.canSwim());
```

Este código usa un concepto llamado ejecución diferida. La ejecución diferida significa que el código está especificado ahora, pero se ejecutará más tarde. En este caso, más tarde es cuando el método `print()` lo llama.

1.2. LAMBDA SYNTAX

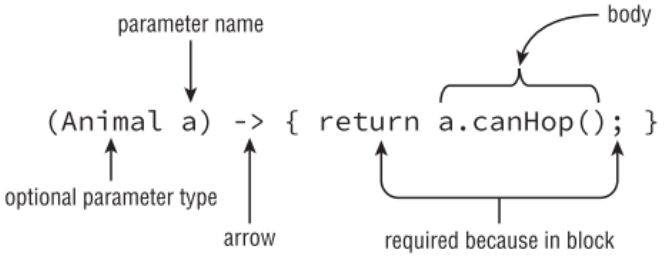
La sintaxis de lambdas es complicada porque muchas partes son opcionales. Estas dos líneas hacen exactamente lo mismo:

```
(Animal a) -> { return a.canHop(); }     //(1)
a -> a.canHop()                          //(2)
```

Para (1): estamos pasando esta lambda como el segundo parámetro del método `print()`. Ese método espera un `CheckTrait` como el segundo parámetro. Como en su lugar aprobamos una lambda, Java intenta asignar nuestra lambda a esa interfaz:

```
boolean test(Animal a);
```

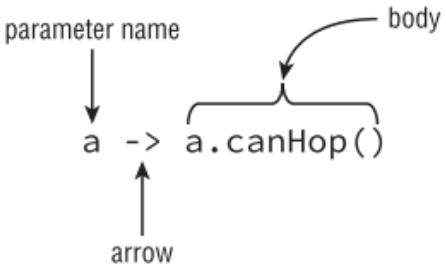
Como el método de esa interfaz toma un Animal, eso significa que el parámetro lambda tiene que ser un Animal. Y dado que el método de esa interfaz arroja un valor booleano, sabemos que lambda devuelve un valor booleano.



Tenemos:

1. Especifique un solo parámetro con el nombre `a` y estableciendo que el tipo es `Animal`.
2. El operador de flecha para separar el parámetro y el cuerpo.

3. Un cuerpo que tiene una o más líneas de código, incluyendo un punto y coma y una declaración de retorno.



Tenemos:

1. Especifique un solo parámetro con el nombre `a`.
2. El operador de flecha para separar el parámetro y el cuerpo.
3. Un cuerpo que llama a un solo método y devuelve el resultado de ese método.

Nota
Aquí hay un dato curioso: `s -> {}` es una lambda válida. Si no hay código en el lado derecho de la expresión, no necesita el punto y coma ni la declaración de retorno.

Este atajo especial no funciona cuando tenemos dos o más declaraciones. Veamos algunos ejemplos de lambdas válidos:

Lambda	# Parámetros
`() -> true`	0
`a -> a.startsWith("test")`	1
`(String a) -> a.startsWith("test")`	1
`(a, b) -> a.startsWith("test")`	2
`(String a, String b) -> a.startsWith("test")`	2

Ahora asegurémonos de que pueda identificar la sintaxis inválida. Por ejemplo:

Invalid Lambda	Motivo
`a, b -> a.startsWith("test")`	Sin paréntesis
`a -> { a.startsWith("test"); }`	Sin return
`a -> { return a.startsWith("test") }`	Sin punto y coma (;)

Es posible que haya notado que todas nuestras lambdas devuelven un booleano. Esto se debe a que el alcance del examen OCA limita lo que necesita aprender.

¿A qué variables puede acceder mi Lambda?
Lambdas tienen permitido acceder a las variables. Aquí hay un ejemplo:

```
boolean wantWhetherCanHop = true;
print(animals, a -> a.canHop() == wantWhetherCanHop);
```

El truco es que no pueden acceder a todas las variables. Instancia y variables estáticas están bien. Los parámetros del método y las variables locales están bien si no se les asignan nuevos valores.

Como Java no nos permite re declarar una variable local, el siguiente es un problema:

```
(a, b) -> { int a = 0; return 5;}     // DOES NOT COMPILE
```

Intentamos re declarar `a`, lo cual no está permitido. Por el contrario, la siguiente línea está bien porque usa un nombre de variable diferente:

