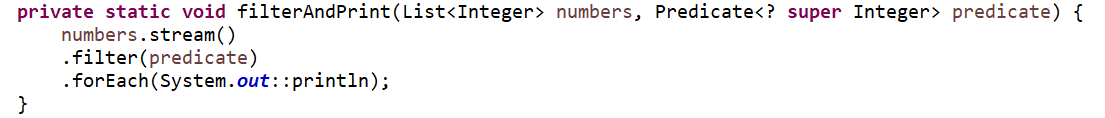
1. Introduction to Functional Programming with Java
   1. **Method Reference**
      1. numbers.stream().forEach(**FP01Functional::print**);
      2. Here stream() provide each element in the list
      3. forEach() run print function for each element
      4. FP01Functional has static print() method which we are passing as method reference.
      5. Method can be static or normal
   2. Functional Program with **filter()**
      1. numbers.stream().**filter(FP01Functional::isEven).**forEach(System.out::println);
      2. pass number if condition is true
   3. Using **Lambda Expression** to enhance Functional Program
      1. numbers.stream().filter(**number -> number % 2 == 0**).forEach(System.out::println);
      2. input -> check condition and return the value
   4. Functional Program with **map()**
      1. numbers.stream().filter(number -> number % 2 == 0).**map(number -> number \* number)**.forEach(System.out::println);
      2. map number with number \* number and pass it.
2. Stream API
   1. Using reduce() operation
      1. Take stream of number and combine them into single value
      2. Use to calculate sum, avg, min, max etc.
      3. numbers.stream().**reduce(0, FP02Functional::sum);**
      4. Here 0 is first aggregate value
   2. Streams Operation – distinct and sorted
      1. The distinct() : return all unique elements
      2. The sorted() : return sorted list
      3. Use comparators to sort stream with sorted() : sort on based on some condition like based on length of string. Some predefined static comparators methods are Comparator.**naturalOrder(),** Comparator.**reverseOrder()**
      4. The collect() : create new list and return list

numbers.stream().map(number -> number \*number)

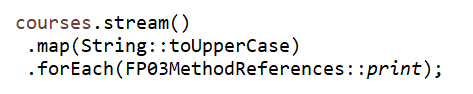
**.collect(Collectors.toList());**

* 1. Streams operation category
     1. Intermediate Operations
        1. Operations which return Stream<T> are called Intermediate operations
        2. Example: map(), filter(), distinct(), sorted()
     2. Terminal Operations
        1. The other operation which doesn’t return Stream<T> and used in last is called terminal operations
        2. Example: forEach(), collect(), reduce()

1. Functional Interface and Lambda Expression
   1. Functional Interfaces
      1. A functional interface has exactly one abstract method.
      2. Instances of functional interfaces can be created with lambda expressions, method references, or constructor references.
   2. Predicate
      1. It contains **boolean test(T t)** method
      2. It is used to test something on given argument
      3. Represents a predicate (boolean-valued function) of one argument.
   3. Function
      1. It contains **R apply(T t)** method
      2. It is used when we take one input and produce an output.
      3. Represents a function that accepts one argument and produces a result.
   4. Consumer
      1. It contains **void accept(T t)** method
      2. It calls the static method, pass the argument and return nothing.
      3. Represents an operation that accepts a single input argument and returns no result.
   5. Supplier
      1. It contains **T get()** method
      2. It takes no input and return an output
      3. Represents a supplier of results.
   6. UnaryOperator
      1. It extends **Function<T, T>**
      2. Represents an operation on a single operand that produces a result of the same type as its operand.
   7. BinaryOperator
      1. It extends **BiFunction<T, U, R>**
      2. Represents an operation upon two operands of the same type, producing a result of the same type as the operands.
   8. BiPredicate
      1. It contains boolean test(T t, U u) method
      2. Represents a predicate (boolean-valued function) of two arguments.
   9. BiFunction
      1. It contains **R apply(T t, U u)** method
      2. Represents a function that accepts two arguments and produces a result.
   10. BiConsumer
       1. It contains **void accept(T t, U u)** method
       2. Represents an operation that accepts two input arguments and returns no result.
2. Primitive Specific Functional Interface
   1. These functional Interface don’t require Autoboxing and unboxing, that’s why it increases the performance.
   2. Its available for long, double and int primitive types
   3. Example: IntPredicate, IntFunction, IntConsumer, IntSupplier, IntUnaryOperator, IntBinaryOperator, IntToDoubleFunction, IntToLongFunction
3. Behavior Parameterization with Functional Programming
   1. When we pass behavior (lambda expression, or functional interface) as parameter to the function.
   2. Passing function as an argument to function



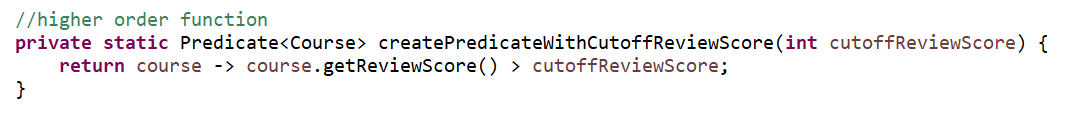
1. Method reference
   1. We can call any type of function by passing reference



