Java 8 tips Documentation

Release 1.0

Sanit

Aug 27, 2017

Contents

Contents:

Quick Introduction

Java 8 launched on 18th March 2014 and it was a next major release after jdk5. It came up with large set of scintillating features that has won the attention of most of the java programmers. It has enhanced various java components like runtime environment, compiler, lexical parser, memory management, command line tools and many more. Java 8 will improve programmer's coding experience with its enticed features of declarative programming, passing code as an argument, method reference, optional for handling null etc. It will assure you to write codes that will be more precise, objective driven and highly readable.

Passing methods as parameter removes verbosity from the code and in fact increases reusability, Streams helps in writing SQL like syntaxes, parallelization that is almost free:- *speed of the execution with efficient use of modern computers having multicore processors*, handling nullable values using Optional and many more. Initially stuffs will be little confusing but once you used to it, you will be reluctant to write code with out using it. Let's look into the below usecase and understand why java 8 is different then all the releases.

Suppose we are trying to find the highest salary paid in each technology of a XYZ company. Before Java 8 the typical implementation could be

```java
public Map<String, Double> method2(List<Employee> list) {
    Map<String, List<Employee>> temp = new HashMap<>();
    for (Employee e : list) {
        temp.putIfAbsent(e.getTechnology(), new ArrayList<>());
        temp.get(e.getTechnology()).add(e);
    }

    Map<String, Double> map = new HashMap<>();
    for (Entry<String, List<Employee>> ent : temp.entrySet()) {
        double max = 0;
        for (Employee e2 : ent.getValue()) {
            max = Double.max(max, e2.getSalary());
        }
        map.put(ent.getKey(), max);
    }
    return map;
}
```

Let's rewrite this code snippet in Java 8 way.

```
Map<String, Double> map = list.stream().collect(
    groupingBy(Employee::getTechnology,                          -- Grouping on␣
→technology
        mapping(Employee::getSalary,                             -- Scale to salary␣
→from Employee object
            collectingAndThen(maxBy(Comparator.naturalOrder()),  -- Find the maximum␣
→among them
                Optional::get))));
```

Isn't it great. I just said "group on technologies" then extract salary from the employee object and finally get me the highest value from each group. Here my code is objective oriented and easy understandable. If you look into the first approch we are using a temporary intermediate map just to keep grouped data and then process it to find the desired result. Every time you implement this kind of funtionality, you will write these boilerplate codes, but now java does these extra coding and returns result to you. You still might be thinking older approach is good because of the confusions and we are not ready to think in functional programming way.

Below are the topics we will investgate in this tutorial. I am excited to walk you through these features, so let's get started.

- What is a functional programming and Functinal interface?

- java.util.stream.Stream and its operations

- Collector and Collectors

- Forkjoin Pool and Spliterators

- How to use parallel streams?

- Nullable values with Optional

- java.util.time package

Behavior Parameterization

Behavior parameterization is the ability of a method to receive multiple different behavior as its parameter and use them internally to accomplish the task. It let you to make your code more adaptive to changing requirements and saves the engineering effort from writing similar piece of code here and there.

If you have come across some of the behavioral design patterns like *Strategy Pattern* then you know we create set of similar algorithms and choose required one at run time to deal with a certain problem scenario. This type of techniques facilitate to add new behaviors in future. Let's look into a problem statement to understand better.

Suppose a company XYZ is trying to group its employees based on certain criterias like proficiency level, technology type, gender etc or new criterias can be added in future. So to solve this problem we will create family of grouping algorithms as described below.

```java
interface Groupable {
    public String findGroup(Employee e);
}

class GroupByExperience implements Groupable {

    @Override
    public String findGroup(Employee e) {
        return e.yearsOfExpr >= 7 ? "Expert" :
            e.yearsOfExpr >= 3 ? "Intermediet" : "Fresher";
    }
}

class GroupByTechnology implements Groupable {

    @Override
    public String findGroup(Employee e) {
        Map<String, List<String>> mapping = new HashMap<String, List<String>>() {
            {
                put("Front-end", Arrays.asList("AngularJS", "ExtJS"));
                put("Middleware", Arrays.asList("Java", ".Net"));
                put("Back-end", Arrays.asList("Oracle", "MySQL", "PostgreSQL"));
            }
```

```
24              };
25
26              for (Entry<String, List<String>> entry : mapping.entrySet()) {
27                  if (entry.getValue().contains(e.technology)) {
28                      return entry.getKey();
29                  }
30              }
31              return "Others";
32          }
33      }
```

Based on our purpose we are passing the required behaviors to the grouping function which is just creating groups.

```
public Map<String, List<String>> group(List<Employee> list, Groupable behavior) {
    Map<String, List<String>> map = new HashMap<>();
    for (Employee e : list) {
        String group = behavior.findGroup(e);
        map.putIfAbsent(group, new ArrayList<>());
        map.get(group).add(e.name);
    }
    return map;
}
```

Great... We solved the problem, as and when new requirements comes we just need to provide some other implementation classes. But from begining we are talking, one of our main objective is to remove verbosity from the code as well as maintain the understandability. If you look into the `GroupByExperience` class, the behaviour is of one liner but still complete class has been written. Another way can be writting Anonymous classes which some what reduces these boilerplate codes but not to the great extent.

Just think, if the interface `Groupable` was given by java SDK itself and we were written only the method and passed to the grouping function, then the code will be clearer and more flexible. Some of the interpreted langauges like python, JavaScript etc support passing of method as parameter to the calling function, similarly Java 8 has also started supporting it with the help of *Functional Interfaces* and *Lambdas*. Most of us already aware of Lambdas which is a very well-known concept that exist from the begining of languages like python. Don't worry about them now, we will slowly have deep drive into it.

Lambdas

In previous chapter we thought of removing `GroupByExperience` class, only the method body should be given to our *group()* function and Lambdas are the best examples of implementing them. Lambdas give us the ability to encapsulate a single unit of code block and pass on to another code. It can also be considered as anonymous function which doesn't have any function name but has list of parameters, function body, returns result and even throws exceptions. Below is the code statement if we convert our `GroupByExperience` class to a lambda expression.

```
(Employee e) -> { return e.yearsOfExpr >= 7 ?
                "Expert" : e.yearsOfExpr >= 3 ? "Intermediet" : "Fresher"; }
```

Basically *Lambda* has 3 parts.

1. **A list of parameters** : In above example "Employee e"

2. **Function body** : The behavior (right hand side of arrow)

3. **An arrow** : Separator between parameter list and function body

Note: Lambda syntax follows some of the below rules.

- Parameter types are optional.

- If you have single parameter then both parameter type and parenthesis are optional.

- If you have multiple parameters, then they should be enclosed with in parenthesis.

- For multiple statements in function body should be enclosed with in courly braces.

