**Functional Interface-**This kind of interfaces are also known as SAM (Single Abstract Method) types.

An interface with exactly one abstract method becomes Functional Interface. We don’t need to use @FunctionalInterface annotation to mark an interface as Functional Interface. @FunctionalInterface annotation is a facility to avoid accidental addition of abstract methods in the functional interfaces. You can think of it like [@Override annotation](http://www.journaldev.com/817/overriding-methods-in-java-always-use-override-annotation) and it’s best practice to use it. java.lang.Runnable,with single abstract method run() is a great example of functional interface. A new package java.util.function with bunch of functional interfaces are added to provide target types for lambda expressions and method references.

For example, take the ITrade functional interface. It has only one abstract method that takes a Trade object and returns a boolean value – perhaps checking the status of the trade or validating the order or some other condition.

@FunctionalInterface

public interface ITrade {

public boolean check(Trade t);

}

In order to satisfy our requirement of checking for *new* trades, we could create a lambda expression, using the above functional interface, as shown here:

ITrade newTradeChecker = (Trade t) -> t.getStatus().equals("NEW");

// Or we could omit the input type setting:

ITrade newTradeChecker = (t) -> t.getStatus().equals("NEW");

Example for finding out big trade (i.e, if the trade’s quantity is greater than 1 million) or checking out a newly created large Google trade:

// Lambda for big trade

ITrade bigTradeLambda = (Trade t) -> t.getQuantity() > 10000000;

// Lambda that checks if the trade is a new large google trade

ITrade issuerBigNewTradeLambda = (t) -> {

return t.getIssuer().equals("GOOG") &&

t.getQuantity() > 10000000 &&

t.getStatus().equals("NEW");

};

These functions can then be passed on to a method (most probably server side) which takes in an ITrade as one of its parameters. Let’s say, we have a collection of trades and wish to filter out some trades based on a certain criteria. This requirement can be easily expressed using the above lambda passing to a method which accepts a list of trades too:

/ Method that takes in list of trades and applies the lambda behavior for each of the trade in the collection

private List<Trade> filterTrades(ITrade tradeLambda, List<Trade> trades) {

List<Trade> newTrades = new ArrayList<>();

for (Trade trade : trades) {

if (tradeLambda.check(trade)) {

newTrades.add(trade);

}

}

return newTrades;

}

The above filterTrades method expects a collection trade and a lambda to be applied on each of the trades in the collection one by one, by iterating over them. If the input trade satisfies the pre-conditions defined by the lambda, the trade is added to the cumulative basket, else thrown away. The good thing about this method is that it accepts any lambda as long as it implements an ITrade interface. The method is shown below by taking the various lambdas:

// Big trades function is passed

List<Trade> bigTrades =

client.filterTrades(bigTradeLambda,tradesCollection);

// "BIG+NEW+ISSUER" function is passed

List<Trade> bigNewIssuerTrades =

client.filterTrades(issuerBigNewTradeLambda,tradesCollection);

// cancelled trades function is passed

List<Trade> bigNewIssuerTrades =

client.filterTrades(cancelledTradesLambda,tradesCollection);

**From Lambdas to Double Colon Operator**

With Lambdas expressions we’ve seen that code can become very concise.

For example, to **create a comparator**, the following syntax is enough:

Comparator c = (Computer c1, Computer c2) -> c1.getAge().compareTo(c2.getAge());

But can we make the code above even more expressive and readable? Let’s have a look:

Comparator c = Comparator.comparing(Computer::getAge);

We’ve used the :: operator as shorthand for lambdas calling a specific method – by name. And the end result is of course even more readable syntax.

## ****How Does It Work****

Very simply put, when we are using a method reference – the target reference is placed before the delimiter *::* and the name of the method is provided after it.

For example:

|  |  |
| --- | --- |
| 1 | Computer::getAge; |

We’re looking at a method reference to the method *getAge* defined in the *Computer* class.

We can then operate with that function:

|  |  |
| --- | --- |
| 1  2 | Function<Computer, Integer> getAge = Computer::getAge;  Integer computerAge = getAge.apply(c1); |

Notice that we’re referencing the function – and then applying it to the right kind of argument.