Functional interfaces

MBF

Examples

```
Comparator<Person> byAge =
                    Comparator.comparing(Person::age);

Predicate<Person> isAdult = p -> p.age() >= 18;
Predicate<Person> isYoung = p -> p.age() <=26;
Predicate<Person> isAdultAndYoung =
                                isAdult.and(isYoung);
```

```
Person p1 = new Person("John", 16);
Person p2 = new Person("Bob", 20);
Person p3 = new Person("Jane", 40);
System.out.println (p1 + " is adult : " + isAdult.test(p1));

System.out.println ((byAge.compare(p1, p2) > 1) ?
      p1 + " is older than " + p2 :
      p2 + " is older than " + p1);

System.out.println (p1 + " is adult but young  " +
isAdultAndYoung.test(p1));
System.out.println (p2 + " is adult but young  " +
isAdultAndYoung.test(p2));
System.out.println (p3 + " is adult but young  " +
isAdultAndYoung.test(p3));
```

```
Person[John, 16] is adult : false
Person[Bob, 20] is older than Person[John, 16]
Person[John, 16] is adult but young  false
Person[Bob, 20] is adult but young  true
Person[Jane, 40] is adult but young  false
```

Examples

```
Function<Integer, String> intToDollar = i -> "$" + i;      //Integer in input, String as output
Function<String, String> quote = s -> "'" + s + "'";

Function<Integer, String> quoteIntToDollar = quote.compose(intToDollar);
System.out.println( quoteIntToDollar.apply(5));
```

```
'$5'
```

Functional interface

- A functional interface is an interface with a single abstract method (and vice versa)
- A functional interface represents a type of function
- To identify these functions, we use the @FunctionalInterface annotation
- You can use a lambda ((parameters) -> code) where a functional interface is expected.

https://www-igm.univ-mlv.fr/~forax/ens/java-avance/cours/pdf/2-lambda.pdf

Functional interface

- Une interface fonctionnelle est une interface avec une seule méthode abstraite (et inversement)
- Une interface fonctionnelle représente un type de fonction
- Pour identifier ces fonctions on utilise l'annotation @FunctionalInterface
- On peut utiliser une lambda ((paramètres) -> code) là une interface fonctionnelle est attendue.

@FunctionalInterface

```
@FunctionalInterface
public interface BiFunction<T,U,V> {
    public V apply(T t, U u); }
```

- The compiler checks that there is only one abstract method.
  - Static or default methods do not matter
- Documents (javadoc) the interface as a type of function

Functional interface vs Function Type

Problem

- Does this mean that I must create an interface every time I need to define a function type?

Yes, but

- The JDK already provides a set of functional interfaces for the most common function types in the java.util.function package

Functions by examples

Example : **Predicate**

```
@FunctionalInterface
public interface Predicate<T> {

    /**
     * Evaluates this predicate on the given argument.
     *
     * @param t the input argument
     * @return {@code true} if the input argument matches the predicate,
     * otherwise {@code false}
     */
    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);
    }

    @SuppressWarnings("unchecked")
    static <T> Predicate<T> not(Predicate<? super T> target) {
        Objects.requireNonNull(target);
        return (Predicate<T>)target.negate();
    }
}
```

Yes. There is still only one abstract method

Example : Comparator

```
@FunctionalInterface
public interface Comparator<T> {

    int compare(T o1, T o2);


    boolean equals(Object obj);

    default Comparator<T> reversed() {
        return Collections.reverseOrder(this);
    }


    default Comparator<T> thenComparing(Comparator<? super T> other) {
        Objects.requireNonNull(other);
        return (Comparator<T> & Serializable) (c1, c2) -> {
            int res = compare(c1, c2);
            return (res != 0) ? res : other.compare(c1, c2);
        };
    }

....

    public static <T> Comparator<T> comparingLong(ToLongFunction<? super T>
keyExtractor) {
        Objects.requireNonNull(keyExtractor);
        return (Comparator<T> & Serializable)
            (c1, c2) -> Long.compare(keyExtractor.applyAsLong(c1),
keyExtractor.applyAsLong(c2));
    }


    public static<T> Comparator<T> comparingDouble(ToDoubleFunction<? super T>
keyExtractor) {
        Objects.requireNonNull(keyExtractor);
        return (Comparator<T> & Serializable)
            (c1, c2) -> Double.compare(keyExtractor.applyAsDouble(c1),
keyExtractor.applyAsDouble(c2));
    }
}
```

```
public static <T, U extends Comparable<? super U>>
Comparator<T> comparing(
        Function<? super T, ? extends U> keyExtractor)
{
    Objects.requireNonNull(keyExtractor);
    return (Comparator<T> & Serializable)
        (c1, c2) ->
keyExtractor.apply(c1).compareTo(keyExtractor.apply(c2));
}
```