- If lambda body exnclosed inside courly braces then `return` keyword is required in case your behavior returns value.

With applying above rules our `GroupByExperience` class can be writtten in following ways.

```
e -> { return e.yearsOfExpr >= 7 ?
        "Expert" : e.yearsOfExpr >= 3 ?
                "Intermediet" : "Fresher"; }
```

```
e -> e.yearsOfExpr >= 7 ?
        "Expert" : e.yearsOfExpr >= 3 ?
                "Intermediet" : "Fresher"
```

Below are some more examples of lambda expressions.

1. ***BiConsumer<List, Integer> addIntoList = (List list, Integer element) -> list.add(element);*** Adding an element to a given list.

2. ***Predicate<Employee> isJavaEmp = e -> "Java".equals(e.technology);*** Checking an employee is is from Java technology.

3. ***Supplier<Integer> uniqueKey = () -> new Random().nextInt();*** Generate a unique number with the help of generator.

We saw couple of more lambda expressions above but what are these left hand side classes (BiConsumer, Predicate etc). If you remember in behavior parameterization chapter I mentioned, what if `Groupable` interface were given by java it self then we don't have to write our own interfaces or abstract classes to give different different implementations. Java 8 has already came up with bundle of general purpose functional interfaces which are included as part of JDK and we are going to visit them soon.

Now we have some ideas on how lambdas look like and their syntaxes. Just look into the below two lambda expressions.

```
1. Runnable runnable   = () -> "I love Lambdas".length();
2. Callable<Integer> ca = () -> "I love Lambdas".length();
```

Both of expressions look similar except the left hand side target types. You would be thinking how does it possible to assign same object to two different types of references. Is the Callable extends Runnble or vice-versa? The answer is a big NO, this is possible due to the *type inference* feature which decides the target type depending upon the context where it is used.

Type Inferencing

There has been much more improvent in compiler intelligence level that it takes advantage of target typing to infer the type parameters of a generic method invocation. When inially Generics introduced in JDK 1.5, the type of generic was mandatory in both side of the expression. For example:

List<String> list = new ArrayList<String>();

But in JDK 1.7 right hand side generic type become optional by changing it to the diamond(<>) operator where type is evaluated from it's left hand side target type declaration. Still there were some limitations in generic type evaluation.

```
1. List<String> l = Arrays.asList();


2. List<String> list = new ArrayList<>();
   list.addAll(Arrays.asList());
```

If you compile above code in JDK 1.7, then the statement-1 will be compiled successfully but not statement-2 and it will generate `The method addAll(Collection<? extends String>) in the type List<String> is not applicable for the arguments (List<Object>)` error message.

So what really happened in statement-2 where as both of the statements looks similar. Just look into the signature of above used methods.

Method Signatures
public static <T> List<T> asList(T... a)
public boolean addAll(Collection<? extends E> c)

The asList() is a type safe method which is able to infer its return type based on the given direct target type but in addAll() case, compiler didn't have idea to deduce the type when applied on method parameter as target type and asList() method returned List<Object> that is incompatible with List<String> reference. Java 8 has enhanced this *type inferencing* technique to deal with such wiered scenarios. Now let's see how type inferencing works in lambda expressions.

The type of lambda is deduced from the context where it is used. If we take our earlier example of Runnable and Callable, the signature of lambda expression matches with the singature of run() and call() methods. Runnable class run() method neither accept any argument nor return anything. Our lambda expression `() -> "I love Lambdas".length()` also doesn't supply any parameter.

```
For run() method fully described lambda expression is
() -> {
    I love Lambdas".length();
}


and for call() it is
() -> {
    return I love Lambdas".length();
}
```

Java compiler always looks for a matching functional interface to associate with the lambda expression from it's surrounding context or target type. Compiler expects you to use lambda expresssion in following places such that it can determine the target type.

- Variable declarations

- Assignment statements

- Return statements

- Method or constructor arguments

- Lambda expression bodies

- Ternary expressions, ?: etc

For method or constructor arguments, the compiler determines the target type with two other language features: *overload resolution* and *type argument inference*. Look into the below code snippet.

```
public static void main(String[] args) throws Exception {
    execute(() -> "done");  // Line-1
}

static void execute(Runnable runnable) {
    System.out.println("Executing Runnable...");
}

static void execute(Callable<String> callable) throws Exception {
    System.out.println("Executing Callable...");
    callable.call();
}
```

```
/* static void execute(PrivilegedAction<String> action) {
    System.out.println("Executing PrivilegedAction...");
    action.run();
} */



Output: Executing Callable...
```

Here we have two overloaded methods: Runnable and Callable. When you call the execute method with the mentioned lambda, the `execute(Callable)` will be called because call() method can return something. Now just uncomment *execute(PrivilegedAction)* method and try to reexecute and this time you will get compilation error: *The method execute(Callable<String>) is ambiguous for the type Lambdas*. The reason is both the last two execute() methods are capable to return and compiler found the ambiguous methods. So to resolve this you have to explicitly type cast the lambda expression as below.

> *execute((Callable<String>) (() -> "done"));*

Accessing outer scope variables

Some of the rules applicable for anonymous classes are also applicable to Lambdas:

- Lambda has access to members of its enclosing scope. (see line-1)

- Like nested class or anonymous class, it can also shadows any other declarations in the enlosing scope that is of same name. (see line-2)

```
public class LambdaFeatures {
    private int x = 10;

    public void example() {
        Consumer<String> funcInterface = str -> {
            System.out.println("x= " + x);   // Line-1

            int x = 50;                       // Line-2
            System.out.println("x= " + x);
        };
    }
}

Output: x= 10
        x= 50
```

Restrictions in Lambdas

Lambda has some restrictions:

- You can't declare any static or non-static initializers.

- It cann't access local variables in its enclosing scope that are not defined final or effectively final. This restriction exists with anonymous class also. Let's discuss why is this limitation with following code snippet.

```
public class LambdaFeatures {
    int y = 50;
```

```
    public static void main(String[] args) throws Exception {
        int x = 50;

        Thread tt = new Thread() {
            public void run() {
                System.out.println("MyThread start.");

                Thread.sleep(1000L);

                System.out.println("MyThread end. x=" + x);
            }
        };

        t.start();

        x++;
        System.out.println("main end");
    }
}
```

Local variables stored in the stack where as instance variables stored in heap. In the above code snippet main thread declares variable "x" and also creates a Thread which is trying to use this x variable. As we know local variables will be stored in the local stack (here stack of main) and when thread "tt" will be created it will executed separate to main thread. There might be chances that main will be completed first and the stack will be released before thread tt trying to use it. So if variable is declared final, them lambda will a copy of it and use whenever require.

Where to use Lambdas

We have discussed enough on lambdas and anonybmous classes. Let's discuss the scenarios where should we use them.

- **Anonymous class:** Use it whenever you want to declare some additional fields or methods which lambda cann't do.

