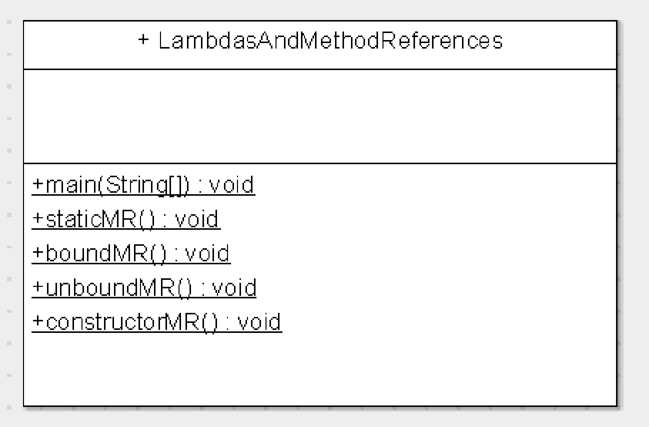

Lambdas and Method References Exercises

1. Static method references:
    - a. in *staticMR()*, declare a *List* of integers with 1, 2, 7, 4, and 5 as values.
    - b. using a *Consumer* typed for *List<Integer>* and the *Collections.sort* static method, code a lambda that sorts the list passed in.
    - c. invoke the lambda.
    - d. prove that the sort worked.
    - e. re-initialise the list (so it is unsorted again).
    - f. code the method reference version.
        - i. *sort()* is overloaded : *sort(List)* and *sort(List, Comparator)*
        - ii. how does Java know which version to call?
    - g. invoke the method reference version.
    - h. prove that the sort worked.
2. Bound method references (calling instance methods on a particular object):
    - a. in *boundMR()*, declare a *String* variable called *name* and initialise it to “Mr. Joe Bloggs”.
    - b. using a *Predicate* typed for *String*, code a lambda that checks to see if name starts with the prefix passed in.
    - c. invoke the lambda passing in “Mr.” which should return true.
    - d. invoke the lambda passing in “Ms.” which should return false.
    - e. code the method reference version.
    - f. repeat c and d above except using the method reference version.
3. Unbound method references (calling instance methods on a parameter):
    - a. in *unboundMR()*, code a *Predicate* lambda typed for *String* that checks to see if the string passed in is empty.
    - b. invoke the lambda passing in “” (returns true).
    - c. invoke the lambda passing in “xyz” (returns false).
    - d. code the method reference version of the lambda from (a).

e. repeat b and c above except using the method reference version.

f. code a *BiPredicate* lambda typed for *String* and *String*:

   i. the lambda takes in two parameters (hence "Bi")

   ii. check if the first parameter starts with the second parameter

   iii. invoke the lambda twice:

      1. passing in "Mr. Joe Bloggs" and "Mr." (returns true)
      2. passing in "Mr. Joe Bloggs" and "Ms." (returns false)

g. code the method reference version of the lambda from (f).

h. test it as per above in (f.iii)

4. Constructor method references:

   a. in *constructorMR()*, code a *Supplier* typed for *List<String>* that returns a *new ArrayList*.

   b. invoke the lambda to create a new List<String> named list.

   c. add "Lambda" to the list.

   d. output the list to show it worked.

   e. code the method reference version of the lambda:

      i. re-initialise list by invoking the method reference version.

      ii. add "Method Reference" to the list.

      iii. output the list to show it worked.

   f. next, we want to use the overloaded ArrayList constructor passing in 10 as the initial capacity (note: the default constructor assumes a capacity of 10).

      i. thus, we need to pass IN something and get back OUT something:

         1. IN: 10   OUT: ArrayList

      ii. we need a Function typed for Integer and List<String> for this.

      iii. code the lambda.

      iv. re-initialise the list by invoking the lambda passing in 10 as the capacity.

      v. add "Lambda" to the list.

      vi. output the list to show it worked.

   g. code the method reference version.

      i. note that the method reference version is the **exact same** as above in e!!

      ii. this is where **context** is all important:

         1. the first method reference was for a Supplier and Supplier's functional method is T get() and thus, Java knew to look for the ArrayList constructor that takes in NO argument
         2. the first method reference was for a Function and Function's functional method is R apply(T t) and thus, Java knew to look for the ArrayList constructor that takes in ONE argument.

Type	Solution

static method references	Consumer<List<Integer>> lambda = x -> Collections.sort(x);
	lambda.accept(list);
	
	Consumer<List<Integer>> methodRef = Collections::sort;
	methodRef.accept(list);

bound method references
(calling instance methods on a particular object)
	String name = "Mr. Joe Bloggs";
	Predicate<String> lambda = prefix -> name.startsWith(prefix);
	System.out.println(lambda.test("Mr."));// true
	
	Predicate<String> methodRef = name::startsWith;
	System.out.println(methodRef.test("Ms."));// false

unbound method references
(calling instance methods on a parameter)
	Predicate<String> lambda = str -> str.isEmpty();
	System.out.println(lambda.test("")); // true "".isEmpty();
	Predicate<String> methodRef = String::isEmpty;
	System.out.println(methodRef.test("xyz")); // false "xyz".isEmpty();
	
	BiPredicate<String, String> lambda2 = (str, prefix) -> str.startsWith(prefix);
	System.out.println(lambda2.test("Mr. Joe Bloggs", "Mr.")); // true
	BiPredicate<String, String> methodRef2 = String::startsWith;
	System.out.println(methodRef2.test("Mr. Joe Bloggs", "Ms.")); // false
	// "Mr. Joe Bloggs".startsWith("Ms.")

constructor method references	Supplier<List<String>> lambda = () -> new ArrayList();
	List<String> list = lambda.get();
	Supplier<List<String>> methodRef = ArrayList::new;
	list = methodRef.get();
	
	Function<Integer, List<String>> lambda2 = n -> new ArrayList(n);
	list = lambda2.apply(20);
	Function<Integer, List<String>> methodRef2 = ArrayList::new; // context!
	list = methodRef2.apply(20);