The equals(Object) method is implicit from the Object class => There is still only one abstract method

Comparator & Predicate in action

```
Comparator<Person> byAge = Comparator.comparing(Person::age);

Predicate<Person> isAdult = p -> p.age() >= 18;
```

```
    Person p1 = new Person("John", 16);
    Person p2 = new Person("Bob", 20);
    System.out.println (p1 + " is adult : " + isAdult.test(p1));
    System.out.println ((byAge.compare(p1, p2) > 1) ?
            p1 + " is older than " + p2 :
            p2 + " is older than " + p1);
}
```

MAP and use of Function

```
public interface Map<K, V> {

default V computeIfAbsent(K key,
        Function<? super K, ? extends V> mappingFunction) {
            ....
```

La fonction prend en paramètre un objet du type de la clef (K)
Et elle retourne un objet du type des valeurs (V)

```
nameMap.computeIfAbsent("Fred", Person::new);
nameMap.computeIfAbsent("Pierre", (k) -> new Person(k) );
```

La fonction prend en paramètre un objet du type de la clef (K) et un objet du type des valeurs (V)
Et elle retourne un objet du type des valeurs (V)

```
default V computeIfPresent(K key,
        BiFunction<? super K, ? super V, ? extends V> remappingFunction) {
```

```
nameMap.computeIfPresent("Bob", (k, v) -> new Person(k, v.age() + 1));
```

A map and computeIfAbsent or Present : function in action

```
Map<String, Person> nameMap = new HashMap<>();
nameMap.put("John", new Person("John", 20));
nameMap.put("Bob", new Person("Bob", 30));
nameMap.put("Alice", new Person("Alice", 25));
nameMap.put("Alice", new Person("Alice", 25));
System.out.println("Before : " + nameMap);
nameMap.computeIfAbsent("Fred", Person::new);
System.out.println("Add Fred : " + nameMap);
nameMap.computeIfAbsent("Bob", Person::new);
System.out.println("Do nothing : " + nameMap);
//Person are Records, so they are immutable
//Making a new Person with the same name and age + 1
nameMap.computeIfPresent("Bob", (k, v) -> new
Person(k, v.age() + 1));
System.out.println(" Bob is older : " + nameMap);
nameMap.computeIfPresent("Lucile", (k, v) -> new
Person(k, v.age() + 1));
System.out.println("Don't fail : " + nameMap);
```

Before : {Bob=Person[Bob, 30], Alice=Person[Alice, 25], John=Person[John, 20]}

Add Fred : {Bob=Person[Bob, 30], Alice=Person[Alice, 25], John=Person[John, 20], Fred=Person[Fred, -1]}

Do nothing : {Bob=Person[Bob, 30], Alice=Person[Alice, 25], John=Person[John, 20], Fred=Person[Fred, -1]}

Bob is older : {Bob=Person[Bob, 31], Alice=Person[Alice, 25], John=Person[John, 20], Fred=Person[Fred, -1]}

Don't fail : {Bob=Person[Bob, 31], Alice=Person[Alice, 25], John=Person[John, 20], Fred=Person[Fred, -1]}

MAP and use of Function : replaceAll and Merge

```
Map<String, Integer> salaries = new HashMap<>();
salaries.put("John", 10000);
salaries.put("Freddy", 10000);
salaries.put("Samuel", 10000);
System.out.println("All the same salaries " + salaries);
//The replaceAll method is used to replace all the values of the map with the new value computed by the given function.
// The salary of all employees is increased by 10,000, except for Freddy.
salaries.replaceAll((name, oldValue) ->
      name.equals("Freddy") ? oldValue : oldValue + 20000);
System.out.println("only Freddy's salary has not changed : " + salaries);

salaries.replaceAll((name, oldValue) ->
      oldValue<30000? oldValue + 10000: oldValue );
System.out.println("only Freddy's salary changed : " + salaries);

//The merge method takes three parameters: the key, the value to be merged, and a BiFunction to compute the new value if the
key is already present. If the key is not present, it simply puts the specified value.
for (Map.Entry<String, Integer> entry : salaries.entrySet())
   salaries.merge(entry.getKey(), 10000, Integer::sum);
System.out.println("an increase for all! " + salaries);
```

```
All the same salaries {Samuel=10000, John=10000, Freddy=10000}
only Freddy's salary has not changed : {Samuel=30000, John=30000, Freddy=10000}
only Freddy's salary changed : {Samuel=30000, John=30000, Freddy=20000}
an increase for all! {Samuel=40000, John=40000, Freddy=30000}
finish : {Samuel=50000, Lucile=10000, John=50000, Freddy=40000}
```