- **Lambda:**

    - Use it if you want to encapsulate a single unit of behavior and pass to some other code. For example: performing certain operation on each element of collection.

    - Use it if you need a simple instance of a functional interface and none of the preceding criteria apply (for example, you do not need a constructor, a named type, fields, or additional methods).

Functional Interfaces

In java 8 context, *functional interface* is an interface having exactly one abstract method called *functional method* to which the lambda expression's parameter and return types are matched. Functional interface provides target types for lambda expressions and method references.

The java.util.function contains general purpose functional interfaces used by JDK and also available for end users like us. While they are not the complete set of funtional interfaces to which lambda expressions might be applicable, but they provide enough to cover common requirements. You are free to create your own functional interfaces whenever existing set are not enough.

The interfaces defined in the this package are annotated with FunctionalInterface. This annotation is not the requirement for the java compiler to determine the interface is an *functional interface* but it helps the compiler to identify the accidental violation of the design intent. Basically I would say this annotation will be very much useful for us while creating our custom functional interfaces.

@FunctionalInterface rules

As discussed @FunctionalInterface is a runtime annotation that is used to verify the interface follows all of the rules that can make this interface as *functional interface*. Below are some of the rules from them:

- Interface must have exactly one abstract method.

- It can have any number of default methods because they are not abstract and implementation is already provided by same.

- Interface can declares an abstract method overriding one of the public method from `java.lang.Object`, that still can be considered as *functional interface*. The reason is any implementation class to this interface will have implementation for this abstract method either from super class (bare minimum java.lang.Object) class or defined by implementation class it self. In the below example `toString()` method declared as abstract which will be implemented in its concrete implementation class or at last derived from `java.lang.Object` class.

Below code snippet is a simple example of functinal interface.

```
@FunctionalInterface
public interface MyFunctionalInterface {

    public abstract void execute();

    @Override
    String toString();

    default void beforeTask() {
        System.out.println("beforeTask... ");
    }

    default void afterTask() {
        System.out.println("afterTask... ");
    }
}
```

Enough prose here, now see some of the basic functional interfaces defined in this package.

Predicate<T>

java.util.function.Predicate has a boolean-valued function that takes an argument and returns boolean value.

Class definition

public interface Predicate<T> {
        boolean test(T t); // functional descriptor
}

Simple usecase of Predicate can be identifying all odd numbers from a given set or finding java employees from list of employees etc.

Example:

```
public class PredicateTest {

    public static void main(String[] args) {
        Predicate<Integer> oddNums = (num -> num % 2 == 0);
        Predicate<Integer> positiveNums = (num -> num > 0);

        Integer[] array = IntStream.rangeClosed(-10, 10).boxed().
→toArray(Integer[]::new);

        filter(array, oddNums);
        filter(array, positiveNums);
    }

    public static <T> List<T> filter(T[] array, Predicate<T> predicate) {
        List<T> result = new ArrayList<>();
        for (T t : array) {
            if (predicate.test(t))
                result.add(t);
        }
        return result;
```

```
        }
}
```

Here if you see *filter* method accepts a Predicate which is calling its test() method to extract the desired result. Later if you want find all primary numbers then you prepare another predicate and pass it to filter method.

It has couple of default methods which you can use it:

Method	Description	Example
and(Predicate<? super T> other)	Returns a composite predicate that represents logical AND of two predicates (P1 AND P2)	Predicate<Integer> positiveOdd = positiveNums.and(oddNums)
or(Predicate<? super T> other)	Returns a composite predicate that represents logical OR of two predicates (P1 OR P2)	Predicate<Integer> positiveOrOdd = positiveNums.or(oddNums)
negate()	Returns a predicate that represents the logical negation of this predicate.	Predicate<Integer> negative = positiveNums.negate();

Consumer<T>

java.util.function.Consumer accepts an argument and returns no result.

Class definition

public interface Consumer<T> {
        void accept(T t);
}

Simple usecase can be persisting elements of a collection into DB or serializing them or printing on the console.

```java
public class ConsumerTest {

    public static void main(String[] args) {
        Consumer<Employee> printOnConsole = (e -> System.out.print(e));
        Consumer<Employee> storeInDB = (e -> DaoUtil.save(e));

                    forEach(empList, printOnConsole);
                    forEach(empList, storeInDB);
                    forEach(empList, printOnConsole.andThen(storeInDB));
    }


    static <T> void forEach(List<T> list, Consumer<T> consumer) {
        int nullCount = 0;
        for (T t : list) {
            if (t != null) {
                consumer.accept(t);
            } else {
                nullCount++;
            }
        }
        System.out.printf("%d null entries found in the list.\n", nullCount);
    }
}
```

Consumer has also one default method called *andThen(Consumer<? super T> after)* which returns a composite consumer where second consumer will be executed after execution of first one. If the first consumer throws any

---

exception then the second consumer will not be executed because non of the functional interfaces provided by JDK handles any exception.

Function<T, R>

java.util.function.Function accepts an argument and returns result.

Class definition

public interface Function<T, R> {
    R apply(T t);
}

A usecase of *Function* can be extracting employee name from Employee class or deriving primary ids from given object etc.

```java
public class FunctionTest {

    public static void main(String[] args) {
        Function<Employee, String> empPrimaryId = (emp -> emp.getEmployeeId());
        Function<Department, String> deptPrimaryId = (dept -> dept.getLocationCode()
→+ dept.getName());

        toMap(employeeList, empPrimaryId);
        toMap(deptList, deptPrimaryId);
    }

    static <T, R> Map<T, R> toMap(List<T> list, Function<T, R> func) {
        Map<T, R> result = new HashMap<>();
        for (T t : list) {
            result.put(t, func.apply(t));
        }
        return result;
    }
}
```

Function has couple of default and static methods:

Method	Description
compose(Function<? super V, ? extends T> before)	Returns a composed function that first applies the before function to its input, and then applies this function to the result.
andThen(Function<? super R, ? extends V> after)	Returns a composed function that first applies this function to its input, and then applies the after function to the result.
static <T> Function<T, T> identity()	Returns a function that always returns its input argument. Basically it is a helper method that used in Collector implementation that we will look later.

Below code snippet shows an example of composed function `andThen()`.

```java
public class ComposedFunctionExample {

    /**
     *  Find the Addrees of given employee from database and return pincode
     */
    public static void main(String[] args) {
        Function<String, Address> first = empid -> EmployeeService.getEmployeesData().
→get(empid);
        Function<Address, Integer> second = addr -> addr.pincode;
        extract("E101", first, second);
```

```
    }

    static <T, R, U> U extract(T input, Function<T, R> first, Function<R, U> second) {
        return first.andThen(second).apply(input);
    }
}
```

It has two subclasses whose type of operand and return types are of same type.

