Lambda Expressions in Java 8

* Lambda also Known as closures is a new expression introduced in JDK 8 using Netbeans 7.4 IDE * Lambda: Clear and concise way to represent a method interface using an expression * Closures: Strictly speaking a closure is a Lambda implementation that has all free variables bound to an environment giving them a value. Lambda expressions are implemented using closures and therefore the terms are used interchangeably. * “But as I learned more about the subtleties of the changes, it became clear that an entire new book (Mastering Lambdas) was needed” –Maurice Naftalin (renowned author in the java community) * By: Austin Russell, Aubree Lytwyn, Matthew Uhlar, Ryan Smith =

Proposal

- Java has a form of closures: anonymous inner classes
- Reasons for proposal
  - Bulky syntax -> Addressed substantially for improvements
  - Inability to capture non-final local variables -> Allow compiler capture of effectively final local variables
  - Transparency issues surrounding the meaning of return, break, continue and 'this' -> Making 'this' lexically scoped
  - No nonlocal control flow operators -> Not Addressed

Proposal

- Lambda Expressions
  - Aimed at correcting the vertical problem with API classes and anonymous inner functions
  - Disadvantages
    - Mixing structural and nominal types
    - Divergence of library styles from callback objects to function types
    - Generic types are erased, which would expose additional places where developers are exposed to erasure

What is Lambda used for?

- Improve libraries to make iterations, filtering and data extraction easier
  - Main Example: Collection Library
    - Concurrency features improve performance in a multicore environment
    - lambdas can be understood as a kind of anonymous method with a more compact syntax that also allows the omission of modifiers, return type, and in some cases parameter types as well.

Lambda Type

- What is the type
  - The target type for a lambda expression Must be a functional interface and be compatible with the target type. Same parameter type as interface function type.
  - Lambda can be used recursively when you are doing variable assignment. Specifically static variable assignment because of the assignment before use rule for local variables

Where can you use lambda Expressions

- Any context that has a target type
  - Ex: Variable Declarations and array initializers
  - Return Statements where target type == return Type
  - Method or constructor arguments where target type is the type of the parameter
  - Lambda expression bodies for which the target type is the type of the body

```
Callable<Runnable> c = () ->
 () -> system.out.println(\hi");};
```

  - Target type being Callable and the the lambda body is the function type of Runnable which takes no arguments and returns no values
  - The expression does not allow for ambiguity example

```
Object o = () -> {system.out.println(\hi");};
Object o = (Runnable) () ->
{system.out.println(\hi");};
```

Scoping Rules

- Names in the body of a lambda are interpreted exactly as in the environment that it resides in. except for new names for the lambda expressions formal parameters.
- Formal parameters follow the same rules as method parameters for shadowing class and instance variables
- Example
  - Can do:

    ```
    Class Bar { int i; Foo foo = i -> i*4; };
    ```

  - Lambda parameter i shadows the instance variable
  - Can't do:

    ```
          Void bar() {int i; Foo foo = i -> i *4; };
    ```

  - Illegal because I is already defined and with local variables shadowing is not possible

Functional Interfaces previously/currently known as Single Abstract Method (SAM)

- An expression whose type can be used for a method parameter when a lambda is supplied as the actual argument
- An interface that has exactly one explicitly declared abstract method. This is necessary because an interface may have non-abstract default methods
- Because a functional interface contains only one abstract method, you can omit the name of that method when you implement it
- Example

```
public interface Runnable { void run(); }
```

Expression Syntax

- Lambda expressions address the bulkiness of anonymous inner classes
- Basic syntax
  - (parameters) -> expression

```
// takes two integers and returns their sum
(int x, int y) -> x+y
```

Anonymous Inner Class (AIC)

- What is it?
  - Can be used to create a subclass of an abstract class or a concrete class
  - Can provide a concrete implementation of an interface
    - including the addition of fields
  - AIC's introduce a new scope
- When is it used?
  - An instance of an AIC can be referred to using this in it's method bodies
  - Further methods can be called on it
    - State can be mutated over time
  - Majority of the time its used to provide stateless implementations of simgle functions
- Many instances like those above can be replaced with lambdas but some cannot

Runnable Lambda

- The runnable lambda expression converts five lines of code into one statement
- Anonymous runnable Ex: `'Java Runnable r1 = new Runnable(){ @Override Public void run(){ System.out.println(\Hello world one!"); } };`

```
Runnable r2 = () ->
 System.out.println(\Hello world two!");
```

Comparator Lambda

- used for sorting collections

```
collections.sort(personList, (p1, p2) ->
 p2.getSurName().compareTo(p1.getSurName()));
```

Listener Lambda

- Lambdas offer simple solutions to event driven programming paradigms
- Listener lambdas can listen/ handle events inline
  - JButtons
  - RadioButtons
  - print alert / send message

Evolution of a Lambda (Naive)

- Naive long form variation

```
List<T> someList = ...

class compareElements implements comparator<T> {
  @override
  Public bool compare(T elem1, T elem2) {
    return elem1.getField().compareTo(
     elem2.getField())
  }
}

Collections.sort(someList, new compareElements())
```

Evolution of a Lambda (Intermediate)

```
List<T> someList = ...

Collections.sort(someList, (T elem1, T elem2){
  return elem1.getField().compareTo(
   elem2.getField());})
```

- Better, more concise. However still room for improvement

```
List<T> someList = ...

Collections.sort(someList, (T elem1, T elem2) ->
 elem1.getField().compareTo(elem2.getField()))
```

- We can do better still. . .

Evolution of a Lambda (Final)

- Final lambda
- Types can be inferred from interface

```
List<T> someList = ...

Collections.sort(someList, (elem1, elem2) ->
 getField().compareTo(elem2.getField()))
```

API's (Problems)

- Problems with API's
  - API classes like CallBackHandler, Runnable, Callable, EventHandler or Comparator use single abstract method
  - To utilize these you often have to write an anonymous inner class like so:

    ```
    foo.doSomething(new CallBackHandler()) {
        Public void callback(Context c) {
            System.out.println("Success");
        }
    }
    ```

  - These are very bulky. Creates what is often a vertical problem using 5 lines of source code for single idea

API's (Solution)

- Lambda expression solution
  - Replace machinery of anonymous inner classes with a simpler mechanism by adding function types to the language
  - Simplified to

    ```
    CallBackHandler cd = #{ c ->
     System.out.println("success")};
    ```

Summary

- Lambda expressions in Java introduce the idea of functions into the language
- Lambdas are a powerful feature that work directly with SAM types.
- Previously complex syntax that utilizes anonymous inner classes has been drastically simplified
- For the first time in Java's history we find something that cannot be assigned to a reference of type object

References

- References
  - Oracle.com
  - Lambdafaq.org
  - JCP.org
  - openjdk.java.net
  - cr.openjdk.java.net
  - Stackoverflow.com
- Useful links
  - YouTube Lambda Walkthrough