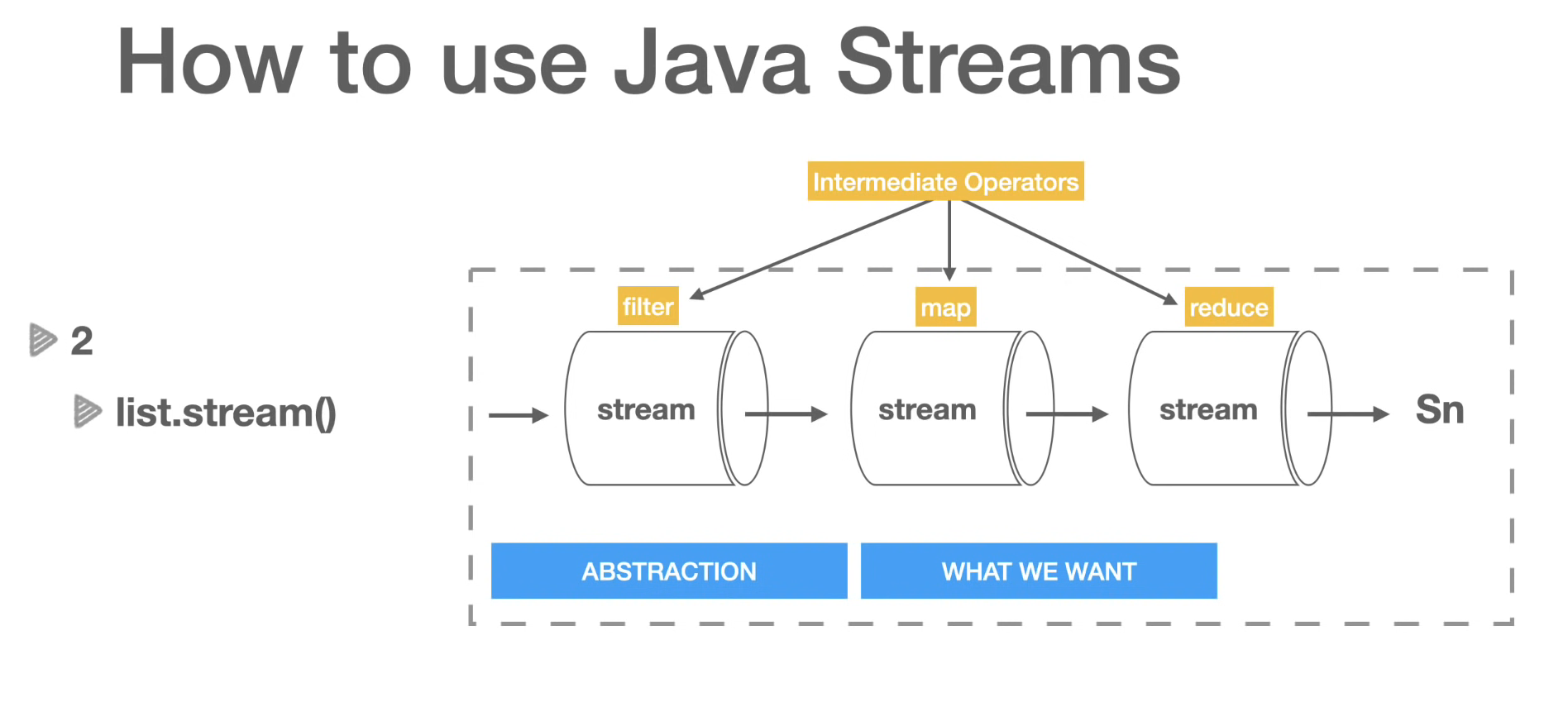
**Java-migoscode:**

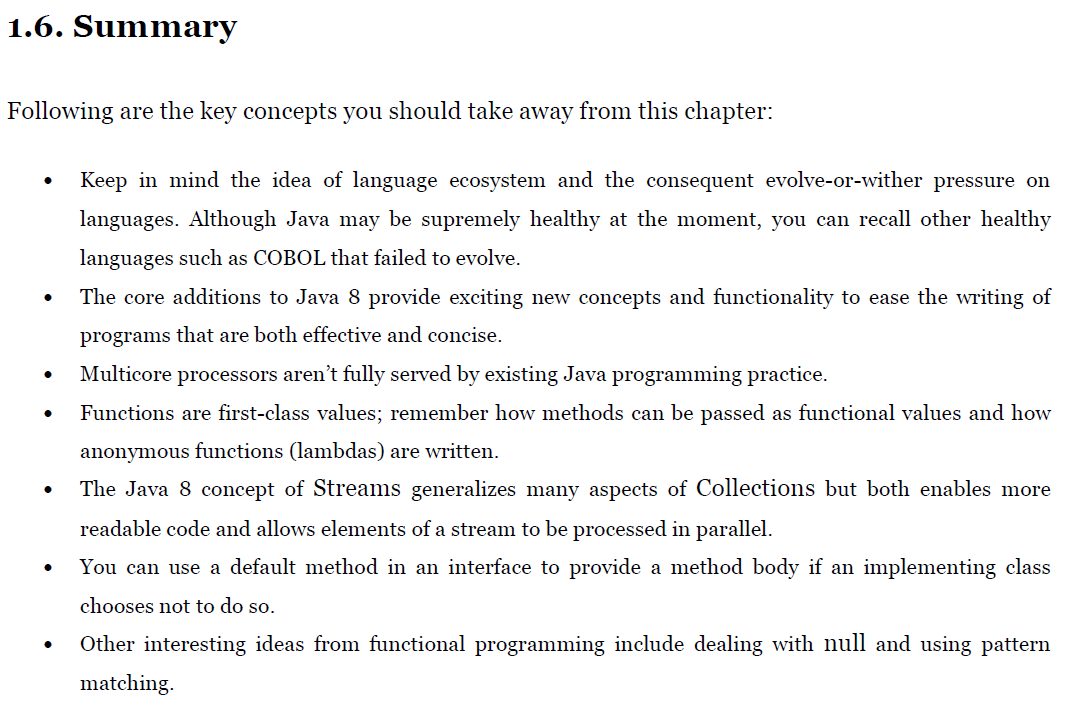




**-----------------------------------------------------------------**

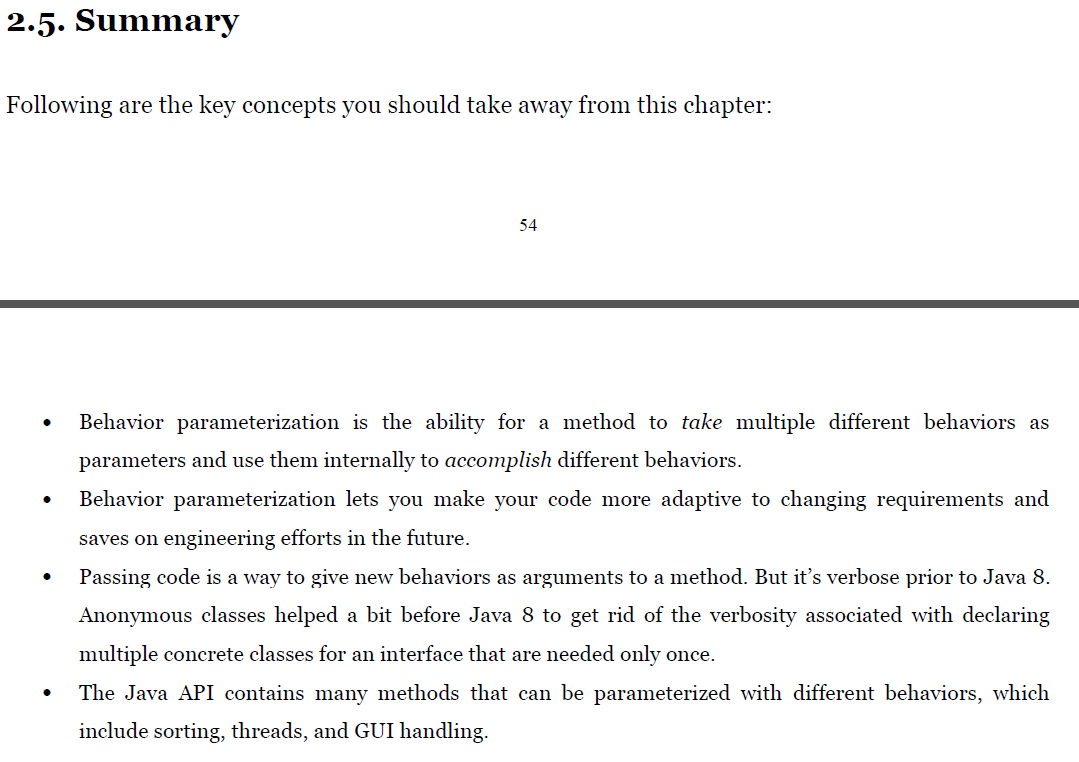
**Java in action:**

**Chapter 1. Java 8: why should you care?**



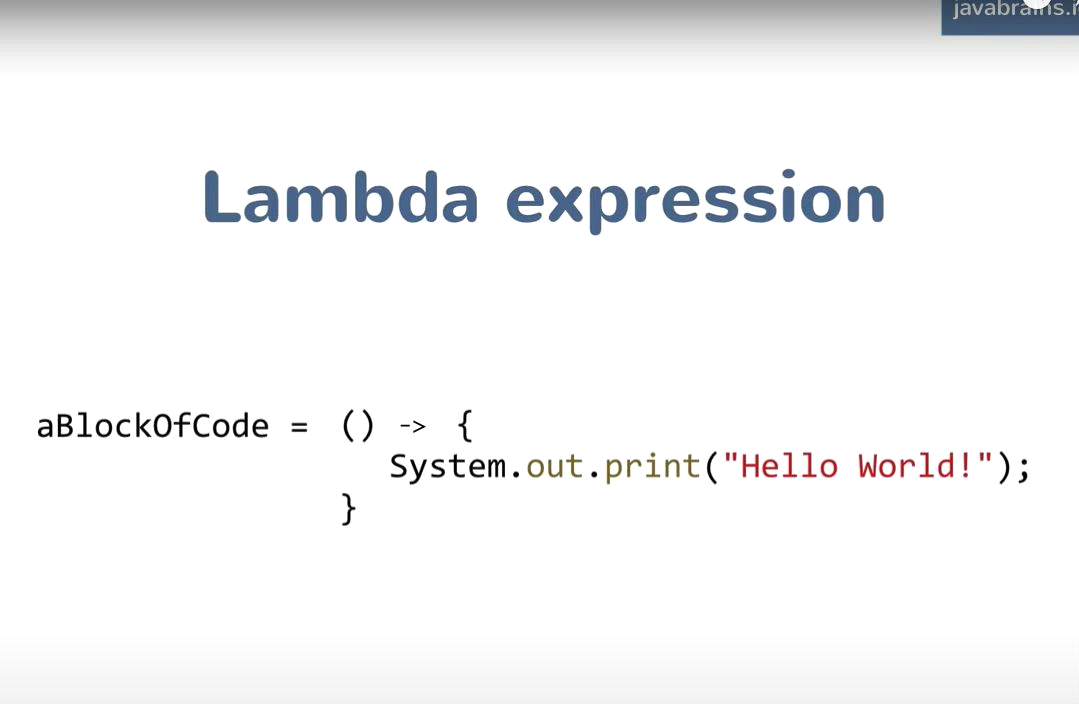
**Chapter 2. Passing code with behavior**