- **UnaryOperator<T>:** This represents an operation on a single operand that produces a result of the same type as its operand. The simple usecase could be calculating square of a number.

  Function descriptor signature: `T apply(T t)`

  Example: UnaryOperator<Integer> square = (Integer in) -> in * in;

- **BinaryOperator<T>:** This represents an operation upon two operands of the same type, producing a result of the same type as the operands. The simple usecase could be calculating sum of two numbers.

  Function descriptor signature: `T apply(T t1, T t2)`

  Example: BinaryOperator<Integer> sum = (i1, i2) -> i1 + i2;

Supplier<T>

java.util.function.Supplier doesn't accept any argument but returns a result.

Class definition

public interface Supplier<R> {
    R get();
}

A simple usecase of Supplier can be generating unique numbers using various algorithms.

```java
public class SupplierTest {

    public static void main(String[] args) {
        Supplier<Long> randomId = () -> new Random().nextLong();
        Supplier<UUID> uuid = () -> UUID.randomUUID();

        Trade trade = new Trade();
        populate(trade, randomId);
        populate(trade, uuid);
    }

    static <R> void populate(Trade t, Supplier<R> supplier) {
        t.tradeDate = new Date();
        t.tradeId = (String) supplier.get();
        t.location = "XYZ Hub";
    }

    static class Trade {
        String tradeId;
        Date tradeDate;
        String location;
    }
}
```

There is another variant of functional interfaces that starts with **Bi**: BiConsumer, BiFunction, BiPredicate etc which accept two input arguments of same or different reference types. These are helper interfaces used when working with tasks expecting two input arguments as an example `list.add(element)`. There is no functional interfaces which accepts more than two input parameters, but still you can deal with such problems by wrapping all inputs to a single container.

Hint: Suppose you want to replace a CharSequence with another CharSequence within a string. Here you have three input parameters: *original string, search string, replace string*. So you can write them in following ways.

- Function<String[], String> f1 = arr -> arr[0].replaceAll(arr[1], arr[2]);

- BiFunction<String, String[], String> f2 = (str, arr) -> str.replaceAll(arr[0], arr[1]);

Primitive Functional Interfaces

We visited couple of functional interfaces which are defined as generic types. Generic types are always reference type which has extra cost associated with it called *Boxing* and *Unboxing*. Reference types are generally a wrapper around primitive types and stored in heap. Therefore, takes extra space. You might not bother about more space taking though cost of hardware is decreased a lot in last decade, but what about the execution time. When you operate on primitive types, your input and expected return type both are primitives but internally due to generics it boxes your input, does the operation then unboxes the result and returns it. So here the boxing and unboxing is an extra effort that takes phenomenon time which is useless for your purpose. Let's see an example.

```java
public class PrimitiveFunc {

    public static void main(String[] args) {
        int[] arr = IntStream.range(1, 50000).toArray();
        BinaryOperator<Integer> f1 = (i1, i2) -> i1 + i2;
        IntBinaryOperator f2 = (i1, i2) -> i1 + i2;

        RunningTime.calculate((Consumer<Void>) v -> reduce1(arr, f1));
        RunningTime.calculate((Consumer<Void>) v -> reduce2(arr, f2));
    }

    static int reduce1(int[] arr, BinaryOperator<Integer> operator) {
        int result = arr[0];
        for (int i = 1; i < arr.length; i++) {
            result = operator.apply(result, arr[i]);  // Boxing and Unboxing here
        }
        return result;
    }

    static int reduce2(int[] arr, IntBinaryOperator operator) {
        int result = arr[0];
        for (int i = 1; i < arr.length; i++) {
            result = operator.applyAsInt(result, arr[i]);
        }
        return result;
    }
}

Output:
reduce1() execution time: 0.006 secs
reduce2() execution time: 0.002 secs
```

In the above example *reduce* methods calculating sum of a given array of numbers and output section shows their running times. `reduce2()` is 3 times faster than `reduce1()` method because it uses `IntBinaryOperator` which avoids unnecessary boxing and unboxing operations.

Java8 brings a bundle of primitive functional interfaces that deals with only three primitive types i.e. int, long and double. Basically it follows a naming conventions to identify as them:

- **XXX:** Examples are IntPredicate, IntFunction, DoubleFunction, LongFunction etc. They accept primitive inputs and returns reference type results.

- **ToXXX:** Examples are ToLongFunction, ToIntFunction etc. They accept reference type as input and returns primitive types.

- **XXXToYYY:** IntToDoubleFunction, DoubleToLongFunction are some examples of this. They accept primitive type and also return primitive types.

Note: There are little caveats in above rules:

- In case of *Supplier*, XXX type returns primitive type because Supplier doesn't accept any input.

- ToXXX and XXXToYYY are only applicable to them who returns something. Functional interfaces like *Predicate* doesn't have flavours of ToIntPredicate or LongToDoublePredicate because its return type is always boolean.

Method References

We have learnt enough to build lambda expressions to create anonymous methods. You might come across the scenarios where your lambda expression can contain just one line of code that calls an existing method. In such scenario lambda expressions will look like:

- Function<String, Integer> func = str -> str.length();

- Supplier<Address> sup = () -> emp.getAddress();

Though java8 talks about removing boiler-plate codes, there is an efficient way called *method references* to build these lambdas which will be more clear and readable. If we rewrite above two lambda expressions using method reference technique then the representations will be `String::length` and `emp::getAddress`. These representation clearly says we are trying to call length method of a string in first case and getAddess in the second.

Syntax: <target reference> **::** <method name>

Above is the syntax for creating method references where the target reference will be placed before the delimeter **::** and then the name of method. There are three kinds of method references exists.

- **Reference to static method:** `Consumer<List<Integer>> c = Collections::sort;` is an example of method reference for static methods. Compiler will automatically consider it as `(list) -> Collections.sort(list)`. Here the target type will be the class name that contains the static method.

- **Reference to an instance method of a particular object:** If you have an object reference then you can call its method like `list::add` which is very similar to `(list, ele) -> list.add(ele)`. Here the target type will be object reference.

- **Reference to an instance method of an arbitrary object of a particular type:** This type of method references are little confusing. If you look into the previous example `String::length`, usually length() method is called on a string reference but we have written class name "String" as like it is a static method. When we use method references they also go through similar checks as lambda expression goes. Compiler

will try to match the method reference with any of functional descriptor syntax and if matches then passes on.

Below table shows some of method references and equal lambda expressions.

Method Reference	Equivalent lambda expression
Integer::parseInt	ToIntFunction<String> f = (str) -> Integer.parseInt(str)
Collections::sort	BiConsumer<List, Comparator<Trade>> f = (list, comp) -> Collections.sort(list, comp)
String::toUpperCase	UnaryOperator<String> f = (str) -> str.toUpperCase()
UUID::randomUUID	Supplier<UUID> f = () -> UUID.randomUUID()
empDao::getEmployee	Function<String, Employee> f = (empid) -> empDao.getEmployee(empid)

Important: There are two things you should be aware of before writing method references.

