Lesson 3.8 Lambda Expressions and Java 8

Software Analysis and Design

2º Year, Computer Science

Universidad Autónoma de Madrid

Index

- **New concepts in interfaces**
  - **Default methods**
  - **Static methods**
- Lambda expressions
- Exercises
- Conclusions and bibliography

New concepts in interfaces

- Consider the following interface

```
public interface Tree<T>{
  T getElement();
  Tree<T> leftChild();
  Tree<T> rightChild();
  boolean isLeaf();
  boolean isEmpty();
  Tree<T> search(T o);
}
```

- For ease of use, we could give a `default` implementation for some methods
- This implementation can use other methods of the interface
- Classes do not need to give an implementation of a default method, but they can

`default` methods in interfaces

```
public interface Tree<T>{
  T getElement();
  Tree<T> leftChild();
  Tree<T> rightChild();
  default boolean isLeaf() {
      return  this.leftChild().isEmpty() && this.rightChild().isEmpty();
  }
  boolean isEmpty();
  default Tree<T> search(T o) {
    if (this.getElement().equals(o)) return this;
    else {
      Tree<T> result = null;
      if (! this.leftChild().isEmpty() ) result = this.leftChild().search(o);
      if (result != null) return result;
      if (! this.rightChild().isEmpty() ) result = this.rightChild().search(o);
      if (result != null) return result;
    }
    return null;
  }
}
```

`default` methods: motivation

- Being able to add methods to an interface without breaking already existing code (the new methods of the interface would be default methods)

- Specify methods that are optional
  - Give an implementation that returns an exception

- Facilitate the implementation of interfaces (similar to an abstract class with a reference implementation)

Differences with abstract classes

- An interface does not have internal state
  - It cannot declare attributes (instance variables), but just constants
- A class can only extend one class, while it can implement multiple interfaces
- The purpose of interfaces is still specifying "*what*" (method signatures) and not "*how*" (code in methods)

Static methods in interfaces

- It is possible to add static methods to an interface, just like in classes
- Useful to define libraries
  - Example: creation of some useful comparators in Comparator<T>

```
public interface Comparator<T> {
   int compare(T o1, T o2);
   static <T extends Comparable<? super T>> Comparator<T> naturalOrder() { … }

   static <T> Comparator<T> nullsFirst(Comparator<? super T> comparator) { … }
   static <T> Comparator<T> nullsLast(Comparator<? super T> comparator) { … }
   //…
}
```

Index

- New concepts in interfaces
- **Lambda Expressions**
  - **Introduction and examples**
  - **Lambda expressions**
  - **Streams [skipped this year]**
- Exercises
- Conclusions and bibliography

Introduction and Examples

What are lambda expressions?

- Functions as first-level concepts
  - Anonymous, without a name
  - We can pass them as parameters

- In Java 8 they are called lambda expressions
- The name comes from λ-calculus (Alonzo Church)
- In other languages (e.g, Ruby) *closures* are a similar concept

- They promote a programming style closer to functional programming
  - Function chaining, operating over *streams*
  - More easily parallelizable (useful to process large amounts of data – Big Data computations)
  - More intentional and less verbose code

Lambda expressions. Example.

- In Swing, we often need to configure graphical components with *callback* methods, that are executed when an event occurs
- Before Java 8, we needed to define a class to define the callback method.

Anonymous class

Method of interest for the button

```
button.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent event) {
    System.out.println("button clicked");
  }
});
```

- A lambda expression permits a more concise syntax

parameter

expression body

```
button.addActionListener(event -> System.out.println("button clicked"));
```

Lambda expression

Another example: filtering a list

Programming using Java 7 style

```
class Product {
  private int price;
  public Product(int p) { this.price = p; }
  public int getPrice() { return this.price; }
}
```

```
List<Product> products = Arrays.asList(
                    new Product(20),
                    new Product(40),
                    new Product (5));

List<Product> discounts = new ArrayList<Product>();

for (Product p : products)
    if (p.getPrice()>10.0)
      discounts.add(p);
```

Another example: filtering a list

Programming with lambdas

```
class Product {
  private int price;
  public Product(int p) { this.price = p; }
  public int getPrice() { return this.price; }
}


List<Product> products = Arrays.asList(
                new Product(20),
                new Product(40),
                new Product (5));

List<Product> discounts = products.stream().
          filter(p -> p.getPrice()>10.0).     // filter those > 10
          collect(Collectors.toList());       // store them in a list
```

Syntactic sugar...