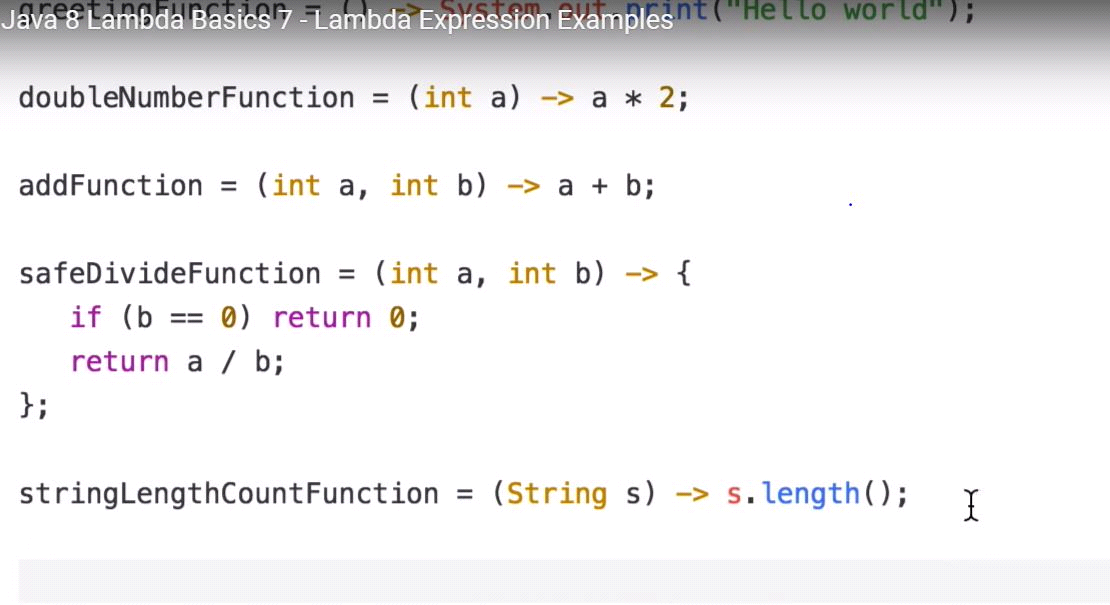
**Parameterization**



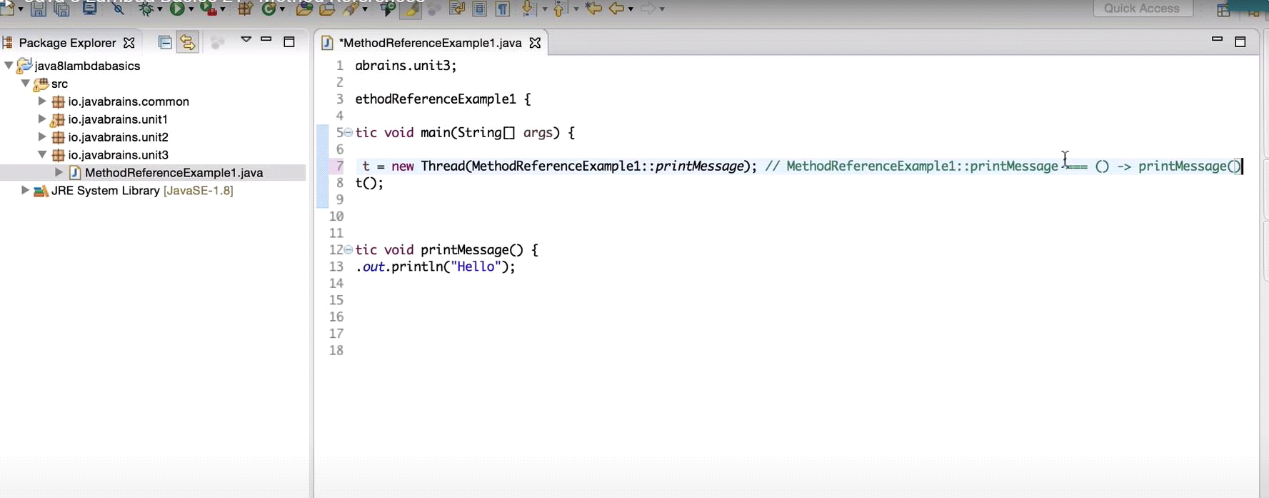
**Chapter 3. Lambda expressions**

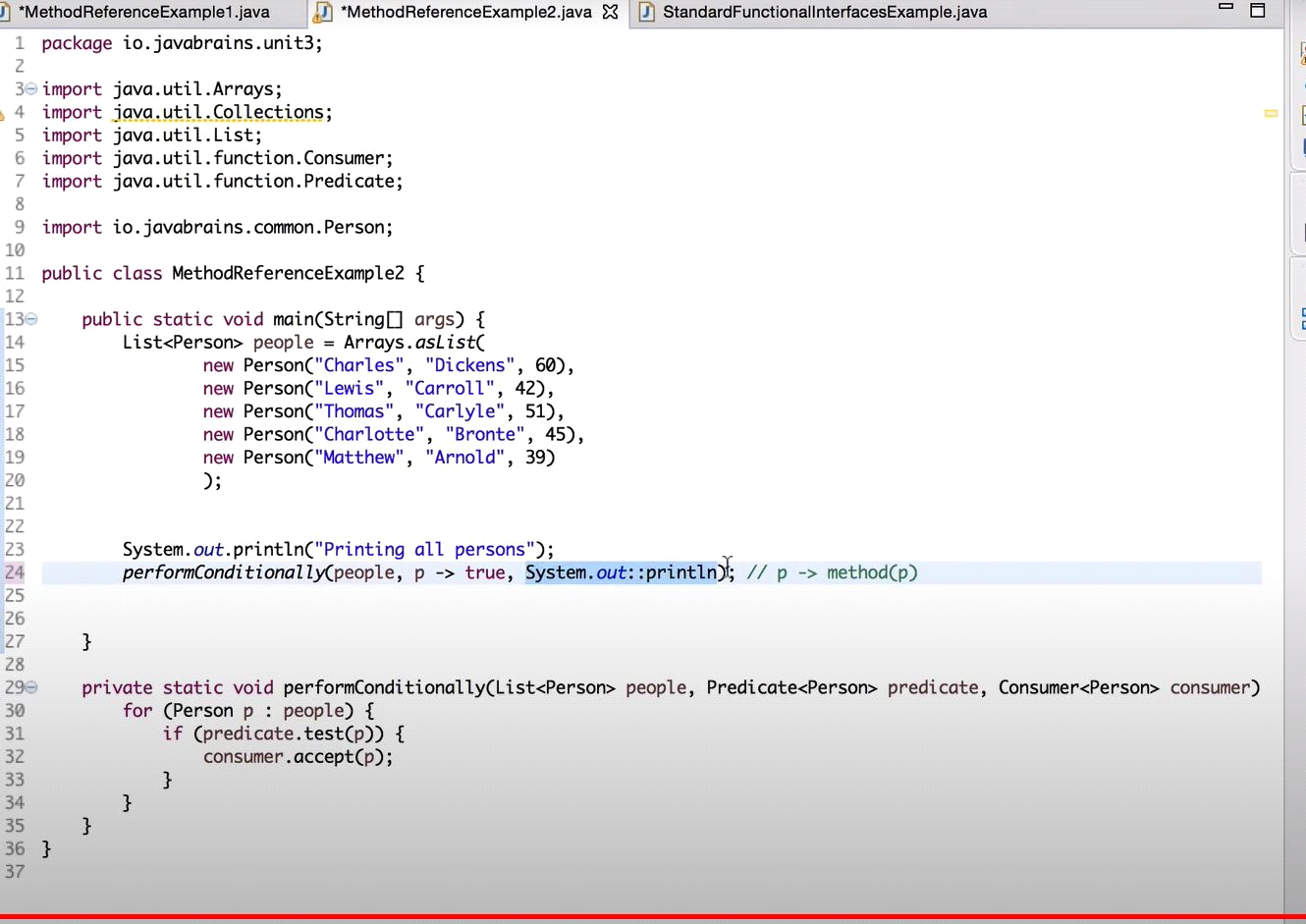


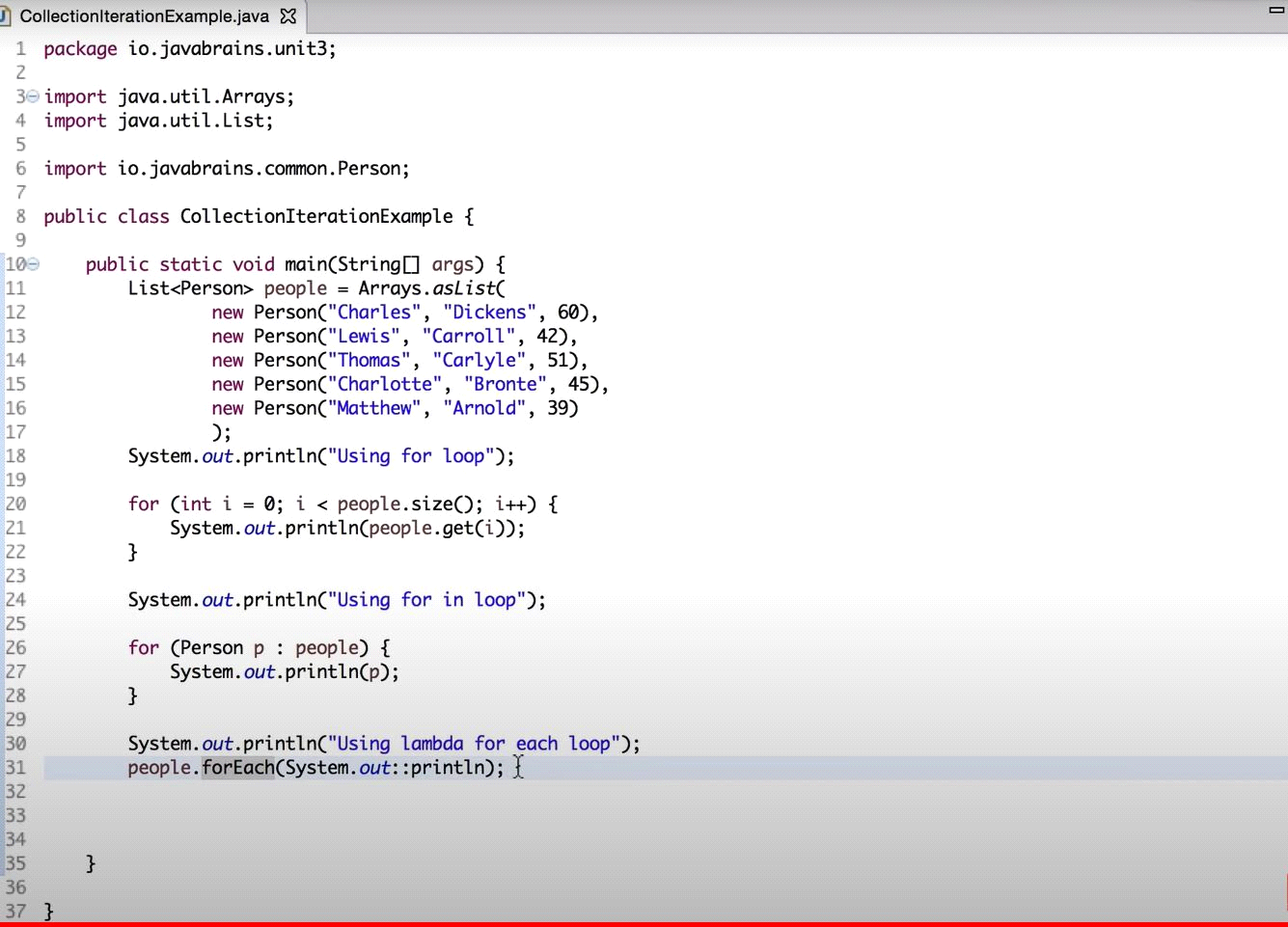