1. Method reference should not contain paranthesis after method name otherwise it will represent a method invocation that will lead to compilation error.

2. It is difficult to write lambdas or method references until and unless you know the signature of the method you are looking for.

Constructor as method reference

As you know constructors are kind of special methods, method reference will also applicable to constructors. Syntax of method reference for constructor is same as static method. Below are some of examples of method references for constructors.

Constructor Type	Lambda Representation
Zero-argument	Supplier<Employee> s = () -> new Employee()
One-argument	Function<String,Employee> f = (id) -> new Employee(id)
Two-argument	BiFunction<String, String> f = (id, name) -> new Employee(id, name)

In the above examples you can clearly see, lambda expression of invoking zero argument constructor matches with functional descriptor of Supplier, similarly one-argument constructors matches to Function and two argument is with BiFunction.

Syntax: ClassName **::** new

Examples: Employee :: new, ArrayList :: new

The constructor reference for all the above lambdas are `Employee::new` and type of the constructor invocation will be decided based on the target type. To understand it better we will see a usecase whose goal is to return a collection of unique ids but the collection type will be supplied as method argument.

```java
public class ConstructorReference {

    public static void main(String[] args) {
        ArrayList<String> a = method(ArrayList::new);
        TreeSet<String> t = method(TreeSet::new);
    }

    static Collection<String> method(Supplier<Collection<String>> container) {
        Collection<String> c = container.get();
        for (int i = 0; i < 5; i++)
            c.add("ID:" + UUID.randomUUID().toString());
        return c;
    }
}
```

Introduction to Streams

Streams are one of the bigwig among java8's released features that let you write codes in declarative style rather than typical imperative programming technique. Declarative programming expect you to mention what you want not how to achieve them. Many of the technologies like unix, database etc are already working on this fashion. In database we write `SELECT technology, max(salary) from employee group by salary` and it returns highest salary paid in each technology. In case of unix we just combine group of commands *(ls -l | grep "search string" | sort)* and ask unix to execute the operations.

Just look into the below example.

```
public static void main(String[] args) {
    List<Trade> trades = TradeData.allTrades();
    Comparator<Trade> comparator = Comparator.comparing(Trade::getNotional);
    List<String> naTrades = trades.stream()
        .filter(trade -> Region.NA.equals(trade.getRegion()))
        .sorted(comparator)
        .map(Trade::getTradeId)
        .collect(toList());
    System.out.println(naTrades);
}
```

In the above code snippet we just created a pipeline of tasks and java 8 will prepare the execution strategy internally to process it. Here we didn't write any external `foreach` loop to traverse through all the elements and it will be internally taken care. If you wish to process trades parallely no need to write any extra milti-threaded code to do it, just replacing the *stream()* method with *parallelStream()* will handle the whole parallelism process. Don't wory about parallelism now, we will look into it later.

Note: Technically stream is a sequence of elements from a source. Source can be anything like collections, arrays, generator functions or I/O resources etc.

Stream vs Collection

Most of the time collections are one of the main source for stream to act on. Stream and collection are used togather, they don't replace each other. Streams differ from collection in several ways:

- **No storage:** Collections are typically physical set of data where as streams are a logial view that will be supplied to a pipeline of operations. Collections are about data and streams are about computations.

- **Functional in nature:** An operation on a stream produces a result, but does not modify its source. For example, if we call filtering on a stream it will return a new stream rather than removing them from the original collection.

- **Lazyness execution:** Many of stream operation like filtering, mapping etc are chained togather and executed in one shot using a terminal operation. This technique helps to create optimized execution strategy to process the operations. For example, to find first three odd numbers from a stream it doesn't go through the complete data set and halts the execution once three values found.

- **Possibly unbounded:** While collections have a finite size, streams need not. Short-circuiting operations such as limit(n) or findFirst() can allow computations on infinite streams to complete in finite time.

- **Consumable:** The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source. If the source is FileInputStream etc, then you are out of luck because inputstream will be closed once consumed and you cann't regenerate the stream.

Stream sources

In above section we saw collections and InputStream are two sources for streams. There are numerous other sources as well from where you can generate the stream.

- From a Collection via the *stream()* and *parallelStream()* methods;

- From an array via *Arrays.stream(T[])*;

- From static factory methods on the stream classes, such as *Stream.of(T[]), IntStream.range(int, int) or Stream.iterate(T, UnaryOperator)*;

- The lines of a file can be obtained from *BufferedReader.lines()*;

- Streams of file paths can be obtained from methods in *Files*;

- Streams of random numbers can be obtained from *Random.ints()*;

Apart from all of these predefined sources, you can also generate stream from your custom source using StreamSupport class. Example:

```java
public class TradePool {
    List<Trade> list;

    public Stream<Trade> stream() {
        return StreamSupport.stream(list.spliterator(), false);
    }
}
```

StreamSupport has some low-level methods which expects you to provide a spliterator that will generate stream. As of now don't worry about this spliterator, just think it is an iterator we will cover this spliterator once we are ready to go for parallelization because you need to understand ForkJoinPool better to know about it.

Stream Operations

Stream operations are broadly divided into intermediate and terminal operations that are combined to form pipeline. A stream pipeline consists of a source (such as a Collection, an array, a generator function, or an I/O channel); followed by zero or more intermediate operations such as Stream.filter or Stream.map; and a terminal operation such as Stream.forEach or Stream.reduce.
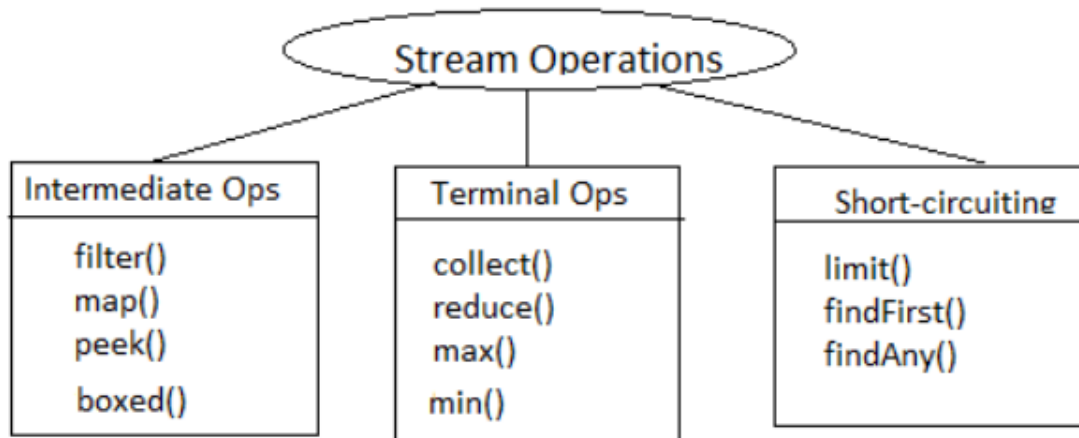
- **Intermediate Operations:** Intermediate operations helps the stream pipeline to build the execution strategy. These are lazy in nature, they don't execute until any terminal operations are invoked. They don't modify the original stream, everytime they return a new stream. Intermediate operations can again divided into stateless and stateful operations.