(and some more things)

```
Stream<Product> filter(Predicate<? super Product> a)
```

```
@FunctionalInterface
public interface Predicate<T> {
  //… more things
  boolean test(T t);
}
```

The compiler generates an anonymous class

```
List<Product> descs2 = products.stream().
    filter(new Predicate<Product>() {
                @Override public boolean test(Product a) {
                    return a.getPrice()>10.0;
                }
    }).collect(Collectors.toList());
```

Lambda Expressions

Lambda Expressions

What are they?

- A block of code with no name, made of:
  - List of formal parameters,
  - Separator “->”
  - Body.

```
(int x) -> x + 1
```

- Looks like a method, but it is not: it is an instance of a functional interface
- More precisely, it is a compact notation for an instance of an anonymous class, typed by a functional interface

Lambda expressions

What are they?

- A functional interface is an interface with a unique non-default, non-static method
- Some important functional interfaces

Name	Arguments	Return	Functional method	Examples
Predicate<T>	T	boolean	test(T t)	Does the product has a discount?
Consumer<T>	T	void	accept(T t)	Print a value
Function<T,R>	T	R	apply(T t)	Obtain the price of a Product
Supplier<T>	None	T	get()	Creation of an objet
UnaryOperator<T>	T	T	apply(T t)	Logical negation (!)
BinaryOperator<T>	(T, T)	T	apply(T t, T u)	Multiply two numbers (*)

- Some of them have specializations: IntConsumer
- Others contain *default* and *static* utility methods

Lambda expressions

What are they?

- Lambda expressions do not have
  - Name
  - Declaration of return type (it is inferred)
  - *throws* clause (it is inferred)
  - Declaration of generic types
- The types of the formal parameters can be omitted (implicit vs explicit lambdas)
  - Either all or none omitted
- If the type is included, we can add the `final` modifier to the parameters

Parameters and return

- With zero parameters and no return

```
Runnable noArguments = () -> System.out.println("Hello World");
noArguments.run();



 /* Equivalent to
 Runnable noArguments = new Runnable() {
                 @Override public void run() {
                      System.out.println("Hello World");
                 }
            };
 */
```

Parameters and return

- The following syntax is incorrect:

```java
Runnable noArguments = -> System.out.println("Hello World");
```

Parameters and return

- With one parameter, several instructions and no return:

```
Consumer<Product> consumer = p -> {
    p.increasePrice(10);
    System.out.println(p.getName()+": "+p.getPrice());
};


List<Product> products = Arrays.asList( new Product(20, "Salt"),
                                        new Product(40, "Sugar"),
                                        new Product (5, "Wine"));


products.forEach(p -> {
    p.increasePrice(10);
    System.out.println(p.getName()+": "+p.getPrice());
});

products.forEach(consumer);  // equivalent to the previous code
```

Parameters and return

- The following syntaxes are equivalent:

```
Consumer<Product> consumer = p -> { // implicit lambda
    p.increasePrice(10);
    System.out.println(p.getName()+": "+p.getPrice());
};
```

```
Consumer<Product> consumer = (p) -> {
    p.increasePrice(10);
    System.out.println(p.getName()+": "+p.getPrice());
};
```

```
Consumer<Product> consumer = (Product p) -> {  // explicit lambda
    p.increasePrice(10);
    System.out.println(p.getName()+": "+p.getPrice());
};
```

Parameters and return

- The following syntax is incorrect:

```
Consumer<Product> consumer = Product p -> {
    p.incrementPrice(10);
    System.out.println(p.getName()+": "+p.getPrice());
};
```