Function composition

```
Function<Integer, String> intToDollar = i -> "$" + i;
Function<String, String> quote = s -> "'" + s + "'";
Function<Integer, String> quoteIntToDollar = quote.compose(intToDollar);
```

```
System.out.println( quoteIntToDollar.apply(5));
```

```
'$5'
```

```
Predicate<Person> isAdult = p -> p.age() >= 18;
Predicate<Person> isYoung = p -> p.age() <=26;
Predicate<Person> isAdultAndYoung = isAdult.and(isYoung);
```

```
Person p1 = new Person("John", 16);
Person p2 = new Person("Bob", 20);
Person p3 = new Person("Jane", 40);
System.out.println (p1 + " is adult but young  " + isAdultAndYoung.test(p1));
System.out.println (p2 + " is adult but young  " + isAdultAndYoung.test(p2));
System.out.println (p3 + " is adult but young  " + isAdultAndYoung.test(p3));
```

```
Person[John, 16] is adult but young  false
Person[Bob, 20] is adult but young  true
Person[Jane, 40] is adult but young  false
```

https://www.baeldung.com/java-8-functional-interfaces

Package java.util.function

- Types de fonction
  - de 0 à 2 paramètres
    - Runnable, Supplier, Consumer, **Function**, **BiFunction**, ...
  - spécialisé pour les types primitifs
    - IntSupplier () → int,
    - LongSupplier () → long,
    - DoubleSupplier () → double
    - **Predicate** (T) → boolean
    - IntFunction (int) → T
    - ToIntFunction (T) → int
  - spécialisé si même type en paramètre et type de retour
    - UnaryOperator (T) → T
    - BinaryOperator (T, T) → T ●
    - DoubleBinaryOperator (double, double) → double, ...

Runnable

```
@FunctionalInterface
public interface Runnable {
   public abstract void run();
}
```

- java.lang.Runnable est équivalent à () → void

```
Runnable code = () -> { System.out.println("hello"); };
…
code.run();
```

https://medium.com/swlh/understanding-java-8s-consumer-supplier-predicate-and-function-c1889b9423d

Supplier

```
public interface Supplier<T> {
        T get();
}
```

- A Supplier indicates that this implementation is a supplier of results.
- The supplier has only one method get().
- One of the primary usage of this interface is to enable deferred execution.

```
Supplier<Double> doubleSupplier1 = () -> Math.random();
DoubleSupplier doubleSupplier2 = Math::random;
System.out.println(doubleSupplier1.get());
System.out.println(doubleSupplier1.get());
System.out.println(doubleSupplier2.getAsDouble());
```

Supplier usage

```
Supplier<Double> doubleSupplier = () -> Math.random();
Optional<Double> optionalDouble = Optional.empty();
System.out.println(optionalDouble.orElseGet(doubleSupplier));
```

https://medium.com/swlh/understanding-java-8s-consumer-supplier-predicate-and-function-c1889b9423d

Supplier

```
Supplier<Integer> fib = new Supplier<Integer>() {
    private int previous = 0;
    private int current = 1;
    @Override
    public Integer get() {
        int nextValue = this.previous + this.current;
        this.previous = this.current;
        this.current = nextValue;
        return this.previous;
    }
};
```

```
Optional<Integer> optionalDouble = Optional.empty();
for (int i = 0; i < 10; i++) {
    System.out.println("Opt : " + optionalDouble.orElseGet(fib));
}
```

```
Opt : 1
Opt : 1
Opt : 2
Opt : 3
Opt : 5
Opt : 8
Opt : 13
Opt : 21
Opt : 34
Opt : 55
```

https://medium.com/swlh/understanding-java-8s-consumer-supplier-predicate-and-function-c1889b9423d

