15 - REVIEW Java Lambdas

Java Lambdas

Benefits of Lambdas in Java

- Enabling functional programming
  - Being able to pass behaviours as well as data to functions (mind that Java supports higher ordering of one level: a function to a function. Pure functional languages as Haskell supports unlimited depth of higher ordering)
  - introduction of lazy evaluation with stream processing
- Writing cleaner and more compact code
- Facilitating parallel programming: without side effects there are not race conditions, hence the parallelism is improved
- Developing more generic, flexible and reusable APIs

Syntax

```java
List<Integer> intSeq = Arrays.asList(1,2,3);
intSeq.forEach(x -> System.out.println(x));
```

- `forEach()` is invoked on `intSeq`, also introduced with Java 8
- `x -> System.out.println(x)` is the lambda expression that defines an *anonymous function* (method) with one parameter name `x` of type `Integer` (the type is inferred for `intSeq`, which is a list of `Integer`)
  - alternative syntax
    - `(Integer x) -> System.out.println(x)`
    - `(x -> {System.out.println(x)})`
      - braces needed if we want to define a block, a sequence of instruction separated by the semicolon ;
    - `(System.out::println)`
      - call the method `println` of the object `out`, the right overloaded method is retrieved by type inference
      - this is called method reference and it can be used to pass an existing function in places where a lambda is expected
      - the signature of the referenced method needs to match the signature of the functional interface method

Variables in Lambdas

No new Scope

Mind that a lambda expression do not introduce a new scope even if we define a new block. The param variables and the variables of the body of the lambda expr have the scope of the block in which the lambda expr is defined.

Example:

```java
int x = 0;
List<Integer> intSeq = Arrays.asList(1,2,3);
intSeq.forEach(x -> {
        System.out.println(x+2);
})
```

- The code in the example do not compile since the `x` before the `->` is an already used identifier!

Local and Static Variable Capture

Like local and anonymous classes, lambda expressions can capture variables; they have the same access to local variables of the enclosing scope. However, unlike local and anonymous classes, lambda expressions do not have any shadowing issues (see Shadowing for more information). Lambda expressions are lexically scoped. This means that they do not inherit any names from a supertype or introduce a new level of scoping.

Local variables used inside the body of a lambda expression must be final/effectively-final or static, otherwise the code do not compile

```java
List<Integer> intSeq = Arrays.asList(1,2,3);
int var = 10;

intSeq.forEach(x -> System.out.println(x + var));
var = 3 // with this line the code does not compile
```

- `var` is a local variable of the code snippet
- vas is visible to the lambda expression because the lambda expressions have as scope the scope of the block in which they are declared

This restriction is given by the fact that Java do not use closures or AR retention.

- `var` is static: there is only one representation of `var` and it is embedded in the class file in the non-heap area (Java Virtual Machine > Non Heap)
- `var` is `final` or effectively-final: the compiler substitute the identifier `var` with its value in all the compiled code

Without the constraint of static/final we would have the problem of having undefined variables that appear in the body of the lambda expression:

- inside a method you define a variable `x` and then a lambda expression that use `x`
- pass the lambda expr to another function exploiting the higher order using a setter method
- the method returns and its AR is popped from the stack. The variable `x` was defined in the AR that has been popped, for the lambda expression there is no way to retrieve the value of `x` when it will be executed

Implementation of Lambdas

The compiler first converts a lambda expression into a function, compiling its code then it generates the code to call the compiled expression where needed.

What type should be generated for the compiled lambda? How it should be called? In which class it should be in?

Functional Interface

As a design decision the lambdas are instances of *functional interfaces*: a java interface with exactly one abstract method

Functional Interfaces can be used as *target type* of lambda expressions

- lambdas exprs can be assigned to variables that have the right type (the right functional interface)
- lambdas exprs can be passed as arguments or returned from functions provided that the formal parameters or the return type is of the same functional interface

Lambdas can be interpreted as instances of anonymous inner classes implementing the functional interface

- *the lambda expr provide an implementation to the (only) abstract method of the functional interface so they are considered a class that implements the interface.*

Default Methods

We said that in Java we see lambdas expression as anonymous classes that implements a functional interface and the lambda expression is the implementation of the method defined in the interface.

Now we have a problem. Let's take the interface `List<E>`, what if we want to define a new method `sort` in the interface to which we can pass a lambda expression?
This is a challenge for the retro compatibility because by design every class that implements an interface have to redefine all of the interface's methods.
Adding new abstract methods to an interface breaks existing implementations of the interface

Java 8 allows interface to include

- abstract (instance) methods, as usual
- static methods
- default methods, defined in terms of other possibly abstract methods

A class that implements an interface have to redefine the abstract methods but not necessarily the static-default methods (static-default methods may even be concrete, have an actual implementation)

example

```java
interface List<E>{
        ...
        default boolean removeIf(Predicate<? super E> filter)
        ...
}
```

- `removeIf()` is not inerithed by the already compiled and existing implementations of list
- `removeIf()` takes as parameter a `Predicate`, which is a functional interface: we can pass as parameter a lambda expression