    - *Stateless* operations such as filter, map are processed independently of operations on other elements

    - *Stateful* operations such as sorted, distinct require to rememeber the result of operations on already seen elements to calculate the result for next element. They execute the entire input before producing the final result.

- **Terminal Operation:** Terminal operation traverse the stream and execute the pipeline of intermediate operations to produce the result. They are eager in nature. After the terminal operation is performed, the stream pipeline is considered consumed, and can no longer be used. A stream implementation may throw IllegalStateException if it detects that the stream is being reused.

Streams are also generated from infinite dataset. Some of the stream operations can be tagged as *short-circuting operations* which acts on these infinite stream or data. An intermediate operation is said to be short-circuting if applying on infinite stream should produce finite stream. As an example `new Random().ints().limit(5)` will return only 5 random numbers. A terminal operation is short-circuting if, when applying on infinite set of input should produce result in finite time. As an example `new Random().ints().filter(no -> no % 10 == 0).findAny()` will return any one random number divisible by 10.

Stream API

In the previous chapter you saw how streams are related to collections, various stream sources and kind of stream operations. In this chapter we will have an extensive look at various operations supported by stream API. java.util.stream.Stream contains numerous methods that let you deal with complex data processing queries such as filtering, slicing, mapping, finding, matching and reducing both in sequential and parallel manner. There are also primitive specialization of streams used for primitive elements and contains additional operations *min, max, sum* etc.

Filtering

Stream interface provides a method `filter` which accepts a Predicate as argument and return a stream that matches the given predicate. The predicate will be applied to each element to determine if it should be included to new stream.

> **Signature** Stream<T> filter(Predicate<? super T> p)

```
// Finding words starts with vowel
List<String> words = Stream.of("apple", "mango", "orange")
                           .filter(s -> s.matches("^[aeiou].*"))
                           .collect(toList());

:Output: [apple, orange]
```

Truncating Stream

Stream supports the `limit(n)` method accepts a numeric value and returns a new stream consisting of the elements of this stream, truncated to be no longer than maxSize in length. If the stream length is less than the given size then complete stream will be returned.

`limit` will truncate the stream from end where as there is another method called `skip(n)` will discard elements from begining.

> **Signature** Stream<T> limit(long maxSize)

Stream<T> skip(long n)

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
stream.filter(i -> i%2 == 0).limit(2).collect(toList());
stream.filter(i -> i%2 == 0).skip(1).collect(toList());
```

Consuming Stream

Stream provides two methods `peek` and `forEach` which accepts a Consumer as argument and performs the action on each element.

Signature   Stream<T> peek(Consumer<? super T> action)

void forEach(Consumer<? super T> action)

The `peek` is an intermediate operation which returns the new stream where as `forEach` is the terminal operation returns void.

```
Stream<Integer> stream = Stream.of(1, 2, -3, 4, 5);
stream.filter(i -> i%2 == 0).peek(System.out::println).toArray();
stream.filter(i -> i%2 == 0).forEach(System.out::println);
```

See also:

forEachOrdered, performs action on encontered order.

Mapping

A very common data processing idiom is to select information from a certain object. For example selecting trade id from an Trade object. Stream supports `map` method which accepts a *Function* as argument and returns a new stream consisting of the results of applying the given function to the elements of this stream.

Signature   <R> Stream<R> map(Function<? super T, ? extends R> mapper)

```
List<Trade> trades = new ArrayList<>();
trades.add(new Trade("T101", "Paul", 5000, "USD", APAC));
trades.add(new Trade("T102", "Mr Bean", 3580, "SGD", NA));
trades.add(new Trade("T103", "Simond", 2300, "CAD", EMEA))

trades.stream().map(Trade::getTradeId).collect(Collectors.toList());

Output: [T101, T102, T103]
```
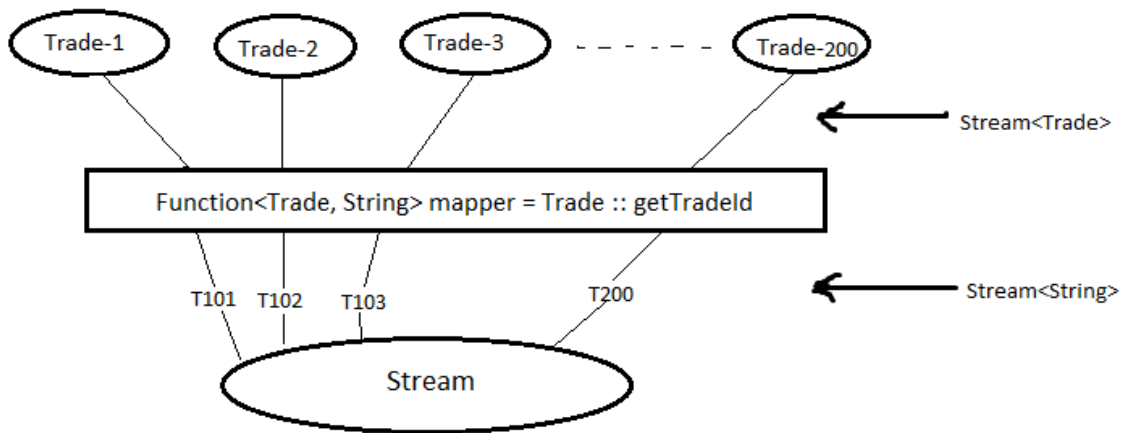
There are primitive variants of map methods `mapToInt`, `mapToDouble` and `mapToLong` that we will see later. Stream interface has method `flatMap` which returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Sometime each element of a stream will produce individual streams that will be amalgamated into single stream and *flatMap* will be used there. It might be confusing you now so let see an example where you need to find distinct words contained in a file. Here we will use `File.lines()` which will return Stream<String> where each element will represent to a single line of the file.