Parameters and return

- With two parameters and with return:

```
List<Integer> numbers = Arrays.asList(1, 1, 2, 3, 5, 8, 13);

// Optional is a type that admits a result or null…
// …permits a “functional” notation for if…then…else
Optional<Integer> result =
       numbers.stream().
               reduce((x, y) -> x+y); // BinaryOperator<Integer>

System.out.println("Sum="+result.orElse(0));
// if there is no result, prints 0
```

Parameters and return

- The following syntaxes are equivalent:

```
Optional<Integer> result =
    numbers.stream().
            reduce((x, y) -> { return x+y; });
```

```
Optional<Integer> result =
    numbers.stream().
        reduce((Integer x, Integer y) -> { return x+y; });
```

Context variables

- Inside a lambda, we can use external context variables that are final…

```
List<Product> products = Arrays.asList(new Product(20, "Salt"), new Product(40, "Sugar"), new Product (5, "Wine"));

final int increment = 10;

Products.forEach(p -> {
    p.increasePrice(increment);  //increment is final, we can use it
    System.out.println(p.getName()+": "+p.getPrice());
});
```

Context variables

- … o effectively final

```
List<Product> products = Arrays.asList(new Product(20, "Salt"), new
Product(40, "Sugar"), new Product (5, "Wine"));

int increment = 10;

Products.forEach(p -> {
    p.increasePrice(increment);  // we do not change increment, OK!
    System.out.println(p.getName()+": "+p.getPrice());
});
```

Context variables

- … o effectively final

```
List<Product> products = Arrays.asList(new Product(20, "Salt"), new
Product(40, "Sugar"), new Product (5, “Wine"));

int increment = 10;

Products.forEach(p -> {
    increment+=3;                   // ERROR!
    p.increasePrice(increment);  // we do not change increment, OK!
    System.out.println(p.getName()+": "+p.getPrice());
});
```

Exercise

Filters and maps

Ejercise

- Design a class that sequentially stores data of any type, and that can return a filtered sequence of that data by configurable criteria.
- Make it possible to chain filters

Exercise: *currying*

- In functional programming, currying is a widely used technique to reduce the parameters of a function
- Given a function $f: X \times Y \rightarrow Z$, curry(f) returns a function $h: X \rightarrow (Y \rightarrow Z)$.
- That is, h takes an argument of type X, and returns a function $Y \rightarrow Z$, defined in such a way that $h(x)(y) = f(x, y)$.
- To do:
  - Implement *curry* using lambda expressions.
  - Hint: the interface BiFunction<X, Y, Z> can be used to model f, and Function<Y, Z> to model h.

Exercise: *currying*

- In functional | dely used technique to | nction
- Given a funct | ns a function h: $X \to (Y \to Z)$
- That is, h tak | returns a function $Y \to$ | $h(x)(y) = f(x, y)$.
- To do:
  - Implemen | ons.
  - Hint: the i | can be used to model f, a

Solution

```
package currying;

import java.util.function.*;

public class Currying {
  private <X, Y, Z> Function<X, Function<Y, Z>> curry(BiFunction<X, Y, Z> f){
    return x -> (y -> f.apply(x, y));
  }

  public static void main(String ...args) {
    Currying c = new Currying();
    Integer result =
      c.<Integer, Integer, Integer>curry((x, y) -> x + y ).
        apply(3).
        apply(4);
     System.out.println(result);
   }
}
```

Functional interfaces

- A functional interface is an interface that has exactly one abstract method.

- Methods not qualifying for the unique method of a functional interface:
  - *default* methods
  - static methods
  - methods inherited from Object

- Functional interfaces can optionally be annotated with *@FunctionalInterface* (in `java.lang`)
  - The compiler checks that the declared interface is functional

Example (1/2)

// An object system with dynamic methods embedded
// in Java

```
@FunctionalInterface interface Method { // To which standard interface
  void exec(ProtoObject o);              // is it equivalent?
}

public class ProtoObject {
  private HashMap<String, Object> slots = new HashMap<>();
  private HashMap<String, Method> methods = new HashMap<>();

  public void add (String name, Method m) { this.methods.put(name, m); }
  public void add (String name, Object v) { this.slots.put(name, v); }
  public Object get (String name) { return this.slots.get(name); }
  public void exec (String name) { this.methods.get(name).exec(this); }
  @Override public String toString() { return this.slots.toString(); }
}
```