```
(a, b) -> { int c = 0; return 5;}
```

2. Introducing Functional Interfaces

Lambdas trabaja con interfaces que tienen solo un método abstracto (SAM: *Single Abstract Method*). Estas se llaman interfaces funcionales, interfaces que se pueden usar con programación funcional. (En realidad es más complicado que esto).

Note
Java proporciona una anotación `@FunctionalInterface` en algunas interfaces funcionales, pero no en todas. Esta anotación significa que los autores de la interfaz prometen que será segura de usar en una lambda en el futuro. Sin embargo, el hecho de que no vea la anotación no significa que no sea una interfaz funcional. Recuerde de tener exactamente un método abstracto es lo que la convierte en una interfaz funcional, no la anotación.

2.1. PREDICATE

Puedes imaginar que tendríamos que crear muchas interfaces como esta para usar lambdas. Java reconoce que este es un problema común y nos proporciona una interfaz de este tipo:

```
public interface Predicate<T> {
  boolean test(T t);
}
```

Esto significa que ya no necesitamos nuestra propia interfaz y podemos poner todo lo relacionado con nuestra búsqueda en una sola clase:

```
1: import java.util.*;
2: import java.util.function.*;
3: public class PredicateSearch {
4:   public static void main(String[] args) {
5:     List<Animal> animals = new ArrayList<Animal>();
6:     animals.add(new Animal("fish", false, true));
7:
8:     print(animals, a -> a.canHop());
9:   }
10:   private static void print(List<Animal> animals, Predicate<Animal> checker) {
11:    for (Animal animal : animals) {
12:      if (checker.test(animal))
13:         System.out.print(animal + " ");
14:     }
15:     System.out.println();
16:  }
17: }
```

2.2. CONSUMER

La interfaz funcional del consumidor tiene un método que debe conocer:

```
void accept(T t)
```

Echemos un vistazo al código que usa un `Consumer`:

```
public static void main(String[] args) {
   Consumer<String> consumer = x -> System.out.println(x);
   print(consumer, "Hello World");
}
private static void print(Consumer<String> consumer, String value) {
   consumer.accept(value);
}
```

2.3. SUPPLIER

La interfaz funcional SUPPLIER tiene un solo método:

```
T get()
```

Un buen caso de uso para `Supplier` es la generación de valores. A continuación, se muestran dos ejemplos:

```
Supplier<Integer> number = () ->  42;
Supplier<Integer> random = () ->  new Random().nextInt();
```

Otro ejemplo:

```
public static void main(String[] args) {
   Supplier<Integer> number = () ->  42;
   System.out.println(returnNumber(number));
}

private static int returnNumber(Supplier<Integer> supplier) {
   return supplier.get();
}
```

2.4. COMPARATOR

Aprendimos las reglas. Un número negativo significa que el primer valor es menor, cero significa que son iguales y un número positivo significa que el primer valor es mayor. La firma del método es la siguiente:

```
int compare(T o1, T o2)
```

Note
La interfaz `Comparator` existía antes de que se añadieran lambdas a Java. Como resultado, está en un paquete diferente. Puede encontrar `Comparator` en `java.util`.

Solo tienes que saber `compare()` para el examen. ¿Puedes averiguar si esto se ordena en orden ascendente o descendente?

```
Comparator<Integer> ints = (i1, i2) -> i1 - i2;  //Orden ascendente
```

El `Comparator ints` utiliza un orden de clasificación natural. Si el primer número es mayor, devolverá un número positivo.

¿Crees que estas dos declaraciones se clasificarían en orden ascendente o descendente?

```
Comparator<String> strings = (s1, s2) -> s2.compareTo(s1);
Comparator<String> moreStrings = (s1, s2) -> - s1.compareTo(s2);
```

Ambos comparadores en realidad hacen lo mismo: ordenar en orden descendente.

Interfaces funcionales básicas:

Interfaces funcionales	# parámetros	Tipo de Retorno
`Comparator`	2	`Int`
`Consumer`	1	`void`
`Predicate`	1	`boolean`
`Supplier`	Ninguno	`One (type varies)`

3. Working with Variables in Lambdas

Las variables pueden aparecer en tres lugares con respecto a las lambdas: la lista de parámetros, las variables locales declaradas dentro del cuerpo lambda y las variables referenciadas desde el cuerpo lambda.

3.1. PARAMETER LIST

Anteriormente en este capítulo, aprendió que especificar el tipo de parámetros es opcional. Además, se puede utilizar `var` en lugar del tipo específico. Eso significa que las tres declaraciones son intercambiables:

```
Predicate<String> p = x -> true;
Predicate<String> p = (var x) -> true;
Predicate<String> p = (String x) -> true;
```

En nuestro ejemplo, la respuesta es `String`. ¿Cómo nos dimos cuenta de eso? Una lambda infiere los tipos del contexto circundante. Eso significa que puedes hacer lo mismo.

Otro lugar para buscar el tipo es la firma de un método. Probemos con otro ejemplo. ¿Puedes averiguar el tipo de x?