BinaryOperator

```
public interface BiFunction<T, U, R> {
R apply(T t, U u);

public interface BinaryOperator<T>
extends BiFunction<T,T,T> {
```

```
BinaryOperator<Integer> add = (a, b) -> a + b;
BinaryOperator<Integer> multiply = (a, b) -> a * b;
System.out.println(add.apply(3, 2));
System.out.println(multiply.apply(4, 2));

BinaryOperator<Integer> maxBy =
BinaryOperator.maxBy(Comparator.naturalOrder());
System.out.println(maxBy.apply(2, 1));
```

```
5
8
2
```

https://medium.com/swlh/understanding-java-8s-consumer-supplier-predicate-and-function-c1889b9423d

A Consumer accepts a single input and returns no output

```
@FunctionalInterface
public interface Consumer<T> {
   void accept(T t);
```

```
Consumer<String> printConsumer = t -> System.out.println(t);
Stream<String> cities = Stream.of("Sydney", "Dhaka", "New York", "London");
        cities.forEach(printConsumer);
```

forEach takes a Consumer as a parameter

```
List<String> cities = Arrays.asList("Sydney", "Dhaka", "New York", "London");
Consumer<List<String>> upperCaseConsumer = list -> {
   for(int i=0; i< list.size(); i++){
      list.set(i, list.get(i).toUpperCase());
   } };
Consumer<List<String>> printConsumer =
      list -> list.stream().forEach(System.out::println);
upperCaseConsumer.andThen(printConsumer).accept(cities);
```

https://medium.com/swlh/understanding-java-8s-consumer-supplier-predicate-and-function-c1889b9423d

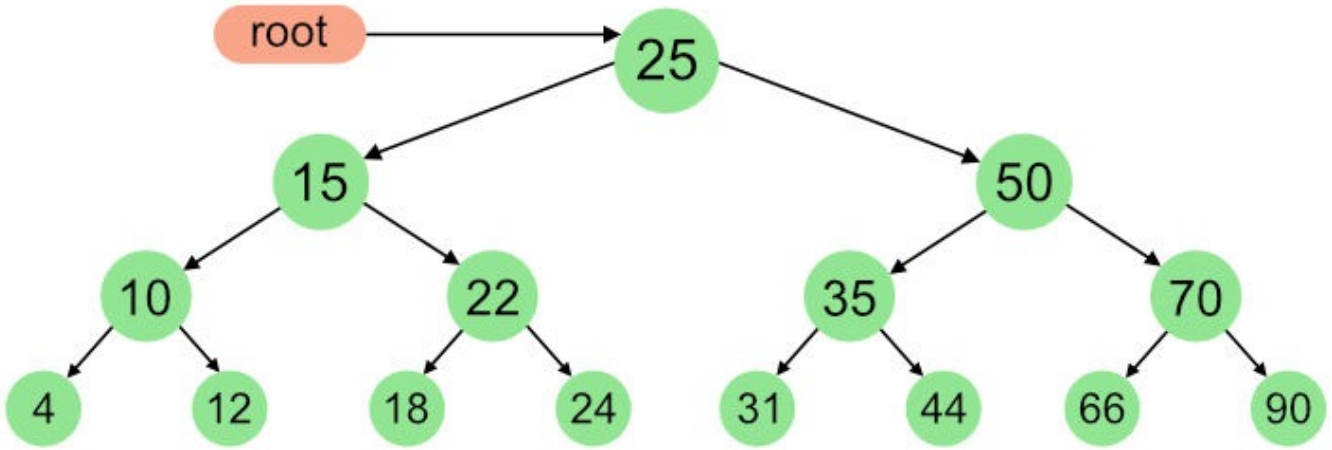
Functions and Trees : Poo in action

Remember : Tree Traversals

InOrder(root) visits nodes in the following order:
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:
25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



Applying a function on each node

```
• public static <T, R> void  preOrderTraversal(
                                        BinaryTreeInterface<T> root,
                                        Function<T, R> fn,
                                        List<R> toCollect) {
    if (root == null) {
        return ;
    }
    toCollect.add(fn.apply(root.data()));
    preOrderTraversal(root.left(), fn, toCollect);
    preOrderTraversal(root.right(), fn, toCollect);
    toCollect.removeIf(Objects::isNull);
}
```