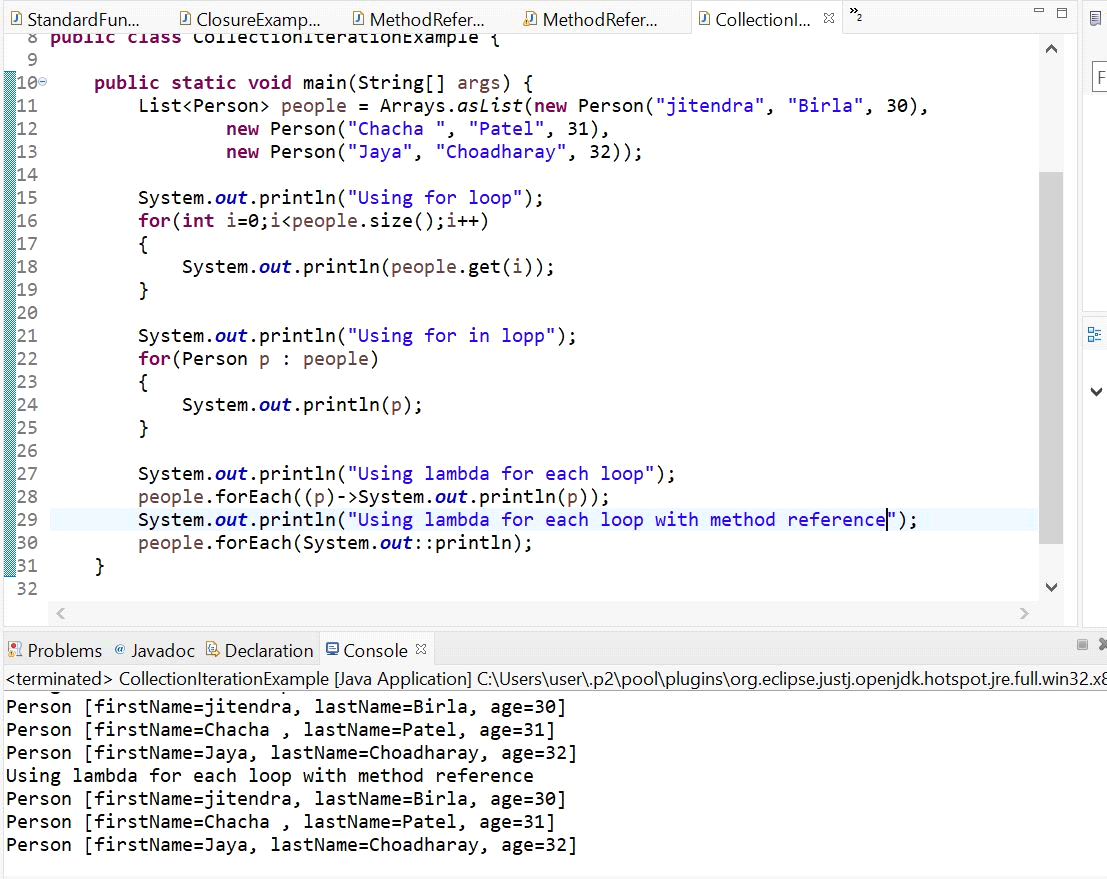


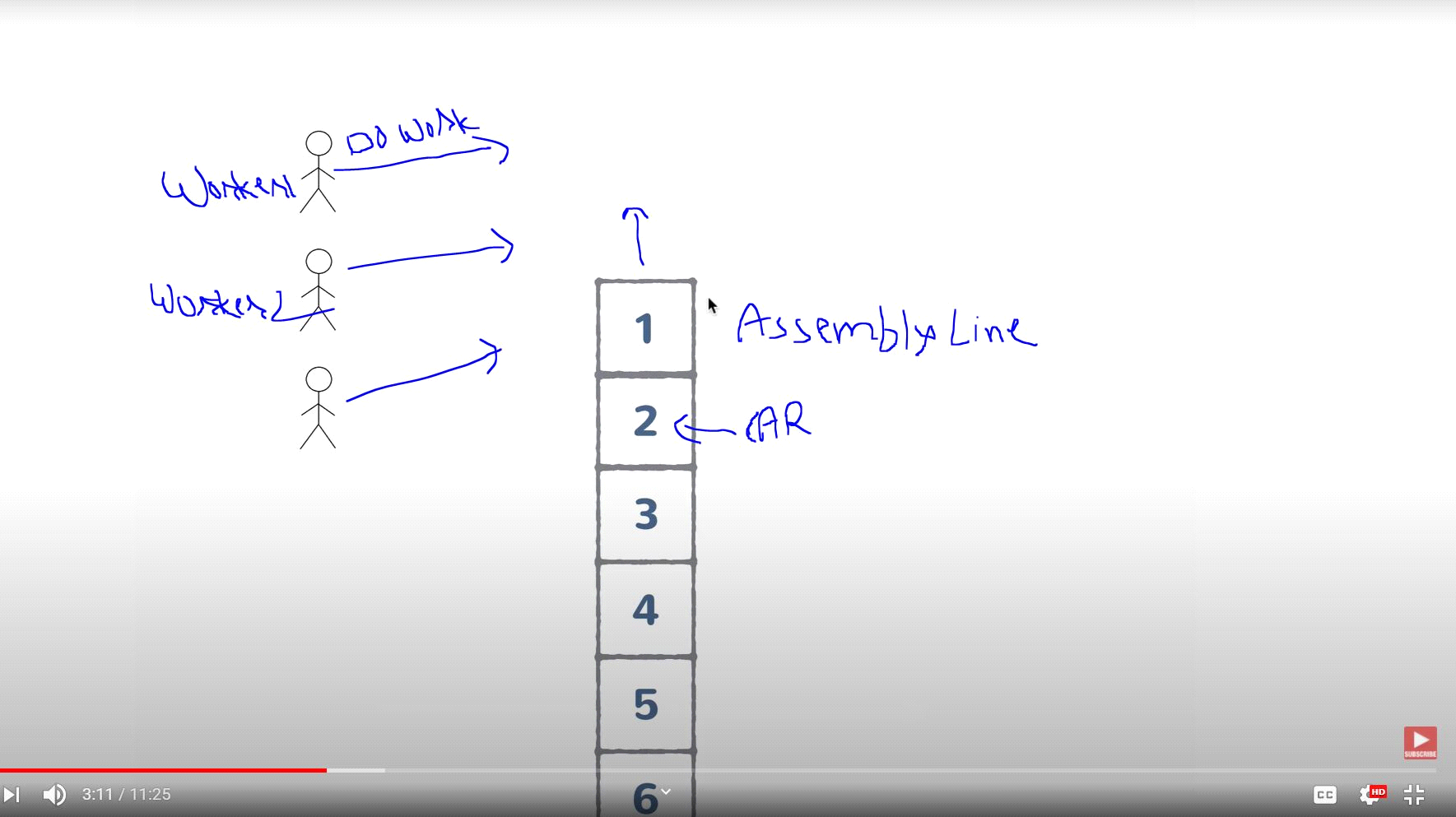


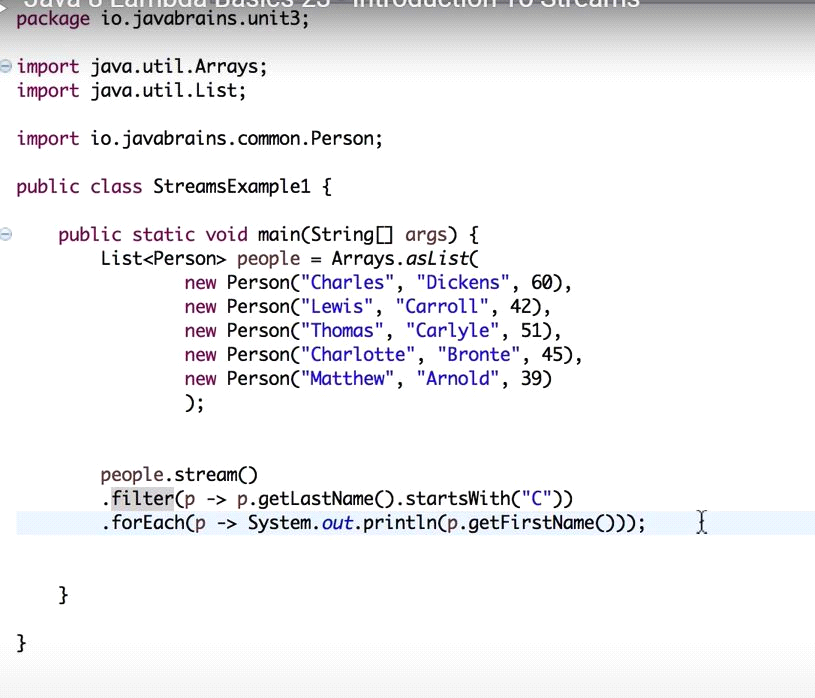


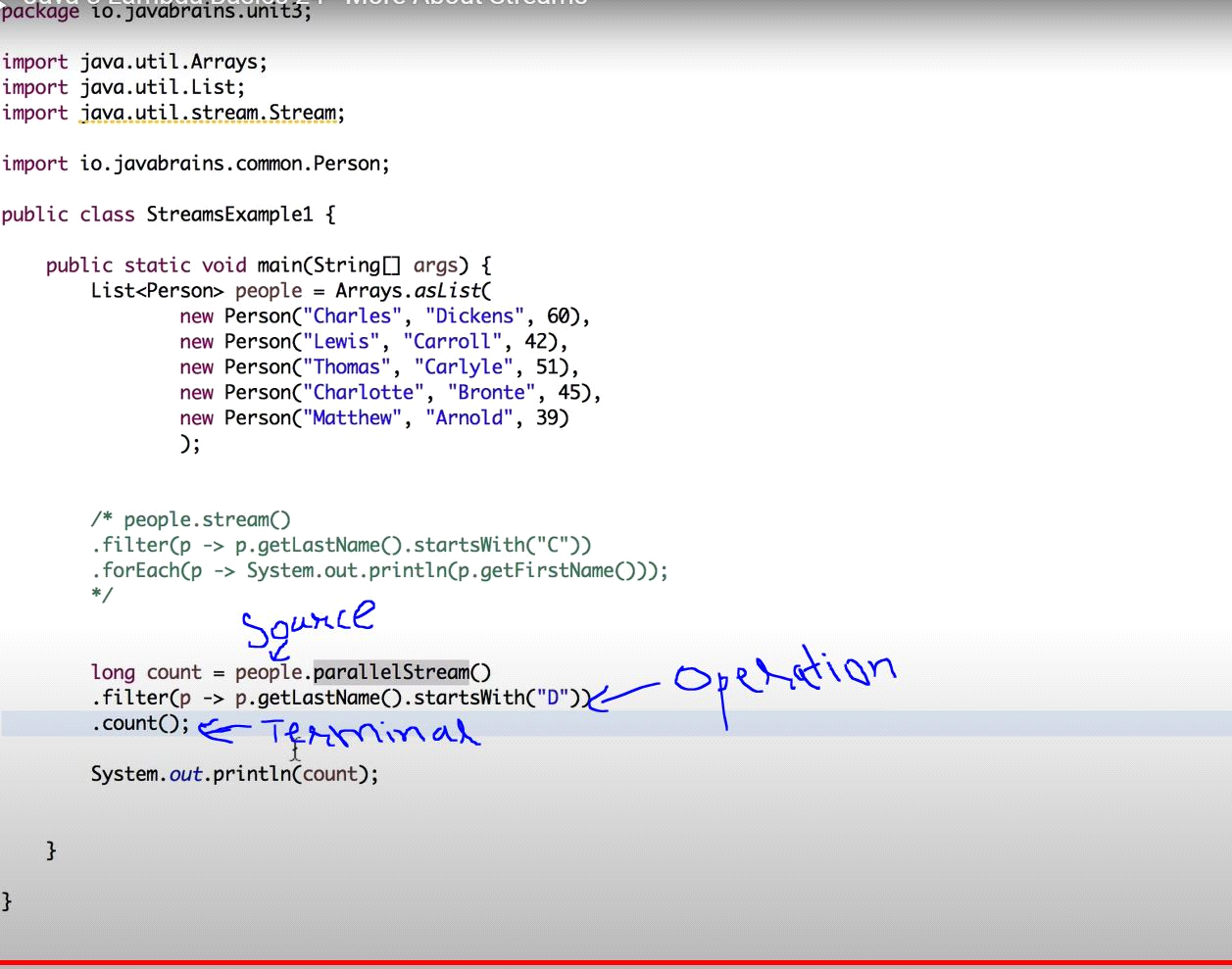




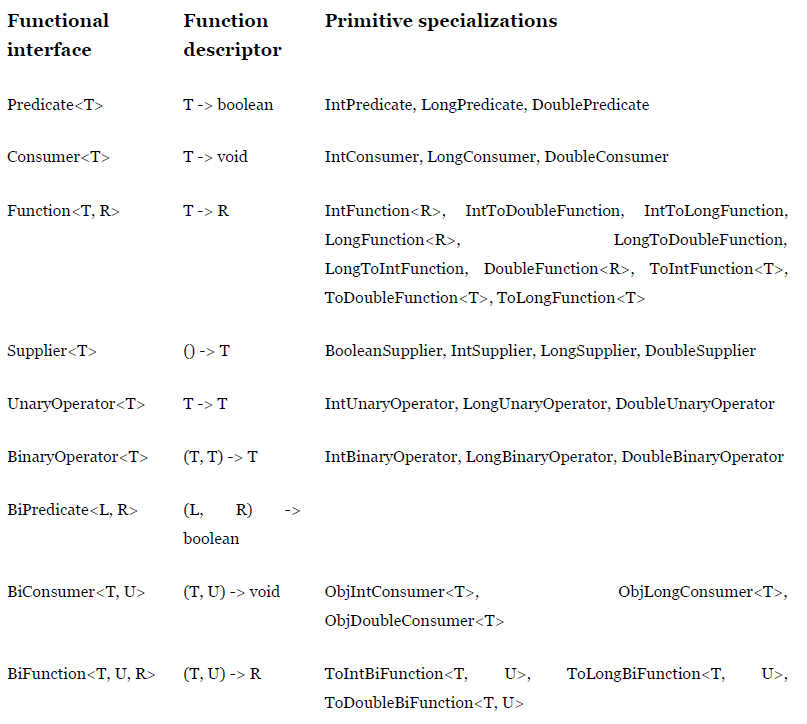