Example (2/2)

```
public class Main {
  public static void main(String[] args) {
    ProtoObject p = new ProtoObject();
    p.add("name", "Leonard Nimoy");
    p.add("age", 83);
    p.add("incrementAge",
          self -> {
                self.add( "age",
                      ((Integer)self.get("age"))+1);
              }
    );
    p.add("print",
          self -> {
                System.out.println("name: "+self.get("age")+
                "\n"+"age: "+self.get("age")+" years.");
              }
    );
    System.out.println(p);
    p.exec("incrementAge");
    p.exec("print");
    System.out.println(p);
  }
}
```

Output:
{name=Leonard Nimoy, age=83}
name: Leonard Nimoy
age: 84 years.
{name=Leonard Nimoy, age=84}

Exercise

- Modify the previous example so that:
  - An object (the prototype) can be cloned.
  - The created object stores a reference to its prototype
  - If we access an object variable, and that object has not given it a value, or does not have it, the variable is sought in the prototype.
  - Adding a method or variable in the prototype is reflected in its clones, but not the other way round.

Some languages with similar working scheme:

Self: http://en.wikipedia.org/wiki/Self_%28programming_language%29

JavaScript: http://en.wikipedia.org/wiki/JavaScript

Generic functional interfaces

- A functional interface can have generic parameters.
- Example:

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

Generic functional interfaces

```
class Person {
  private String name;
  private int age;

  public Person(String n, int e) { this.name = n; this.age = e; }
  public String toString() { return "name: "+this.name+" age: "+this.age; }
  public int getAge() { return this.age; }
}

public class Compare {
  public static void main(String[] args) {
   List<Persona> list = Arrays.asList(new Person("Leonard Simon Nimoy", 83),
                                      new Person("William Shatner", 84),
                                      new Person("Jackson DeForest", 79));

   Collections.sort(list, (x, y) -> x.getAge() - y.getAge());
   System.out.println(list);
   Collections.sort(list, (x, y) -> y.getAge() - x.getAge());
   System.out.println(list);
  }
}
```

Use of Functional Interfaces

Function and its specializations

```
import java.util.function.*;

public class FunctionExample {
  public static void main(String[] args) {
    // Using Function and its specializations
    Function<Integer, Integer> square = x -> x * x;
    IntFunction<String> toStrn = x -> String.valueOf(x);// From int to String
    ToIntFunction<Float> floor   = x -> Math.round(x);// From float to Integer
    UnaryOperator<Integer> square2 = x -> x * x; // From Integer to Integer
    System.out.println(square.apply(5));
    System.out.println(toStrn.apply(5));
    System.out.println(floor.applyAsInt(5f));
    System.out.println(square2.apply(5));
  }
}
```

Output

```
25
5
5
25
```

Function

Some default and static methods

```
@FunctionalInterface
public interface Function<T, R> {
   R apply(T t);   // The functional method
   default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {
         Objects.requireNonNull(before);
         return (V v) -> apply(before.apply(v));
   }
  default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {
         Objects.requireNonNull(after);
         return (T t) -> after.apply(apply(t));
  }
  static <T> Function <T, T> identity() {
         return t->t;
  }
}
```

Composing Functions

```
public class ComposedFunctions {
  public static void main(String[] args) {
    // Create two functions
    Function<Long, Long> square = x -> x * x;
    Function<Long, Long> addOne = x -> x + 1;
    // Compose functions from the two functions
    Function<Long, Long> squareAddOne = square.andThen(addOne);
    Function<Long, Long> addOneSquare = square.compose(addOne);
    // Get an identity function
    Function<Long, Long> identity = Function.<Long>identity();
    // Test the functions
    long num = 5L;
    System.out.println("Number : " + num);
    System.out.println("Square and then add one: " + squareAddOne.apply(num));
    System.out.println("Add one and then square: " + addOneSquare.apply(num));
    System.out.println("Identity: " + identity.apply(num));
  }
}
```

Output:

```
Number : 5
Square and then add one: 26
Add one and then square: 36
Identity: 5
```

