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

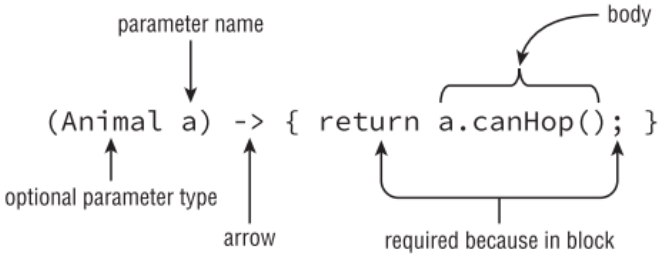
La sintaxis de lambdas es complicada porque muchas partes son opcionales. Estas dos líneas hacen exactamente lo mismo:

```
(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);
```

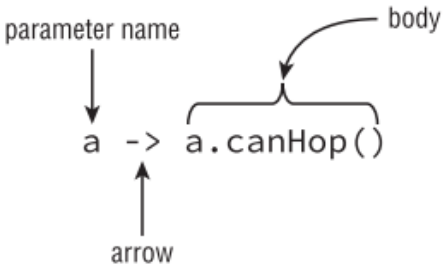
Como el método de esa interfaz toma un Animal, eso significa que el parámetro lambda tiene que ser un Animal. Y dado que el método de esa interfaz arroja un valor booleano, sabemos que lambda devuelve un valor booleano.



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: }
```

3. Review Questions

3.1.

3.2.

3.3.

3.4.

3.5.