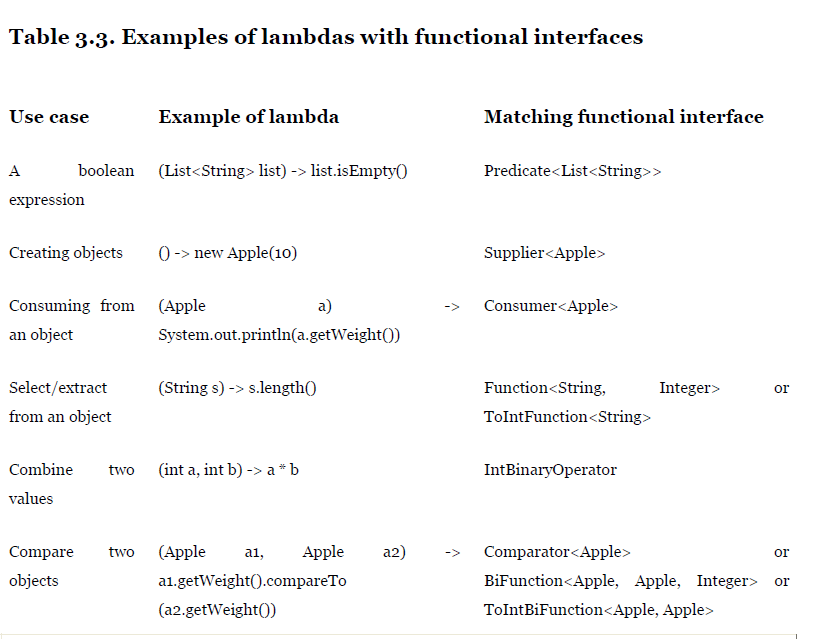


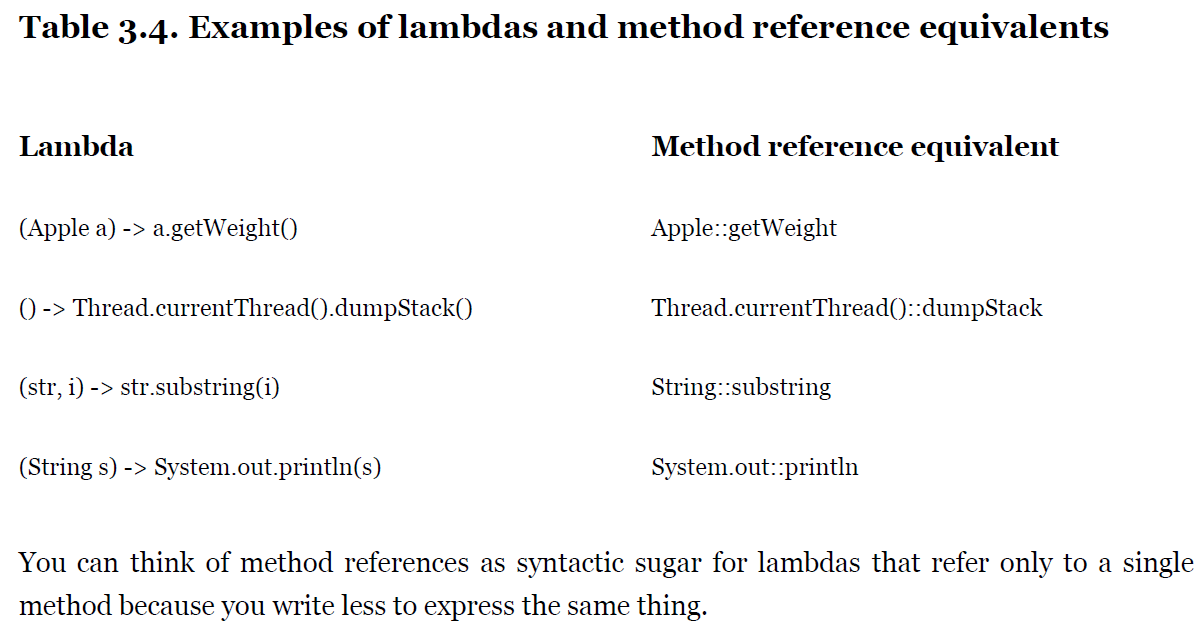


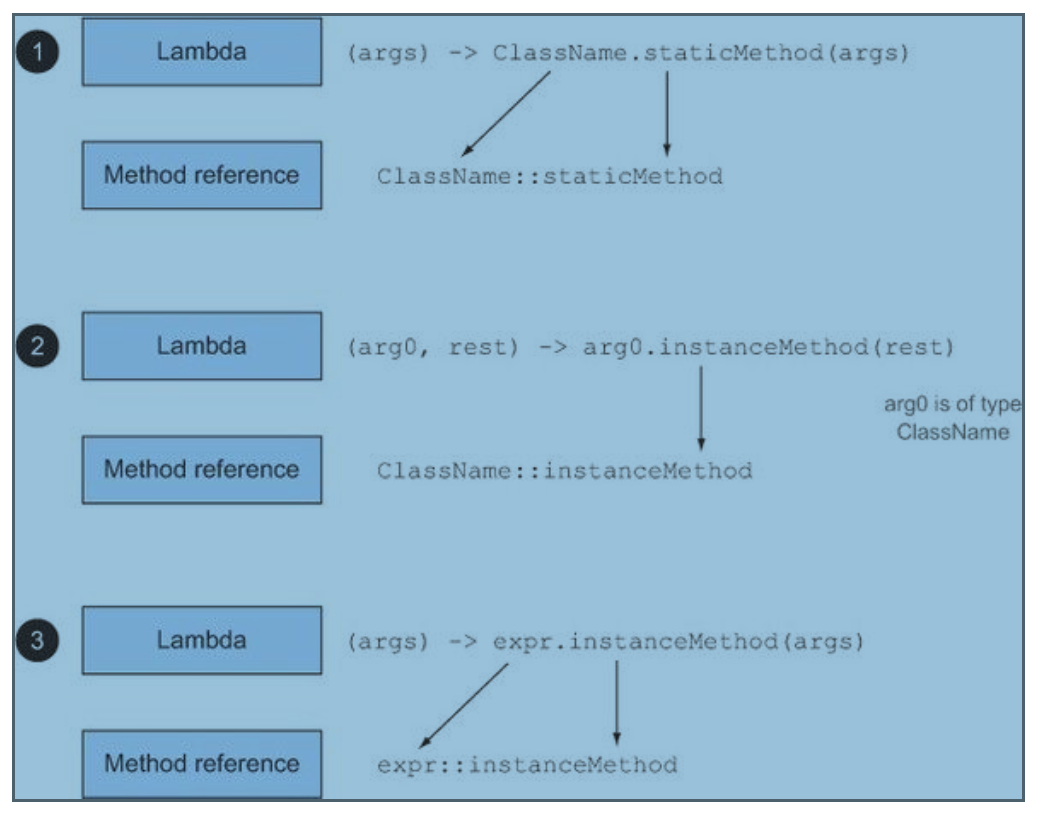


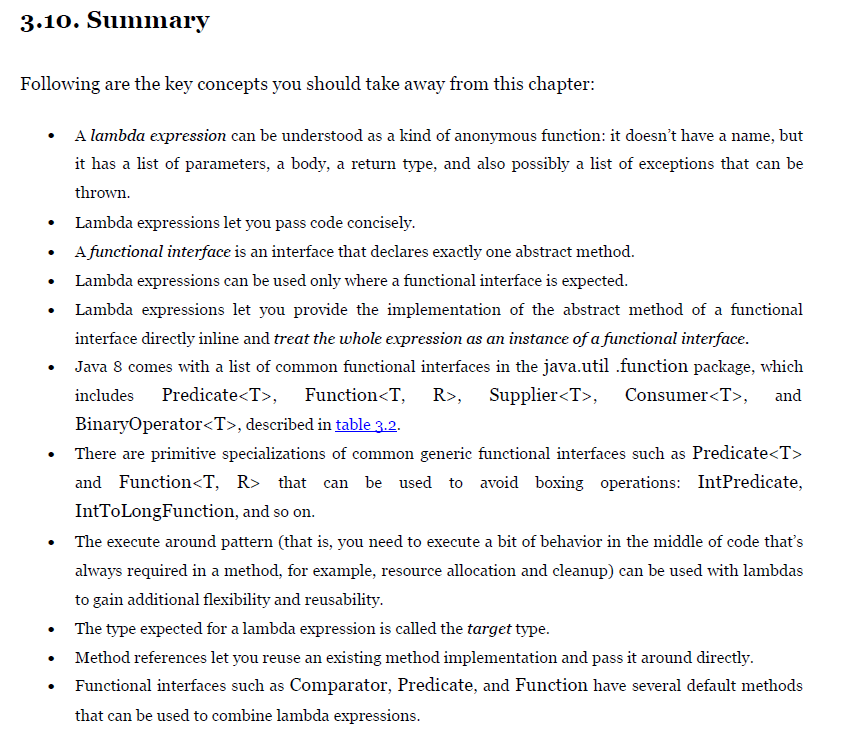
**Table 3.2. Common functional interfaces in Java 8**