```
public void whatAmI() {
   consume((var x) -> System.out.print(x), 123);
}
public void consume(Consumer<Integer> c, int num) {
   c.accept(num);
```

```
}
```

Dado que el método `consume()` espera un `Integer` como genérico, sabemos que ese es el tipo inferido de x.

En algunos casos, puede determinar el tipo sin siquiera ver la firma del método. ¿Cuál crees que es el tipo de x aquí?

```
public void counts(List<Integer> list) {
  list.sort((var x, var y) -> x.compareTo(y));
}
```

Dado que estamos ordenando una lista, podemos usar el tipo de la lista para determinar el tipo de parámetro lambda.

3.2. LOCAL VARIABLES INSIDE THE LAMBDA BODY

Si bien es más común que un cuerpo lambda sea una sola expresión, es legal definir un bloque.

Por ejemplo:

```
(a, b) -> { int c = 0; return 5;}
```

Ahora probemos con otro. ¿Ves lo que está mal aquí?

```
(a, b) -> { int a = 0; return 5;}     // DOES NOT COMPILE
```

Intentamos volver a declarar a, lo cual no está permitido. Java no le permite crear una variable local con el mismo nombre que una ya declarada en ese ámbito. Ahora intentemos uno difícil. ¿Cuántos errores de sintaxis ve en este método?

```
11: public void variables(int a) {
12:    int b = 1;
13:    Predicate<Integer> p1 = a -> {   // a ya ha sido declarado
14:       int b = 0;                    // se redeclara b
15:       int c = 0;
16:       return b == c;}               // Falta el ; de la sentencia de p1
17: }
```

3.3. VARIABLES REFERENCED FROM THE LAMBDA BODY

Los cuerpos lambda pueden hacer referencia a algunas variables del código circundante. El siguiente código es legal:

```
public class Crow {
   private String color;
   public void caw(String name) {
      String volume = "loudly";
      Consumer<String> consumer = s ->
            System.out.println(name + " says "
                  + volume + " that she is " + color);
   }
```

```
}
```

Esto muestra que lambda puede acceder a una variable de instancia, parámetro de método o variable local bajo ciertas condiciones. Siempre se permiten variables de instancia (y variables de clase).

Se permite hacer referencia a parámetros de método y variables locales si son **efectivamente finales**. Esto significa que el valor de una variable no cambia después de que se establece, independientemente de si se marca explícitamente como final.

El siguiente código es similar al anterior (solo se agrega modificador **final**):

```
public class Crow {
   private String color;
   public void caw(final String name) {
      final String volume = "loudly";
      Consumer<String> consumer = s ->
         System.out.println(name + " says "
            + volume + " that she is " + color);
   }
}
```

El siguiente código no compila:

```
2:  public class Crow {
3:     private String color;
4:     public void caw(String name) {
5:        String volume = "loudly";
6:        name = "Caty";                      //name se esta modificando
7:        color = "black";                    //variables de instancia
8:                                            //son permitidas modificar
9:        Consumer<String> consumer = s ->
10:          System.out.println(name + " says "
11:             + volume + " that she is " + color);
12:       volume = "softly";                  //Variable local no debe modificarse
13:    }
14: }
```

En resumen:

Variable type	Rule
Instance variable	Permitido
Static variable	Permitido
Local variable	Permitido si son efectivamente `final`
Method parameter	Permitido si son efectivamente `final`
Lambda parameter	Permitido

4. Calling APIs with Lambdas

4.1. REMOVEIF()

`List` y `Set` declaran un método `removeIf()` que toma un `Predicate`.

Por ejemplo, si deseamos eliminar un elemento cuyo nombre no empiece con h:

```
3: List<String> bunnies = new ArrayList<>();
4: bunnies.add("long ear");
5: bunnies.add("floppy");
6: bunnies.add("hoppy");
7: System.out.println(bunnies);      // [long ear, floppy, hoppy]
8: bunnies.removeIf(s -> s.charAt(0) != 'h');
9: System.out.println(bunnies);      // [hoppy]
```

El método `removeIf()` funciona de la misma manera en un Set. Elimina cualquier valor del conjunto que coincida con el predicado. No hay un método `removeIf()` en un `Map`. Recuerde que los mapas tienen tanto claves como valores. ¡No estaría claro qué se estaba quitando!

4.2. SORT()

El método `sort()` toma `Comparator` que proporciona el orden de clasificación. Recuerde que `Comparator` toma dos parámetros y devuelve un int. Por ejemplo:

```
3: List<String> bunnies = new ArrayList<>();
4: bunnies.add("long ear");
5: bunnies.add("floppy");
6: bunnies.add("hoppy");
7: System.out.println(bunnies);      // [long ear, floppy, hoppy]
8: bunnies.sort((b1, b2) -> b1.compareTo(b2));
9: System.out.println(bunnies);      // [floppy, hoppy, long ear]
```

No hay un método `sort` en `Set` o `Map`. Ninguno de esos tipos tiene indexación, por lo que no tendría sentido ordenarlos.

4.3. FOREACH()

Toma un `Consumer` y llama a esa lambda para cada elemento encontrado. Por ejemplo:

```
3: List<String> bunnies = new ArrayList<>();
4: bunnies.add("long ear");
5: bunnies.add("floppy");
6: bunnies.add("hoppy");
7:
8: bunnies.forEach(b -> System.out.println(b));
9: System.out.println(bunnies);
```

Este código imprime lo siguiente:

```
long ear
floppy
hoppy
[long ear, floppy, hoppy]
```

Podemos usar `forEach()` con un conjunto o mapa. Para un conjunto, funciona de la misma manera que una lista. Por ejemplo:

```
Set<String> bunnies = Set.of("long ear", "floppy", "hoppy");
bunnies.forEach(b -> System.out.println(b));
```

Para un mapa, debe elegir si desea pasar por las claves o los valores:

```
Map<String, Integer> bunnies =  new HashMap<>();
bunnies.put("long ear", 3);
bunnies.put("floppy", 8);
bunnies.put("hoppy", 1);
bunnies.keySet().forEach(b -> System.out.println(b));
bunnies.values().forEach(b -> System.out.println(b));
```

Resulta que los métodos `keySet()` y `values()` devuelven cada uno un `Set`. Ya que sabemos cómo usar `forEach()` con un `Set`, ¡esto es fácil!

USING FOREACH() WITH A MAP DIRECTLY

Java tiene una interfaz funcional llamada `BiConsumer`. Funciona igual que `Consumer`, excepto que puede tomar dos parámetros. Esta interfaz funcional le permite usar `forEach ()` con pares clave/valor de `Map`.

```
Map<String, Integer> bunnies = new HashMap<>();
bunnies.put("long ear", 3);
bunnies.put("floppy", 8);
bunnies.put("hoppy", 1);
bunnies.forEach((k, v) -> System.out.println(k + " " + v));
```

5. Review Questions

5.1.

What is the result of the following class?

```
1:  import java.util.function.*;
2:
3:  public class Panda {
4:     int age;
5:     public static void main(String[] args) {
6:        Panda p1 = new Panda();
7:        p1.age = 1;
8:        check(p1, p -> p.age < 5);
9:     }
10:    private static void check(Panda panda,
11:       Predicate<Panda> pred) {
12:       String result =
13:          pred.test(panda) ? "match" : "not match";
14:       System.out.print(result);
15: } }
```

A. match
B. not match
C. Compiler error on line 8.

D. Compiler error on lines 10 and 11.
E. Compiler error on lines 12 and 13.
F. A runtime exception is thrown.

5.2.

Which of these statements is true about the following code?

```
public void method() {
 x((var x) -> {}, (var x, var y) -> 0);
}
public void x(Consumer<String> x, Comparator<Boolean> y) {
}
```

A. The code does not compile because of one of the variables named x.
B. The code does not compile because of one of the variables named y.
C. The code does not compile for another reason.
D. The code compiles, and the `var` in each lambda refers to the same type.
E. The code compiles, and the `var` in each lambda refers to a different type.

5.3.

Which of the following can be inserted without causing a compilation error? (Choose all that apply.)

```
public void remove(List<Character> chars) {
 char end = 'z';
 chars.removeIf(c -> {
      char start = 'a'; return start <= c && c <= end; });
      // INSERT LINE HERE
}
```

A. `char start = 'a';`
B. `char c = 'x';`
C. `chars = null;`
D. `end = '1';`
E. None of the above

5.4.

How many lines does this code output?

```
Set<String> set = Set.of("mickey", "minnie");
List<String> list = new ArrayList<>(set);

set.forEach(s -> System.out.println(s));
list.forEach(s -> System.out.println(s));
```

A. 0
B. 2
C. 4
D. The code does not compile.
E. A runtime exception is thrown.

5.5.

Which is true of the following code?

```
int length = 3;

for (int i = 0; i<3; i++) {
 if (i%2 == 0) {
      Supplier<Integer> supplier = () -> length; // A
      System.out.println(supplier.get());        // B
 } else {
      int j = i;
      Supplier<Integer> supplier = () -> j;      // C
      System.out.println(supplier.get());        // D
 }
}
```

A. The first compiler error is on line A.
B. The first compiler error is on line B.
C. The first compiler error is on line C.
D. The first compiler error is on line D.
E. The code compiles successfully.