Function Functional Interface

Function is a Functional interface that applies function on any input type *T* and returns the result *R*. We apply function on given argument using *apply()* method. There are also ways to compose functions using *compose()* and *andThen()* method.

Let us first under *apply()* method.

|  |
| --- |
| **apply() method**  R apply(T t);  Apply method applies this function to given argument *T* and returns result *R*.  Let us write a simple apply method that accepts an Integer argument and convert it to String and return it.  Function<Integer, String> convertToString = val -> String.*valueOf*(val);  String result = convertToString.apply(10); //Outputs 10 |
| **compose() method**  **default** <V> Function<V, R> compose(Function<? **super** V, ? **extends** T> before) {  Objects.*requireNonNull*(before);  **return** (V v) -> apply(before.apply(v));  }  compose() method is used to return composed function that first applies to *before* function and then applies *this* function. *So Result of before function becomes an input for this function.*  If after is null then it will throw NullPOinterException.  *V* is the type of input to *this* function and returns the result of type *T*.  Now the returned result of type *T* is applied as input to this function which returns result *R*.  Read the comment to code carefully.  Function<Integer, String> convertToString = val -> String.*valueOf*(val);  // String result = convertToString.apply(10);  Function<String, Integer> convertToInteger = str -> Integer.*valueOf*(str);  // int resultInInt = convertToInteger.apply("100");  /\*\*  \* *this* is convertToString  \* *before* is convertToInteger  \*  \* before.apply(v) will get executed first which returns Integer.  \* Secondly, apply(v) will get executed with input Integer  \* and output String.  \* \*/  Function<String, String> funct = convertToString.compose(convertToInteger);  System.***out***.println(funct.apply("56")); |
| **andThen() method**  **default** <V> Function<T, V> andThen(Function<? **super** R, ? **extends** V> after) {  Objects.*requireNonNull*(after);  **return** (T t) -> after.apply(apply(t));  }  andThen() method returns a composed function that first applies *this* function to its input and then applies the *after* function to *result*.  If *after* is null method will throw NullPointerException.  So first this function will get execute with input type *T* and output is of type *R.*  The output *R* is then taken as input for *after* function and generates output of type *V.*  So compose and andThen() does the opposite tasks.  Function<Integer, String> convertToString = val -> String.*valueOf*(val);  // String result = convertToString.apply(10);  Function<String, Integer> convertToInteger = str -> Integer.*valueOf*(str);  // int resultInInt = convertToInteger.apply("100");  Function<Integer, Integer> funct = convertToString.andThen(convertToInteger);  System.***out***.println(funct.apply(44)); |
| **identity() method**  identity() is a static method that returns a function that always returns its input arguments.  **static** <T> Function<T, T> identity() {  **return** t -> t;  }  See the type arguments all of them are of type *T.*  Function<Integer, Integer> identity = Function.*identity*();  System.***out***.println(identity.apply(30)); // Outputs 30 |

That’s all on Function Interface.