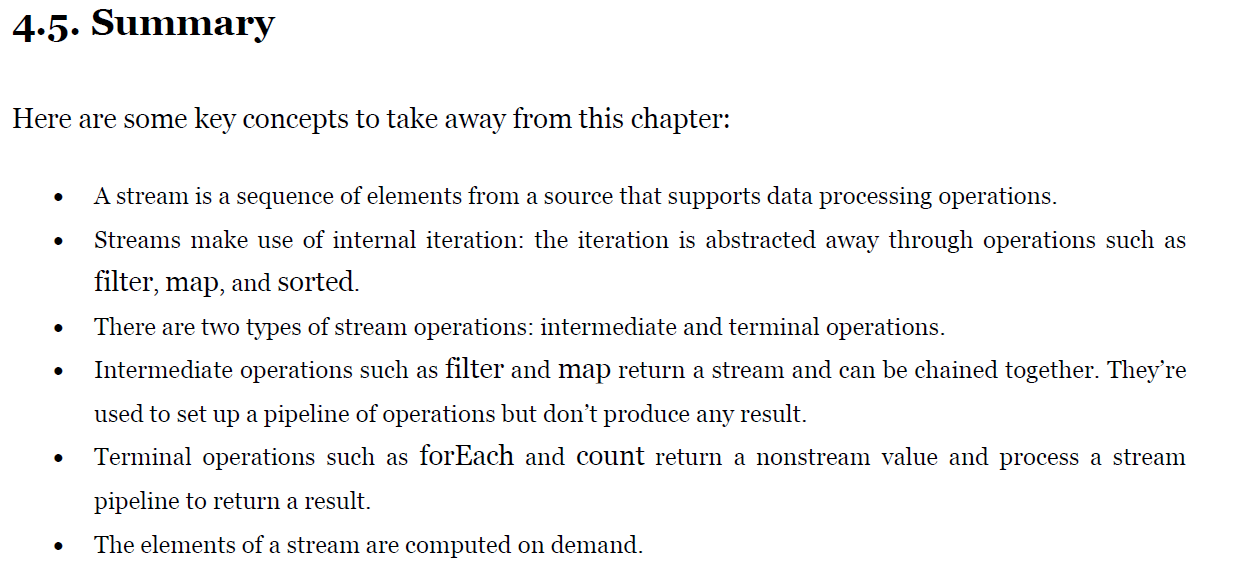




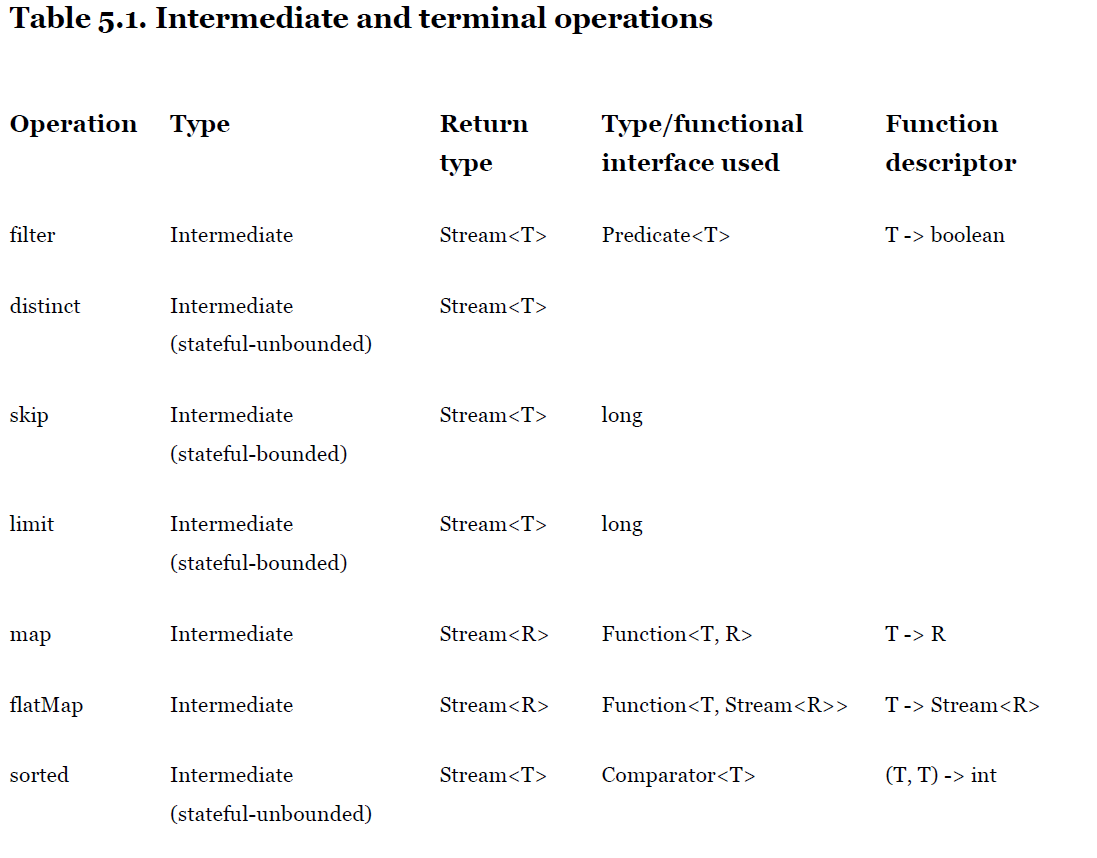


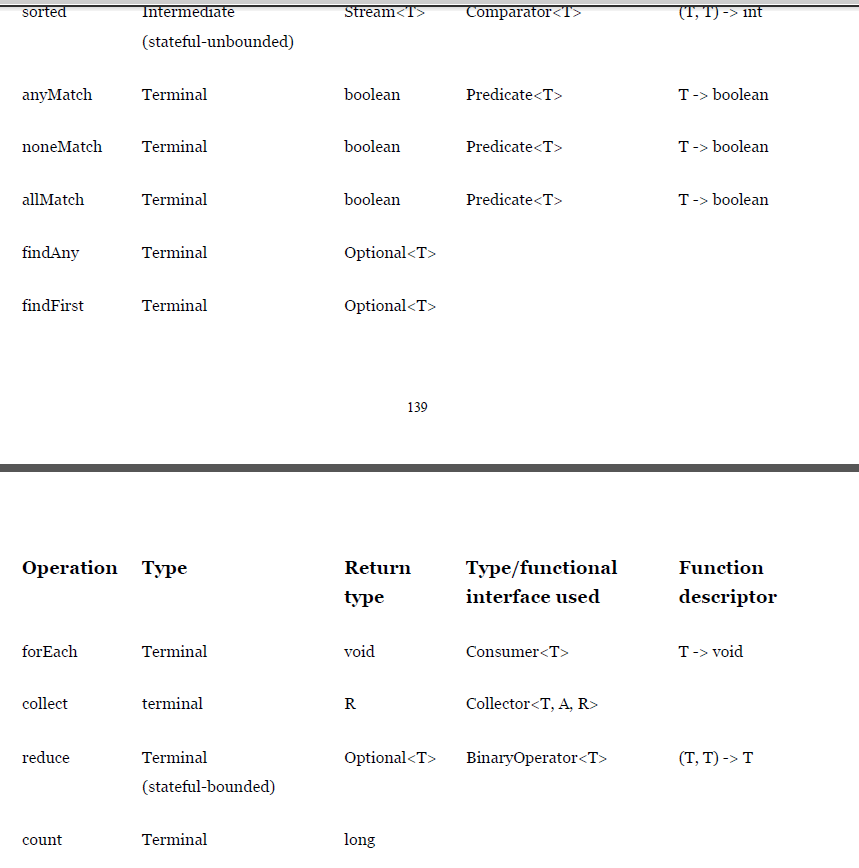


**Chapter 4. Introducing streams**

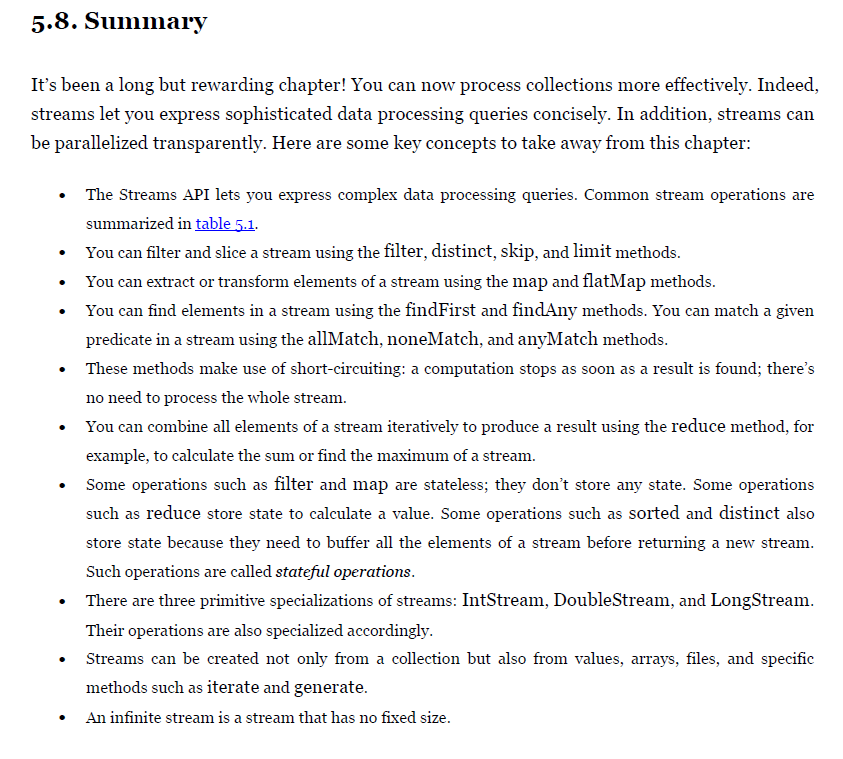


**Chapter 5. Working with streams**





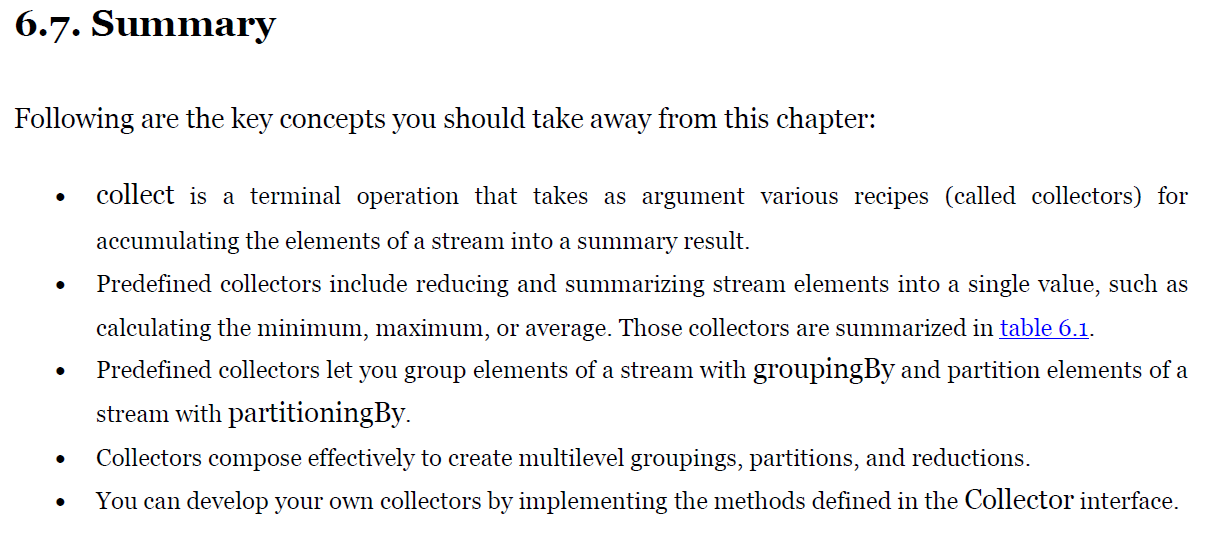
|  |
| --- |
| package streams.world;    import java.util.Arrays;  import java.util.Comparator;  import java.util.List;  import java.util.Optional;  import java.util.stream.Collectors;    public class JavaInActionExcercise1 {    public static void main(String[] args) {  List<Transaction> transactions = dataPopulate();    System.out.println("transactions : " + transactions);    System.out  .println("\*\*\*\*\*//Q1: Find all transactions in the year 2011 and sort them by value(small to high)\*\*\*");    List<Transaction> transaction2011 = transactions.stream().filter(t -> t.getYear() == 2011)  .sorted(Comparator.comparing(Transaction::getValue)).collect(Collectors.toList());  System.out.println("transaction2011 : " + transaction2011);  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");    System.out.println("\*\*\*\*\*//Q2: What are all the unique cities where the traders work\*\*\*");    List<String> uniqueCities = transactions.stream().map(Transaction::getTrader).map(Trader::getCity).distinct()  .collect(Collectors.toList());    System.out.println("uniqueCities : " + uniqueCities);    System.out.println("\*\*\*\*\*//Q3 : Find all traders from Cambridge and sort them by name\*\*\*");  List<Trader> listTraders = transactions.stream().map(Transaction::getTrader)  .filter(t -> t.getCity() == "Cambridge").sorted(Comparator.comparing(Trader::getName))  .collect(Collectors.toList());  System.out.println("listTraders : " + listTraders);    System.out.println("\*\*\*\*\*//Q4 : Return a string of all traders names sorted alphabetically\*\*\*");    String sortedNames = transactions.stream().map(t -> t.getTrader().getName()).distinct().sorted().reduce("",  (n1, n2) -> n1 + " " + n2);  System.out.println("SortedNames : " + sortedNames);    System.out.println("\*\*\*\*\*//Q5 : Are any traders based in Milan\*\*\*");  System.out.println(transactions.stream().map(Transaction::getTrader).anyMatch(t -> t.getCity() == "Milan"));    System.out.println(  transactions.stream().map(Transaction::getTrader).filter(t -> t.getCity() == "Milan").findAny());    System.out.println("\*\*\*\*\*//Q6 :Print all transactions values from the traders living in cambridge \*\*\*");  List<Integer> listCambridgeTrx = transactions.stream().filter(t -> t.getTrader().getCity() == "Cambridge")  .map(Transaction::getValue).collect(Collectors.toList());  System.out.println("listCambridgeTrx : " + listCambridgeTrx);    System.out.println("\*\*\*\*\*//Q7 :What's the highest value of all the transaction?\*\*\*");  Optional<Integer> highestValue = transactions.stream().map(Transaction::getValue).reduce(Integer::max);  System.out.println("highestValue : " + highestValue.get());    System.out.println("\*\*\*\*\*//Q8 :Find the transaction with the smallest value\*\*\*");  Optional<Transaction> smallesValue = transactions.stream()  .reduce((t1, t2) -> t1.getValue() < t2.getValue() ? t1 : t2);  System.out.println("smallesValue : " + smallesValue.get());    }    private static List<Transaction> dataPopulate() {  Trader raoul = new Trader("Raoul", "Cambridge");  Trader mario = new Trader("Mario", "Milan");  Trader alan = new Trader("Alan", "Cambridge");  Trader brian = new Trader("Brian", "Cambridge");    List<Transaction> transactions = Arrays.asList(new Transaction(brian, 2011, 300),  new Transaction(raoul, 2012, 1000), new Transaction(raoul, 2011, 400),  new Transaction(mario, 2012, 710), new Transaction(mario, 2012, 700), new Transaction(alan, 2012, 950));    return transactions;  }  } |



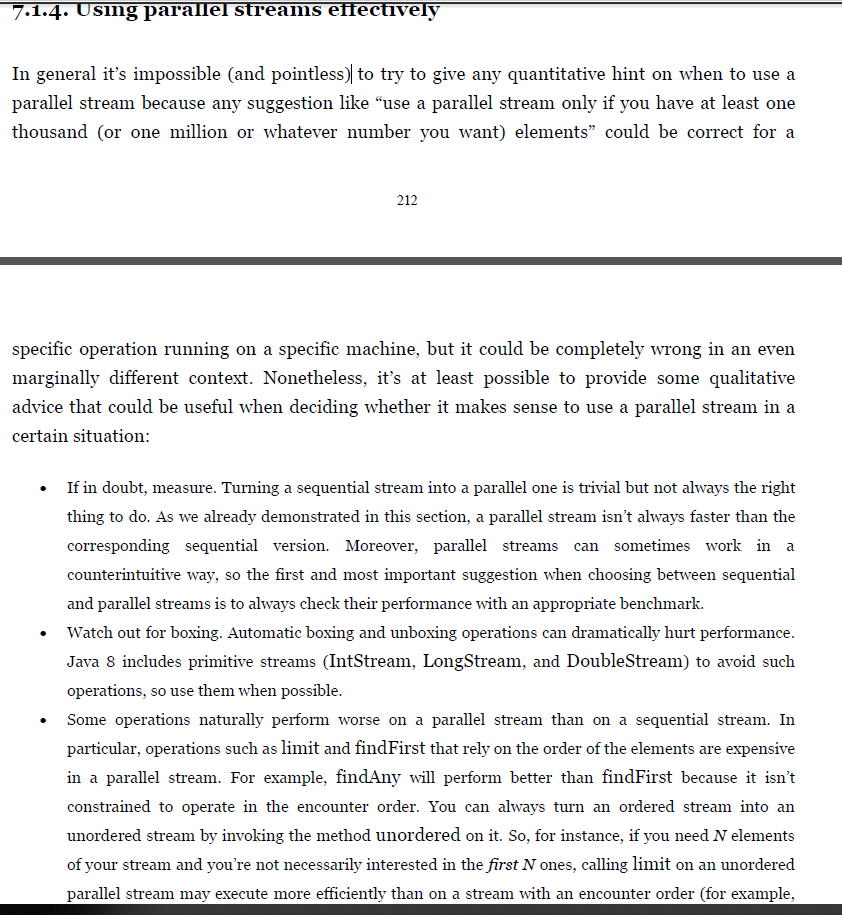
**Chapter 6. Collecting data with streams**