1. **Constructor Reference**
   1. We can call constructor to create an object



1. Functional Programming with custom classes
   1. **Stream** API utility Functions (Stream interface)
      1. **allMatch( Predicate<T> predicate )** method: Returns whether all elements of this stream match the provided predicate.
      2. **noneMatch( Predicate<T> predicate )** method: Returns whether no elements of this stream match the provided predicate.
      3. **anyMatch( Predicate<T> predicate )** method: Returns whether any elements of this stream match the provided predicate.
      4. **limit(long maxSize)** method: return stream of length maxSize and truncated element after maxSize.
      5. **skip(long n)** method: return stream after discarding first n element.
      6. **takeWhile( Predicate<T> predicate )** method: return stream till the provided predicate fail first time.
      7. **dropWhile( Predicate<T> predicate )** method: return stream after discarding the elements which predicate pass till first time fails.
      8. **max(Comparator<T> comparator) method:** Returns the maximum element of this stream according to the provided comparator.
      9. **min(Comparator<T> comparator) method:** Returns the minimum element of this stream according to the provided comparator.
      10. **findFirst() method**: Returns an Optional describing the first element of this stream, or an empty Optional if the stream is empty.
      11. **findAny() method**: Returns an Optional describing some element of the stream, or an empty Optional if the stream is empty.
      12. **mapToInt() method**: Returns an IntStream consisting of the results of applying the given function to the elements of this stream.
      13. **peek() method:** Returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream. Can be used to print like forEach.
   2. **IntStream** interface
      1. **sum() method**: Returns the sum of elements in this stream.
      2. **average() method**: Returns an OptionalDouble describing the arithmetic mean of elements of this stream, or an empty optional if this stream is empty.
      3. **count() method:** Returns the count of elements in this stream.
      4. **max() method:** Returns an OptionalInt describing the maximum element of this stream, or an empty optional if this stream is empty.
   3. **Comparator** Functional Interface
      1. It contains **int compare(T o1, T o2)** method
      2. The static functions
         1. **comparing(Function<T, U>)** method and available for Primitive type as well (Int, Long, Double)
         2. **thenComparing(Function<T, U>)** method and available for Primitive type as well (Int, Long, Double)
         3. **nullsFirst()** method
         4. **nullsLast()** method
      3. The default functions
         1. **reversed()** method
   4. **Collectors** class
      1. **groupingBy() method**: Returns a Collector implementing a "group by" operation on input elements of type T, grouping elements according to a classification function, and returning the results in a Map.
      2. **maxBy() method**: Returns a Collector that produces the maximal element according to a given Comparator, described as an Optional<T>.
      3. **joining() method**: Returns a Collector that concatenates the input elements, separated by the specified delimiter, in encounter order.
2. Creating Streams using Stream method
   1. Using List, Set and Map
      1. List.of(…T t)
      2. Set.of(…T t)
      3. Map.of(…T t).values()
      4. Map.of(…T t).keySet()
      5. Map.of(…T t).entrySet()
   2. Using Stream Class
      1. Stream.of(…T t)
      2. IntStream.of(…T t)
      3. LongStream.of(…T t)
      4. DoubleStream.of(…T t)
   3. Using Array
      1. Arrays.stream(arrayObject);
   4. Create Range of Stream
      1. IntStream.range(1, 10)
      2. IntStream.rangeClosed(1,10)
      3. IntStream.iterate(1, e -> e + 1).limit(10) {First 10 no}
      4. IntStream.iterate(1, e -> e + 1).limit(10) {First 10 odd no}
      5. IntStream.iterate(2, e -> e + 2).limit(10) {first 10 even no}
      6. IntStream.iterate(2, e -> e \* 2).limit(10) {first 10 square of 2}
   5. Create list from IntStream
      1. boxed(): IntStream.iterate(2, e -> e \* 2).limit(10).boxed().collect(Collectors.toList());
   6. Create Big number than Integer.MAX\_VALUE
      1. IntStream.range(1, 50).mapToObj(BigInteger::valueOf).reduce(BigInteger.ONE, BigInteger::multiply);
3. String Opreations
   1. flatMap() method
      1. Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
      2. Used to convert stream of String[] -> stream of Strings
4. Higher Order Functions
   1. If we return function from a function, then it is called higher order functions



1. FP and Performance
   1. In Stream, all the intermediate operation are lazy means they are only executed when java execute the terminal operation.
   2. It executes all intermediate and terminal operation for one element then go to next element.
   3. If we don’t have terminal operation, then it will not run any of intermediate operations for stream.
2. Improvement Performance with Parallelization of streams
   1. **parallel()** method
      1. Returns an equivalent stream that is parallel. May return itself, either because the stream was already parallel, or because the underlying stream state was modified to be parallel.
      2. It takes less time than normal stream()
3. Modifying List
   1. List method also accept functional interface like replaceAll, removeIf etc.
4. Files using Functional Programming
   1. Files.lines() method: Read all lines from a file as a Stream
   2. Files.list() method: Return a lazily populated Stream, the elements of which are the entries in the directory.
5. Threads using Functional Programming