```
List<String> words =
        Files.lines(Paths.get("flatmap.txt"))        // Stream<String>
        .map(line -> line.split(" "))                // Stream<String[]>
        .map(Arrays::stream)                         // Stream<Stream<String>>
        .distinct()
        .collect(Collectors.toList());

System.out.println(words);
```

In the above code snippet each line will be splitted to array of words. Each array of words then passed to `Arrays.stream()` which will return Stream<String> for every line. `map(Arrays::stream)` will return Stream<Stream<String>> so our final output will be List<Stream<String>> where as our requirement is List<String>.

Now if you replace `map(Arrays::stream)` with `flatMap(Arrays::stream)` then all the elements from the each inner stream will be merged to a single outer stream.
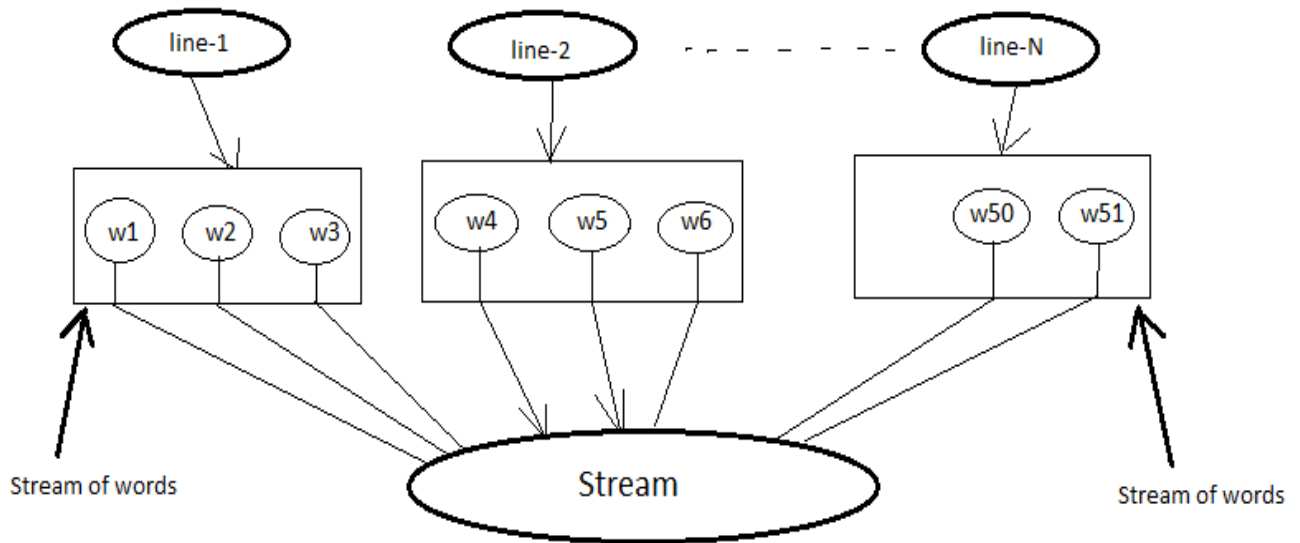
Note: 'W' represents a word.

```
List<String> words =
        Files.lines(Paths.get("flatmap.txt"))      // Stream<String>
        .map(line -> line.split(" "))               // Stream<String[]>
        .flatMap(Arrays::stream)                    // Stream<String>
        .distinct()
        .collect(Collectors.toList());

System.out.println(words);
```

Matching

Stream API provides `anyMatch`, `allMatch` and `noneMatch` short-circuiting terminal operations which takes a Predicate as argument and returns a boolean result by applying the Predicate to the elements of the stream. Predicate might not be applied to all the elements if further execution is not require.

- **anyMatch:** Returns true if any element found matching with the predicate.Predicate will not be applied to other elements if any matching found.

- **allMatch:** Returns true if all elements are matching to the given predicate.

- **noneMatch:** Returns true if none of the elements are matching to the predicate.

```
Stream.of(5, 10, 15, 20).anyMatch(i -> i % 10 == 0);
Stream.of(5, 10, 15, 20).allMatch(i -> i % 5 == 0);
Stream.of(5, 10, 15, 20).noneMatch(i -> i % 3 == 0);
```

Finding element

Stream interface has `findAny` method which returns an arbitrary element from the stream. The behaviour of this operation is nondeterministic; it is free to select any element in the stream because in case of parallelization stream source will be divided into multiple chunks where any element can be returned. It has `findFirst` method also which returns the first element of the stream.

> **Signature** Optional<T> findFirst()
>
> Optional<T> findAny()

If you see the signature of above two methods, they return an Optional object which is a wrapper describing absence or presence of the element because there might be chance that these operations were called on empty stream. Don't worry about Optional now, use get() or orElse() methods to get value from the optional.

```
Stream.of(5, 10, 15).filter(i -> i % 20 == 0).findAny().orElse(0);
Stream.of(5, 10, 15).map(i -> i * 2).findFirst().get();
```

Stream Reduction

Stream interface supports overloaded reduction operations that contineously combines elements of the stream until reduced to single output value.

Suppose I asked you to calculate sum of array of numbers, then if i am not wrong your answer would be something like below.

```
int[] arr = { 1, 2, 3, 4, 5, 6 };
int result = 0;
for (int num : arr) {
    result += num;
}
```

Now, I changed my requirement to calculate multiplication of elements of the array. So you will update your code to `result=0` and then `result *= num`. So if you notice here all the time you will have an initialization logic, an iteration and an operation on the two elements, only your intialized value and the operation varies.

To generalize these kind of tasks Stream API has provided overloaded `reduce` methods that does the same operation what we saw. If we re-write above codes then they will be

`Arrays.stream(arr).reduce(0, Integer::sum)`

`Arrays.stream(arr).reduce(1, (i1, i2) -> i1 * i2)`

- **T reduce(T identity, BinaryOperator<T> accumulator)** The reduce operation here takes two arguments:

  - identity: The identity element is both the initial value of the reduction and the default result if there are no elements in the stream. In the `reduce(0, Integer::sum)` example, the identity element is 0; this is the initial value of the sum of the numbers and the default value if no members exist in the array.

  - accumulator: The accumulator function takes two parameters: a partial result of the reduction (in this example, the sum of all processed integers so far) and the next element of the stream (in this example, an integer). It returns a new partial result. In this example, the accumulator function is a lambda expression that adds two Integer values and returns an Integer value:

- **Optional<T> reduce(BinaryOperator<T> accumulator)**

  This is almost equivalent to first reduction method except there is no initial value. Sometime you might be interested to perform some task in case stream has no elements rather than getting a default value. As an

example if the `reduce` returns zero, then we are not sure that the sume is zero or it is the default value. Though there is no default value, its return type is an Optional object indicating result might be missing. You can use `Optional.isPresent()` to check presense of result.
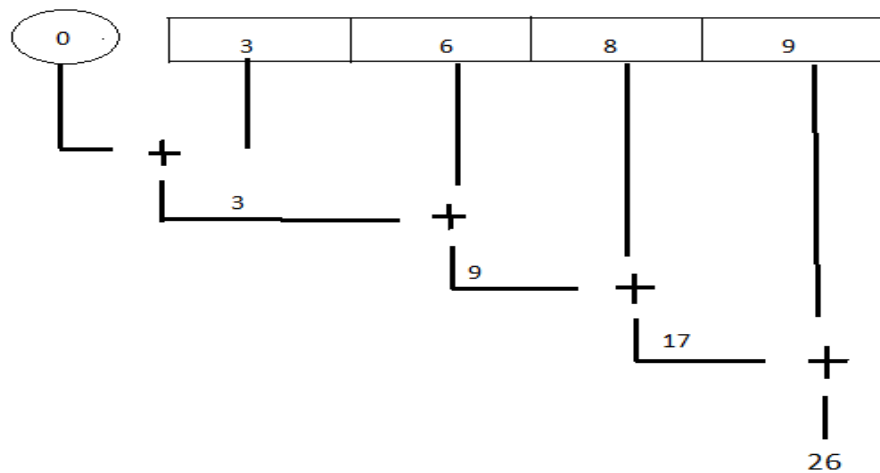

Fig. 6.1: **Sequential reduction**

- **U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)** In first two reduction operations your stream element type and return type were same means before using the reduce method you should convert your elements of type T to type U. But there is an 3 arguments reduce method which facilitates to pass elements of any type. So here *accumulator* accepts previous partial calculated result and element of type T and return type U result. Below example shows the usage of all three reduction operations.