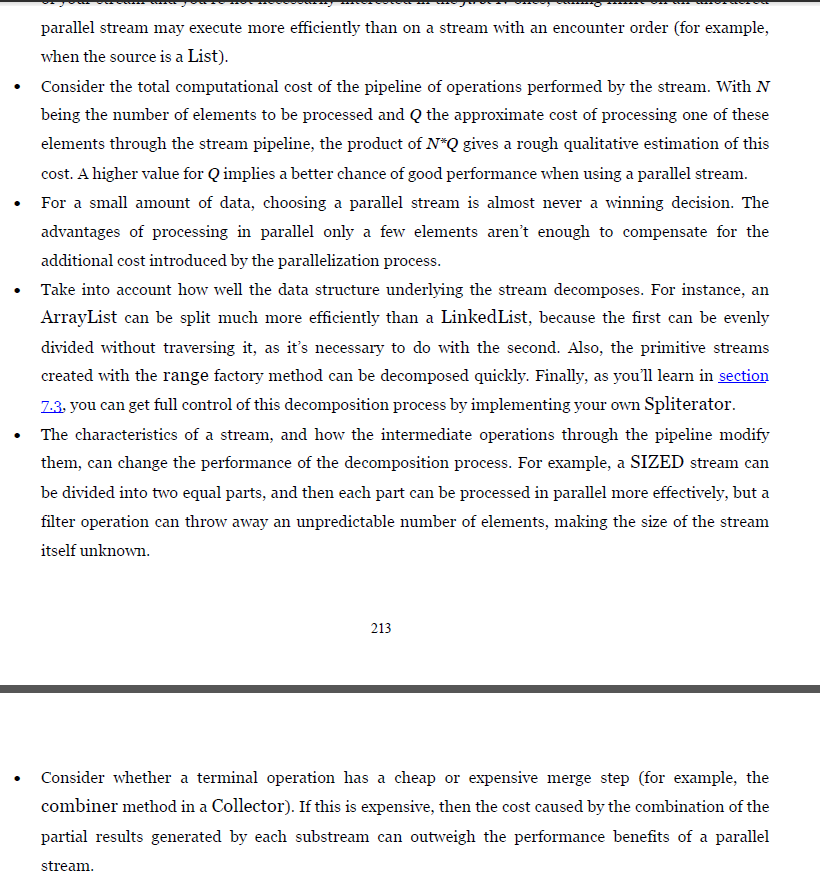
**Table 6.1. The static factory methods of the Collectors class**

|  |  |  |  |
| --- | --- | --- | --- |
| **Factory**  **method** | **Returned type** | **Used to** | Example use |
| toList | List<T> | Gather all the stream’s items in a List. | Example use: List<Dish> dishes = menuStream.collect(toList()); |
| toSet | Set<T> | Gather all the stream’s items in a Set, eliminating  duplicates. | Example use: Set<Dish> dishes = menuStream.collect(toSet()); |
| toCollection | Collection<T> | Gather all the stream’s items in the collection created  by the provided supplier. | Example use: Collection<Dish> dishes = menuStream.collect(toCollection(), ArrayList::new); |
| counting | Long | Count the number of items in the stream. | Example use: long howManyDishes = menuStream.collect(counting()); |
| summingInt | Integer | Sum the values of an Integer property of the items in  the stream. | Example use: int totalCalories = menuStream.collect(summingInt(Dish::getCalories)); |
| averagingInt | Double | Calculate the average value of an Integer property of  the items in the stream. | Example use: double avgCalories = menuStream.collect(averagingInt(Dish::getCalories)); |
| summarizingInt | IntSummary-Statistics | Collect statistics regarding an Integer property of the  items in the stream, such as the maximum, minimum, total, and average. | Example use: IntSummaryStatistics menuStatistics =  menuStream.collect(summarizingInt(Dish::getCalories)); |
| joining | String | Concatenate the strings resulting from the invocation  of the toString method on each item of the stream | Example use: String shortMenu = menuStream.map(Dish::getName).collect(joining(", ")); |
| maxBy | Optional<T> | An Optional wrapping the maximal element in this  stream according to the given comparator or  Optional.empty() if the stream is empty. | Example use: Optional<Dish> fattest = menuStream.collect(maxBy(comparingInt(Dish::getCalories))); |
| minBy | Optional<T> | An Optional wrapping the minimal element in this  stream according to the given comparator or  Optional.empty() if the stream is empty. | Example use: Optional<Dish> lightest = menuStream.collect(minBy(comparingInt(Dish::getCalories))); |
| reducing | The type produced by the  reduction operation | Reduce the stream to a single value starting from an  initial value used as accumulator and iteratively  combining it with each item of the stream using a  BinaryOperator. | Example use: int totalCalories = menuStream.collect(reducing(0, Dish::getCalories, Integer::sum)); |
| collectingAndThen | The type returned by the  transforming function | Wrap another collector and apply a transformation  function to its result | Example use: int howManyDishes = menuStream.collect(collectingAndThen(toList(), List::size)); |
| groupingBy | Map<K, List<T>> | Group the items in the stream based on the value of  one of their properties and use those values as keys in  the resulting Map. | Example use: Map<Dish.Type, List<Dish>> dishesByType =  menuStream.collect(groupingBy(Dish::getType)); |
| partitioningBy | Map<Boolean, List<T>> | Partition the items in the stream based on the result of  the application of a predicate to each of them. | Example use: Map<Boolean, List<Dish>> vegetarianDishes =  menuStream.collect(partitioningBy(Dish::isVegetarian)); |
|  |  |  |  |



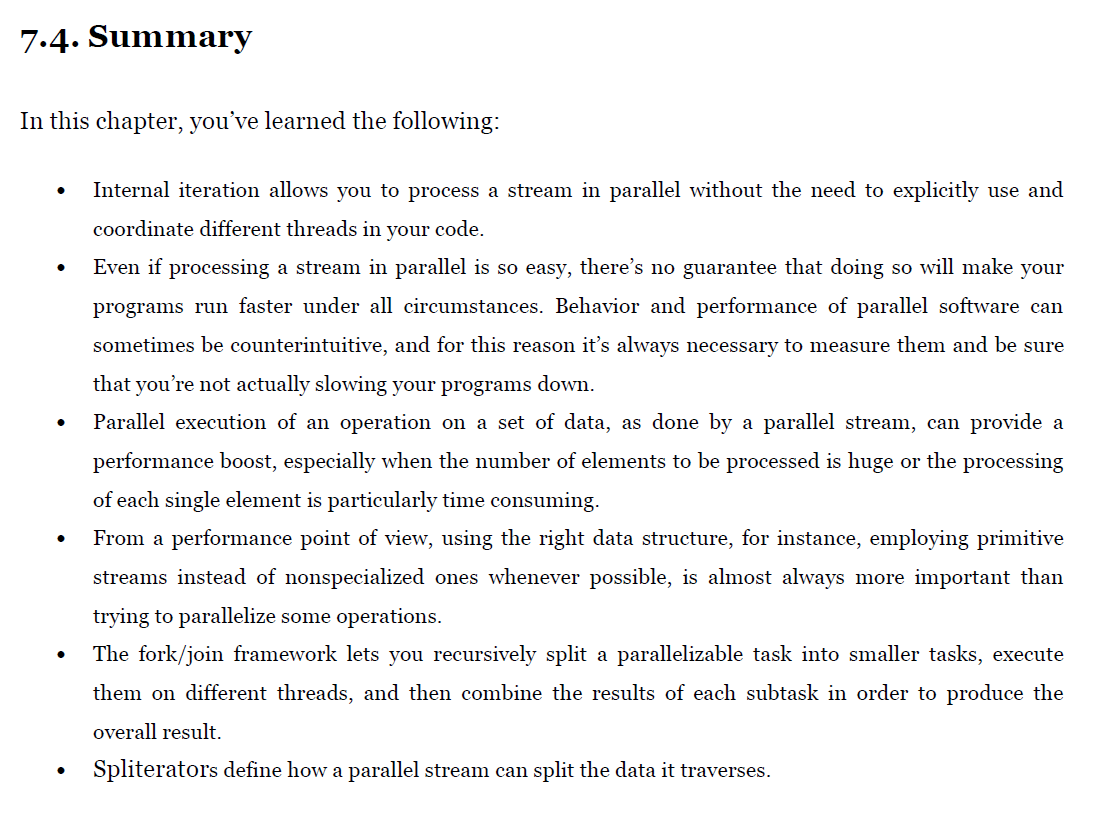
**Chapter 7. Parallel data processing and performance**



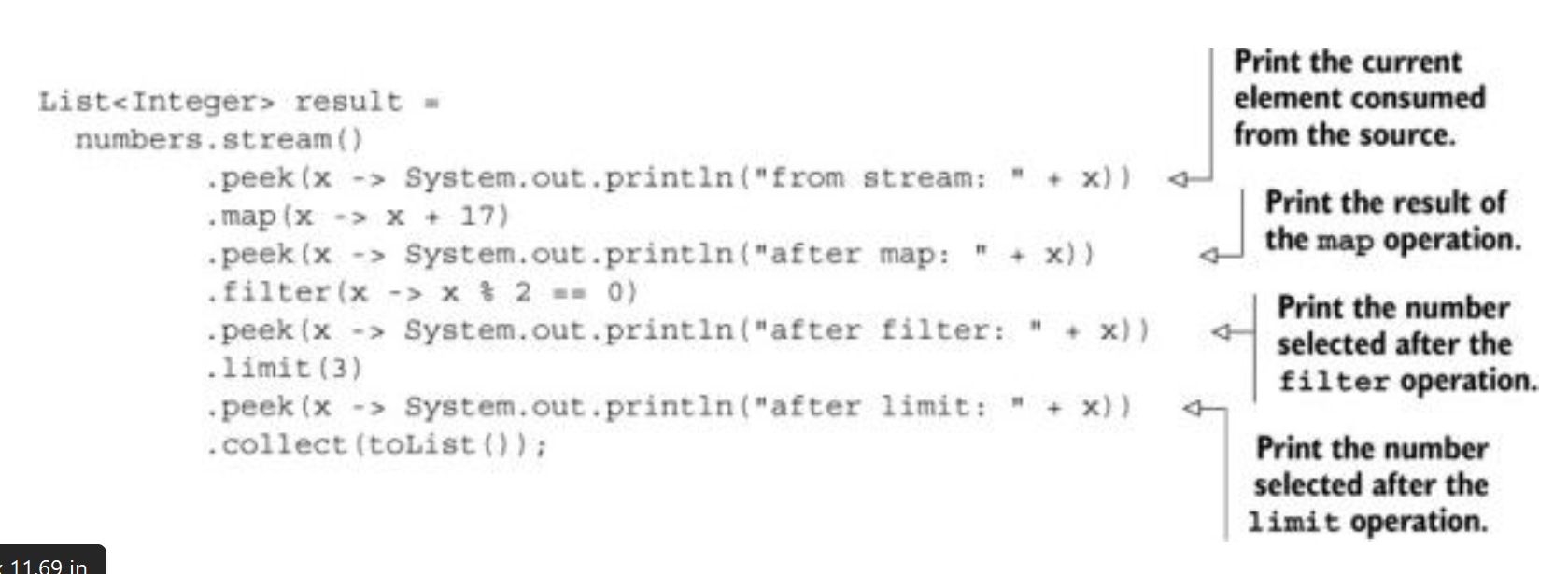


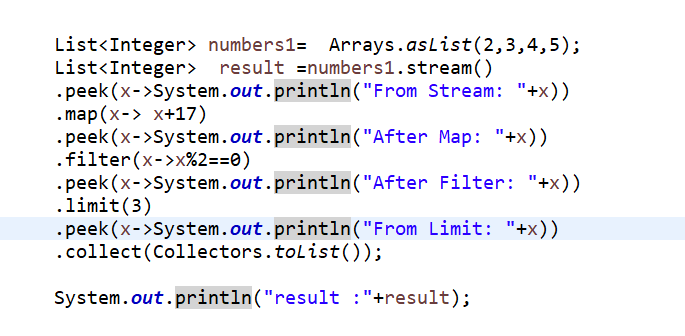
**Table 7.1. Stream sources and decomposability**

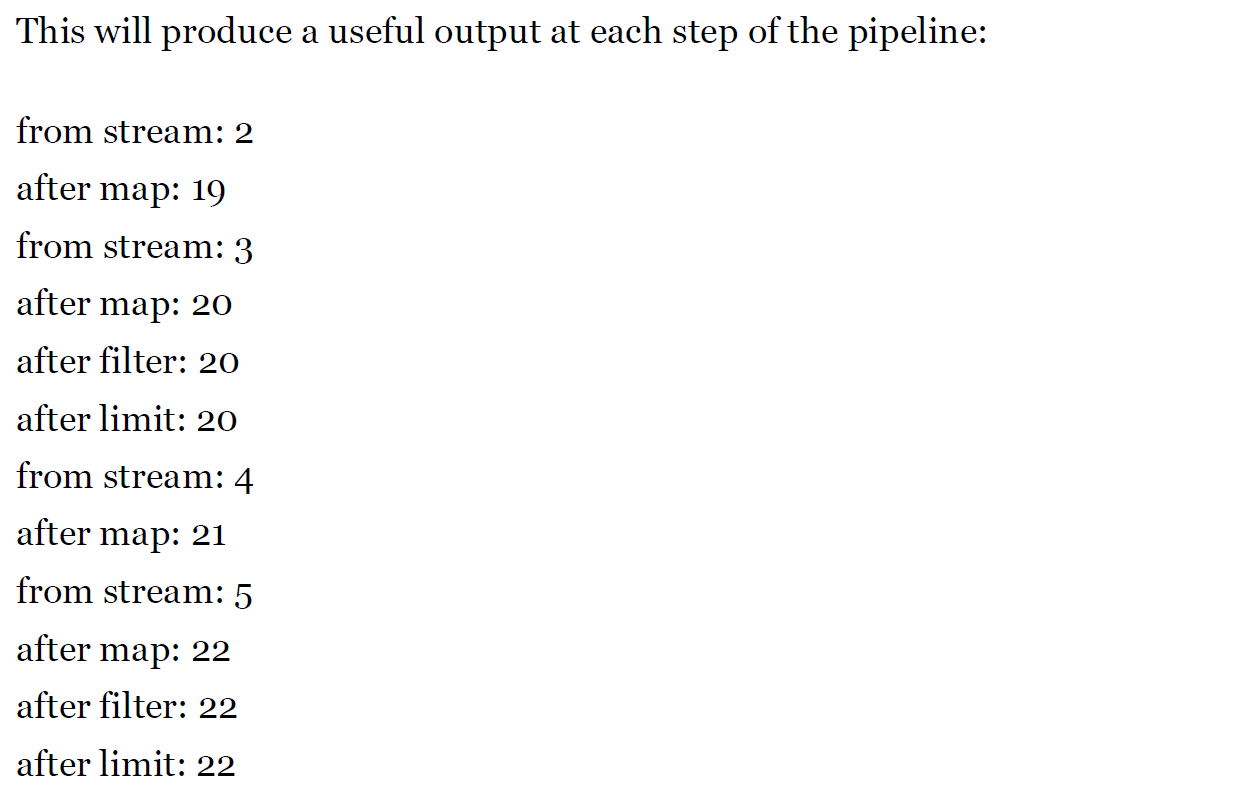
|  |  |
| --- | --- |
| **Source** | **Decomposability** |
| ArrayList | Excellent |
| LinkedList | Poor |
| IntStream.range | Excellent |
| Stream.iterate | Poor |
| Stream.iterate | Poor |
| HashSet | Good |
| TreeSet | Good |