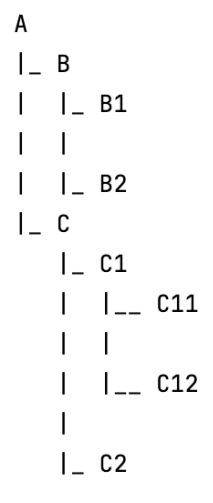
Applying a function on each node

```
public static <T, R> void postOrderTraversal(
        BinaryTreeInterface<T> root,
        Function<T, R> fn,  List<R> toCollect) {

   if (root == null) {
      return ;
   }

   postOrderTraversal(root.left(), fn, toCollect);
   postOrderTraversal(root.right(), fn, toCollect);
   toCollect.add(fn.apply(root.data()));
   toCollect.removeIf(Objects::isNull);
}
```

Applying a function on each node : usage

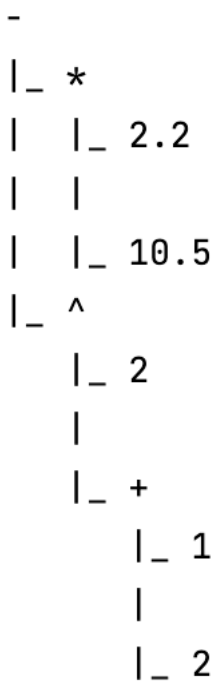
```
BinaryTreeInterface<String> bt = BinaryTree.read("A B B1$ B2$ C C1 C11$ C12$ C2$");
List<String> names = new ArrayList<>();
postOrderTraversal(bt, String::toLowerCase, names);
```

[b1, b2, b, c11, c12, c1, c2, c, a]




```
ExpressionTree e = ExpressionTree.read("- * 2.2$ 10.5$ ^ 2$ + 1$ 2$");
List<Double> numbers = new ArrayList<>();
Function<String, Double> extractNumber = s -> {
    try {
        return Double.parseDouble(s);
    } catch (NumberFormatException nfe) {
        return null;
    }
};
preOrderTraversal(e, extractNumber, numbers);
```

[2.2, 10.5, 2.0, 1.0, 2.0]

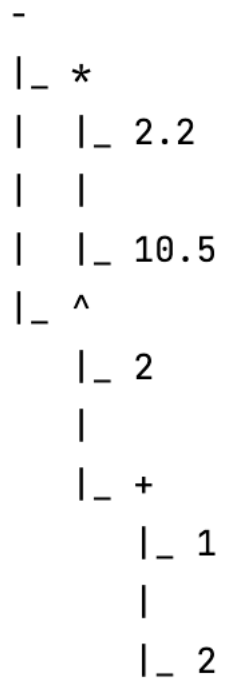


Applying a function on each node : usage

```
ExpressionTree e = ExpressionTree.read("- * 2.2$ 10.5$ ^ 2$ + 1$ 2$");
int i = 4;
List<Integer> ints = new ArrayList<>();
preOrderTraversal(e,
        s -> {
            Integer n = Traversals.toInteger(s);
            if (n == null)
                return null;
            if (n < i)
                return n;
            else
                return null;
        },
        ints);
System.out.println("only less than 4 :" + ints);
```

[2, 2, 1, 2]

```
public static Integer toInteger(String s) {
  try {
    return (int) Double.parseDouble(s);
  } catch (NumberFormatException nfe) {
    return null;
  }
```



Expression Tree extends BinaryTree<String>{

```
    private static final Map<String, BiFunction<Double, Double, Double>>
    OPERATORS =
          Map.of(
                "+", (a, b) -> a + b,
                "-", (a, b) -> a - b,
                "*", (a, b) -> a * b,
                "/", (a, b) -> a / b,
                "^", (a, b) -> Math.pow(a,b)
          );

private BiFunction<Double,Double,Double> operation;

public double evaluate() {
   if (operation != null) {
      return operation.apply(
            ((ExpressionTreeWithFunction)left()).evaluate(),
            ((ExpressionTreeWithFunction)right()).evaluate());
   } else {
      return     Double.parseDouble(data());
   }
```

```
-
|_ *
|  |_ 2.2
|  |
|  |_ 10.5
|_ ^
   |_ 2
   |
   |_ +
      |_ 1
      |
      |_ 2
```