**Type inference**

The java compiler decides what functional interface to be associated to the lambda expression from its surrounding context (the target type).

So type can be inferred when defining a lambda for better readability as compiler can know which type of parameter it is from the context object/target type (is a functional interface).

*Comparator<Apple> c = (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());*

Type can be inferred as

*Comparator<Apple> c = (a1, a2) -> a1.getWeight().compareTo(a2.getWeight());*

**Using local variables**

Lambdas can capture instance and static variables inside their body. But local variables should be explicit declared final or effective final inside their body.

*Ex.*

*Int portNumber = 1337;*

*Runnable r = () -> sysout(portNumber); // local variable portNumber should be final or effective final*

*portNumber = 8080;*

**Method references**

Method reference is a shorthand for lambdas calling only a specific method, the basic idea before it is if a method is called directly then why to provide a description on how to call the method. Its best to call by method name.

*Ex:*

*Inventory.sort((Apple a1, Apple a2)*

* *a1.getWeight().compareTo(a2.getWeight()));*

After using a method reference and java.util.Comparator.comparator

*Inventory.sort(comparing(****Apple::getWeight****));*

|  |  |
| --- | --- |
| (Apple a) -> a.getWeight() | Apple::getWeight |
| O -> Thread.currentThread().dumpStack() | Thread.currentThread()::dumpStack() |
| (str,i) -> str.subString(i) | String::substring |
| (String s) -> sysout(s) | System.out::println |

**Reversed Order**

With an example if we want to sort Apple by decreasing order of their weight

*Inventory.sort(Comparator.comparing(Apple::getWeight).reversed());*

**Chaining Comparators:**

With example if we find two Apples with same weight then sort it by the country where it got imported.

*Inventory.sort(comparing(Apple::getWeight).revesed().thenComparing(Apple::getCountry));*

**Composing Predicates**

The Predicate interface includes 3 more methods that lets us reuse an existing predicate to create a more complicated one. **Negate, and, or**

*EX: Predicate of apples which are not red*

*Prediccate<Apple> notRedApple = redApple.negate();*

*Ex: Combine two lambdas to say apple is both red and heavy*

*Predicate<Apple> appleRedHeavy = redApple.and(a -> a.getWeight() > 150);*

*Ex. Apples red and heavy or just green apple*

*Predicate<Apple> appleRedHeavyOrJustGreen = redApple.and(a -> a.getWeight() > 150)*

*.or(a -> “green”.equals(a.getColor()));*

**Composing Functions**

We can compose lambda expression represented by the Function interface

The Function interface comes with 2 default method for this **andThen** and **compose.**

*Function<Integer, Integer> f = x -> x+1*

*Function<Integer, Integer> g = x -> x\*1*

*Function<Integer, Integer> h = f.andThen(g);*

*h.apply(1); // 4*

**compose** will produce the result as f(g(x)), in the previous example

Function<Integer, Integer> f = x -> x+1;

Function<Integer, Integer> g = x -> x\*1;

Function<Integer, Integer> h = f.compose(g); // f(g(x))

h.apply(1); // **3**