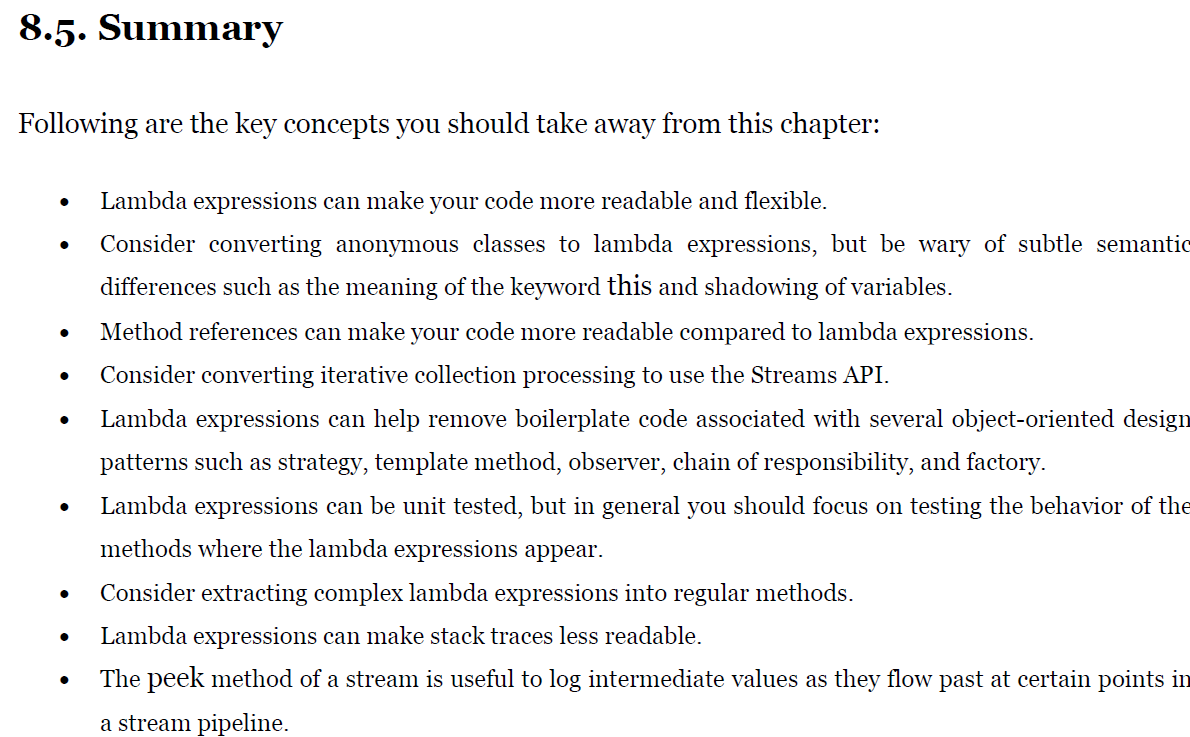


**Chapter 8. Refactoring, testing, and debugging**

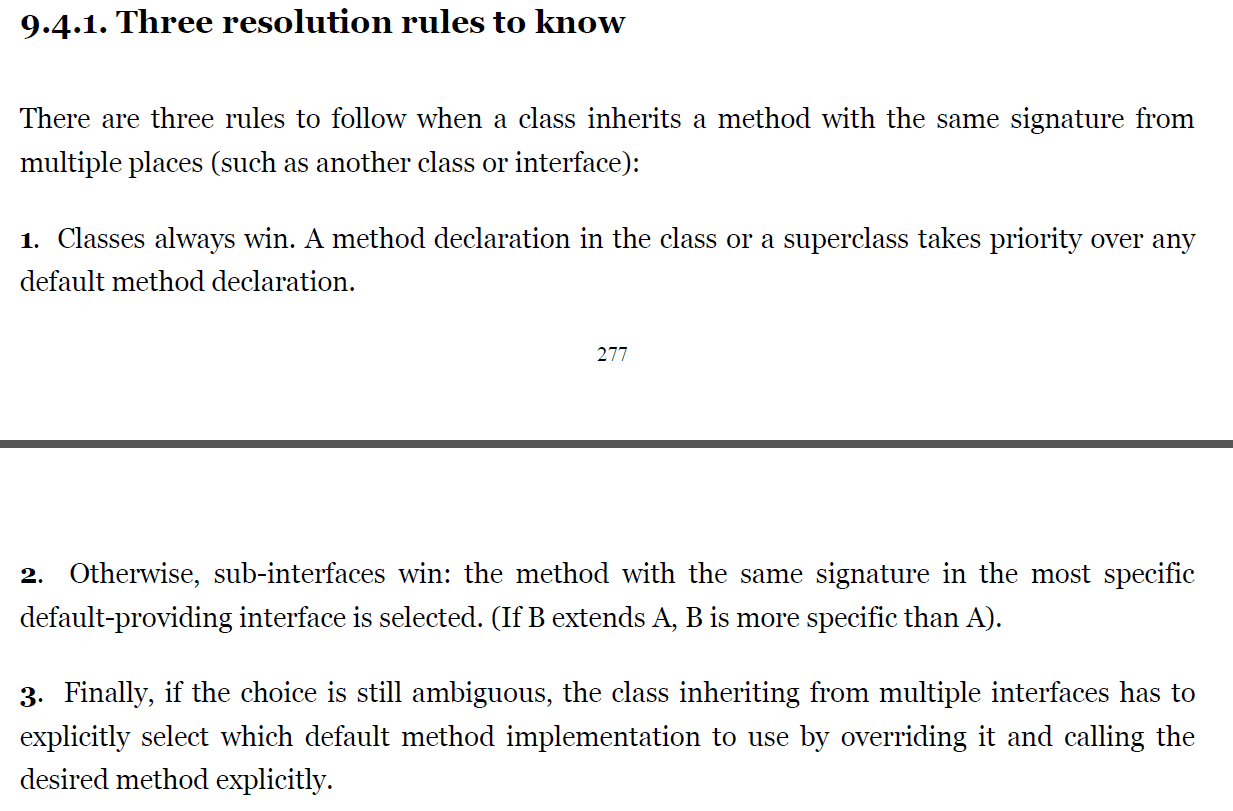


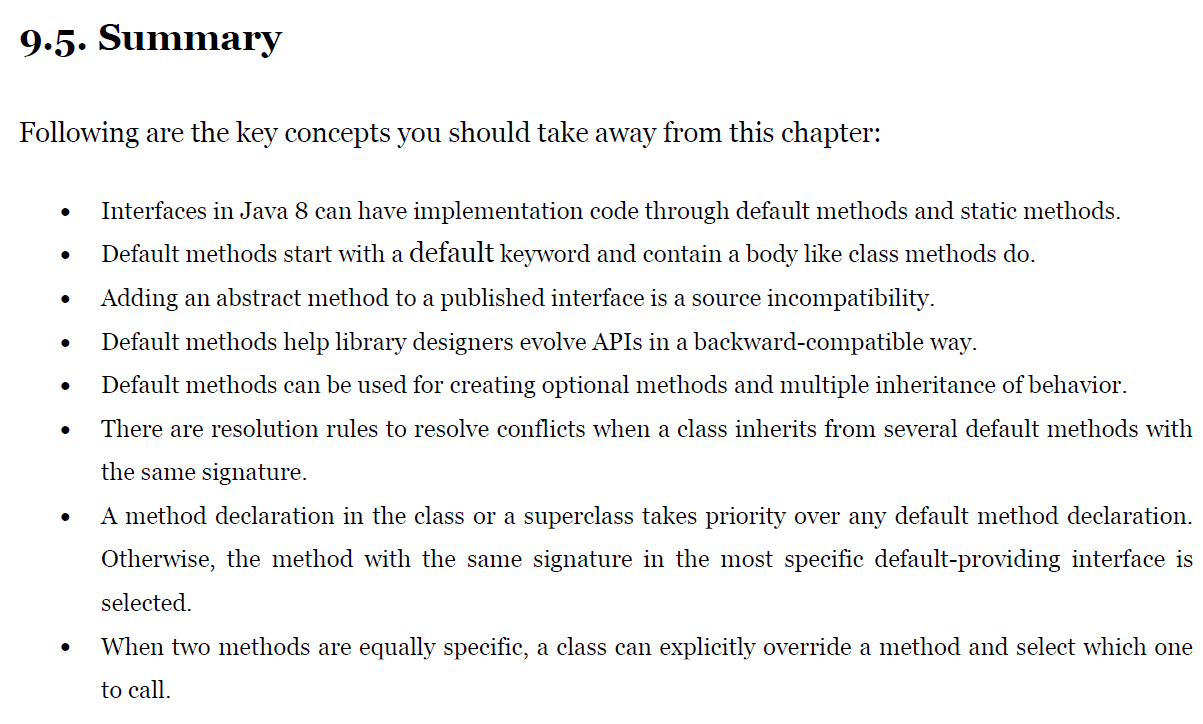






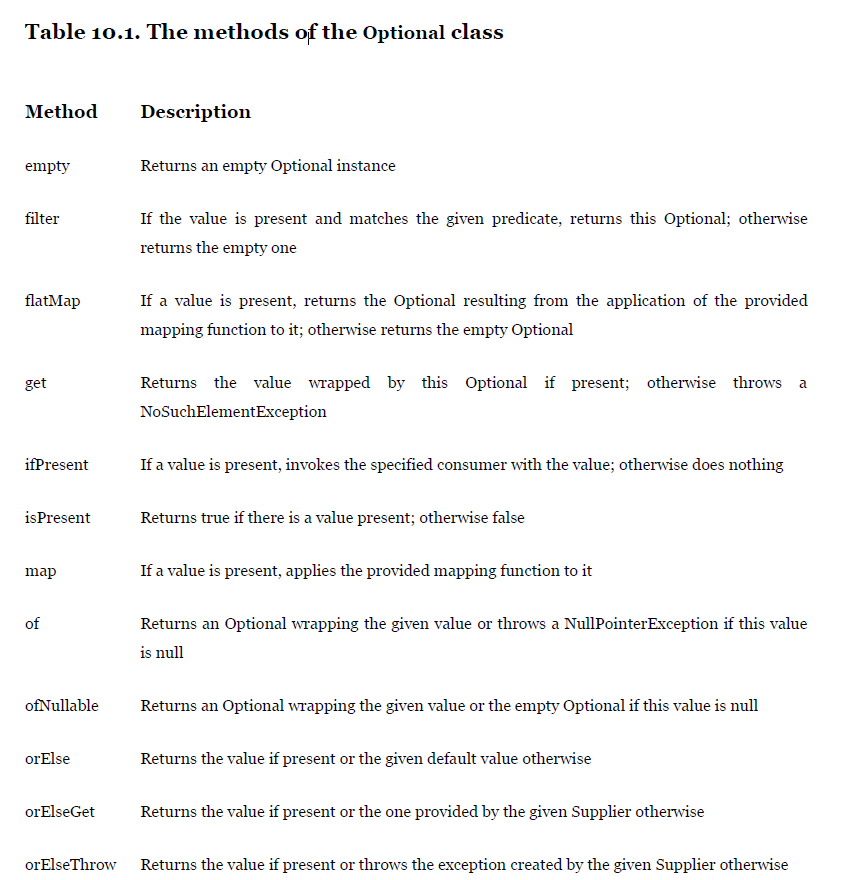
**Chapter 9. Default methods**





**Chapter 10. Using Optional as a better alternative to**

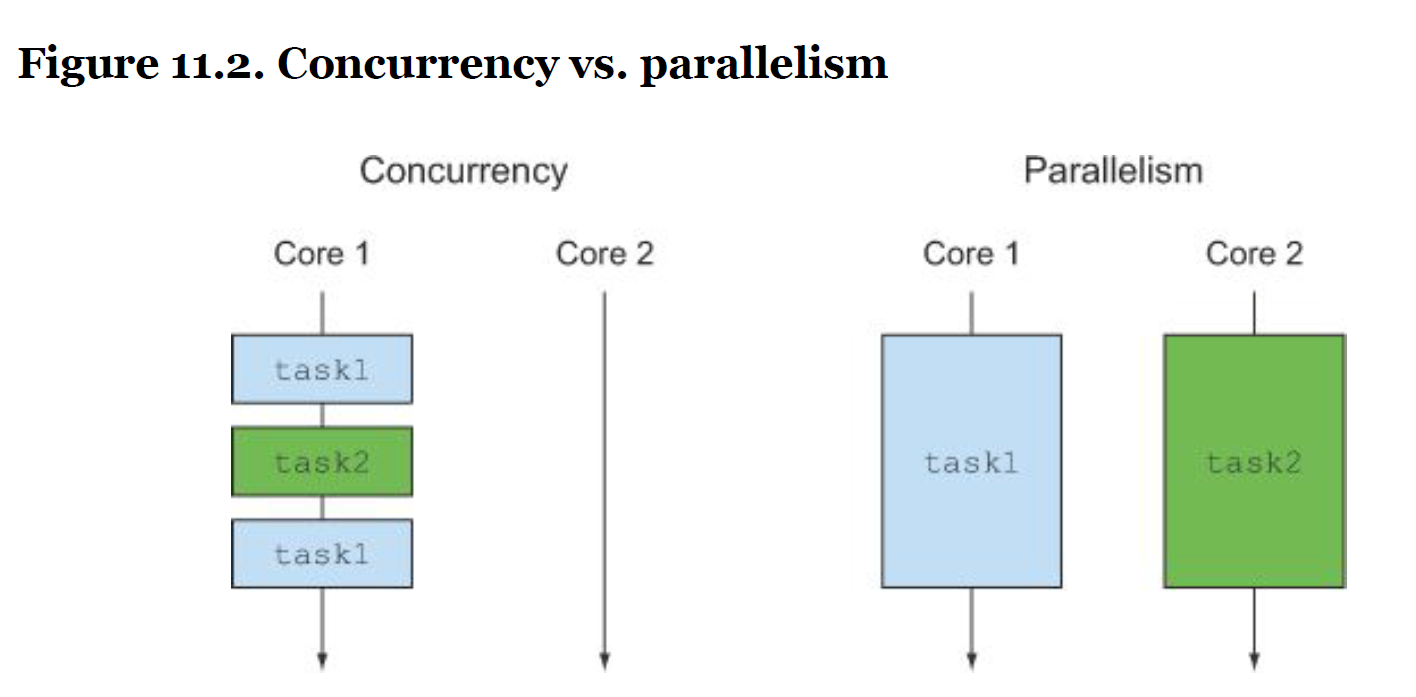