Predicate

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

    default Predicate<T> and(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) && other.test(t);
    }

    default Predicate<T> negate() {
        return (t) -> !test(t);
    }

    default Predicate<T> or(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) || other.test(t);
    }

    static <T> Predicate<T> isEqual(Object targetRef) {
        return (null == targetRef) ? Objects::isNull  : object -> targetRef.equals(object);
    }
}
```

Predicate

```java
public class Predicates {
    public static void main(String[] args) {
        // Create some predicates
        Predicate<Integer> greaterThanTen = x -> x > 10;
        Predicate<Integer> divisibleByThree = x -> x % 3 == 0;
        Predicate<Integer> divisibleByFive = x -> x % 5 == 0;
        Predicate<Integer> equalToTen = Predicate.isEqual(null);
        // Create predicates using NOT, AND, and OR on other predicates
        Predicate<Integer> lessThanOrEqualToTen=greaterThanTen.negate();
        Predicate<Integer> divisibleByThreeAndFive=divisibleByThree.and(divisibleByFive);
        Predicate<Integer> divisibleByThreeOrFive=divisibleByThree.or(divisibleByFive);
        // Test the predicates
        int num = 10;
        System.out.println("Number: " + num);
        System.out.println("greaterThanTen: " + greaterThanTen.test(num));
        System.out.println("divisibleByThree: " + divisibleByThree.test(num));
        System.out.println("divisibleByFive: " + divisibleByFive.test(num));
        System.out.println("lessThanOrEqualToTen: " + lessThanOrEqualToTen.test(num));
        System.out.println("divisibleByThreeAndFive: " +
                           divisibleByThreeAndFive.test(num));
        System.out.println("divisibleByThreeOrFive: " +
                           divisibleByThreeOrFive.test(num));
        System.out.println("equalsToTen: " + equalToTen.test(num));
    }
}
```

Exercise (1/2)

- Using lambdas, create a simulator for state machines.
- A state machine is made of states and a series of variables, which we assume to be of the Integer type.
- The machine transitions from one state to another one when an event (of type String) occurs.
- States can have associated actions, which are executed when exiting the state due to some event, and can for example modify variables.

Exercise (2/2)

```
public class Main {
  public static void main(String[] args) {
    StateMachine sm = new StateMachine("Light", "num"); // name y variable
    State s1 = new State("off");
    State s2 = new State("on");
    s1.addEvent("switch", s2);
    s2.addEvent("switch", s1);
    s1.action((State s, String e) -> s.set("num", s.get("num")+1) );
    s2.action((State s, String e) -> s.set("num", s.get("num")+1) );

    sm.addStates(s1, s2);
    sm.setInitial(s1);

    System.out.println(sm);
    MachineSimulator ms = new MachineSimulator(sm);
    ms.simulate(Arrays.asList("switch", "switch"));
   }
}
```

```
Output:

Machine Light : [off, on]
switch: from [off] to [on]
 Machine variables: {num=1}
switch: from [on] to [off]
 Machine variables: {num=2}
```

Index

- New concepts in interfaces
- Lambda expressions
- Exercises
- **Conclusions and bibliography**

Conclusions

- Lambda expressions introduce flexibility and conciseness in specifying operations with collections of elements
- Other advantages, such as ease of parallelization

(but this year we could not cover streams)

- We have **NOT** covered other advanced aspects:
  - The API of functional interfaces and streams is extensive
    - Part of this API will be explored and practiced in the lab assignment.
    - The functional paradigm (lisp) will be studied in more detail in the Artificial Intelligence course
  - Design of embedded domain specific languages:
    - In Java using lambdas: (http://en.wikipedia.org/wiki/Domain-specific_language)
    - The design of languages embedded in Ruby will be studied in the optional course automated software development.

Bibliography

- Java 8 Lambdas. Functional Programming for the masses. O’Reilly. Richard Warburton. 2014.

- Beginning Java 8 Language Features. Kishori Sharan. Apress. 2014.

- Functional Programming in Java. Harnessing the power of Java 8 Lambda Expressions. The Pragmatic Programmers. V. Subrmanian. 2014.