**Null**

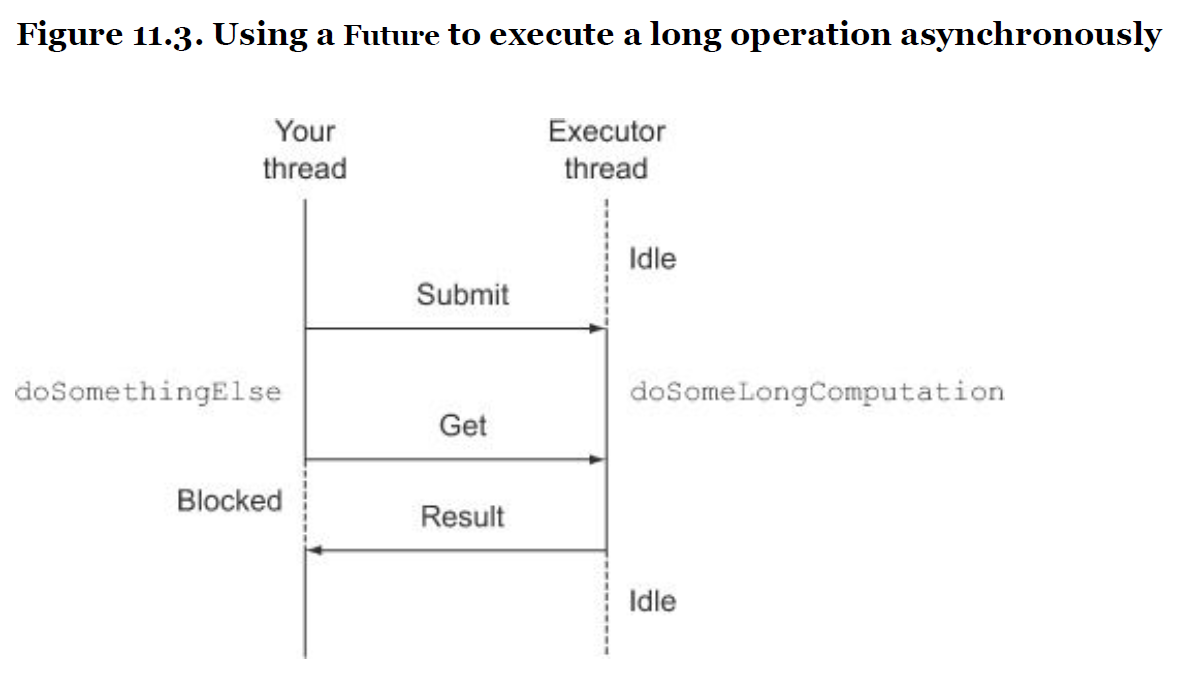


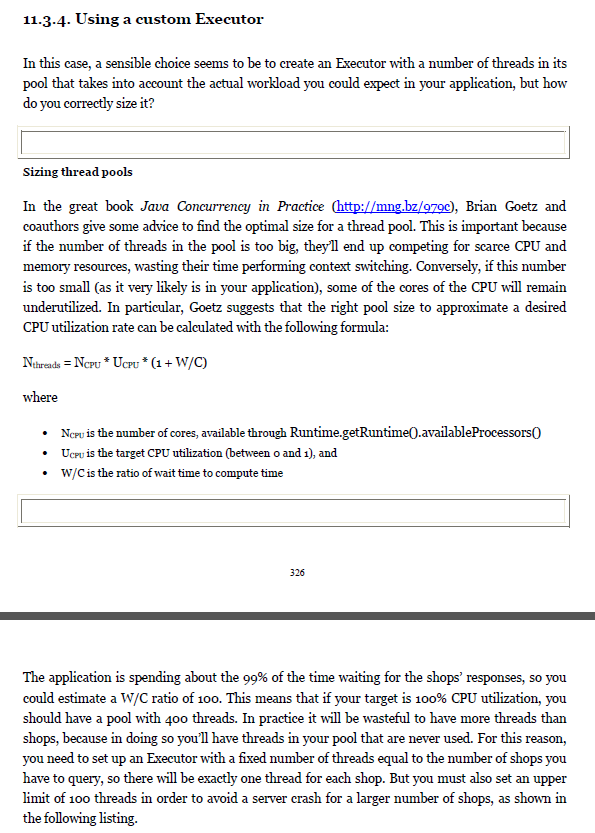


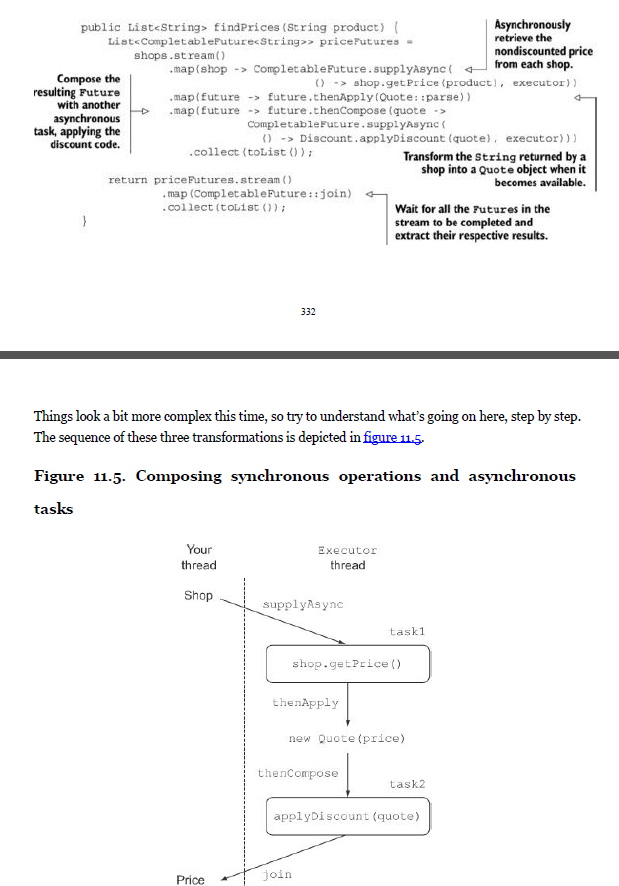
**Chapter 11. CompletableFuture: composable**

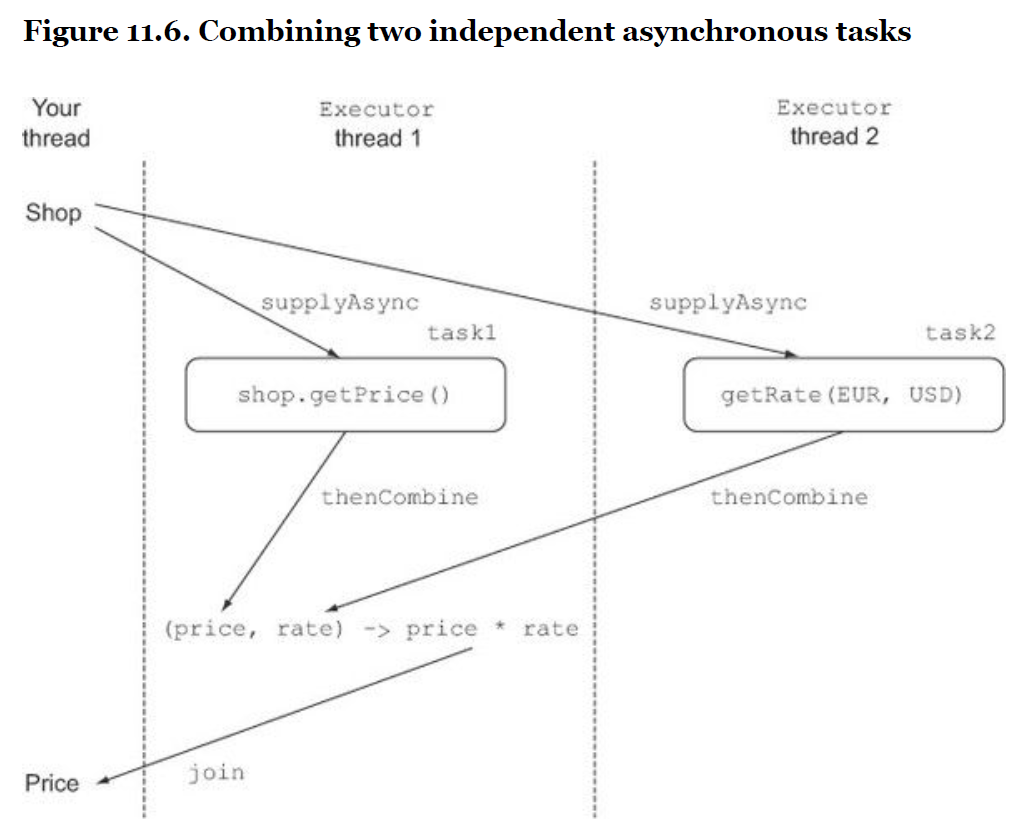
**asynchronous programming**

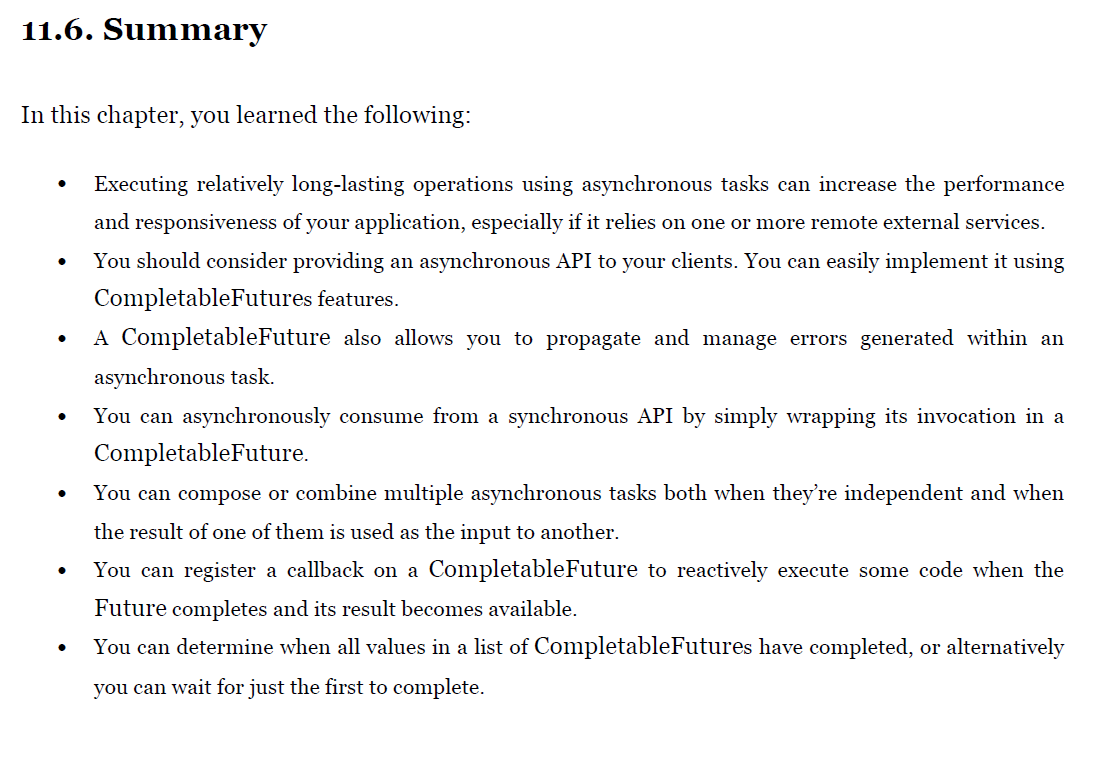












**Chapter 12. New Date and Time API**