```
// Find the number of characters in a string.
List<String> words = Arrays
        .asList("This is stream reduction example learn well".split(" "));
int result = words.stream().map(String::length).reduce(0, Integer::sum);
Optional<Integer> opt = words.stream().map(String::length).reduce(Integer::sum);
result = words.stream().reduce(0, (i, str) -> i + str.length(), Integer::sum);
```

We saw the sample use of these reduction methods so let's explore more on this 3-argument reduction operation.

```
public static void reduceThreeArgs(List<String> words) {
    int result = words.stream().reduce(0, (p, str) -> {
        System.out.println("BiFunc: " + p + "  " + str);
        return p + str.length();
    }, (i, j) -> {
        System.out.println("BiOpr: " + i + "  " + j);
        return i + j;
    });
}


output:
BiFunc: 0  This
BiFunc: 4  is
BiFunc: 6  stream
BiFunc: 12  reduction
BiFunc: 21  example
BiFunc: 28  learn
BiFunc: 33  well
```

If you have noticed accumulator function itself calculated the final result and it didn't even use the last parameter *BinaryOperator combiner* at all then what the combiner is doing here. So the answer here is parallelization. In the begining of the tutorial I told you parallelization is almost free, there will be very minimal modification (use parallelStream method) require to run your code in parallel. This is not the right time to learn parallelization but i will give you some overal idea just to get the visibility of *combiner* in this reduction operation.

In parallelization the whole input data set is splitted to multiple chunks, each chunk process individually and combine all the results at the end. So in the above example, complete word set are splitted to groups then they will calculate total number of characters in each group finally sum all these partial results.
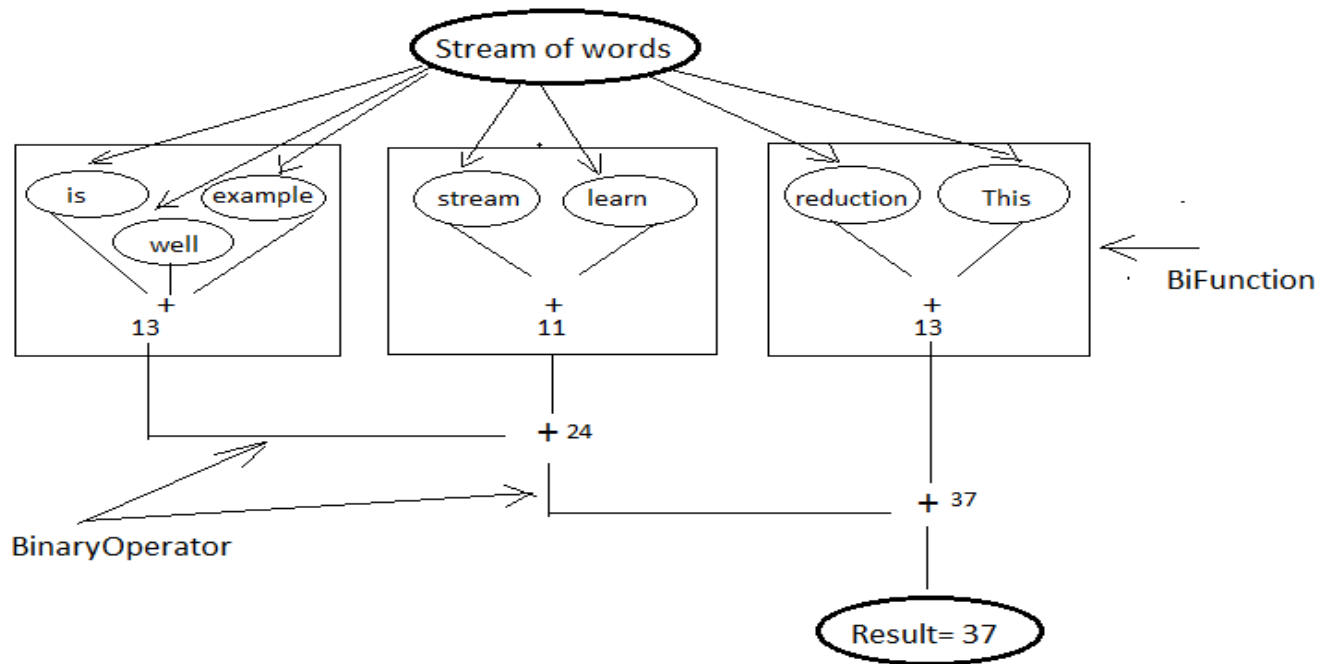

Fig. 6.2: **Parallel reduction**

Now re-run the code in parallel (words.parallelStream()...) and look into the output. Combiner calculate the sum of two partial results.

```
BiFunc: 0   This
BiFunc: 0   stream
BiFunc: 0   well
BiFunc: 0   learn
BiOpr: 5   4
BiFunc: 0   reduction
BiFunc: 0   example
BiOpr: 9   7
BiOpr: 16   9
BiFunc: 0   is
BiOpr: 2   6
BiOpr: 4   8
BiOpr: 12   25
```

To Array

Stream interface supports two overloaded `toArray` methods that will collect stream elements as an array.

- **Object[] toArray():** This is the simplest form of toArray operation which returns an Object array of length equal to Stream length.

  Example: Integer[] arr = Stream.<Integer>of(10, 20, 30, 40, 50).toArray();

- **T[] toArray(IntFunction<T[]> generator):** You saw the first toArray method always returns array of Object type, but this overloaded method will return array of desired type. It accepts an IntFunction as argument that describes the behaviour of taking array length as input and returns the array of generic type.

```
Employee[] arr = employees.stream().filter(e -> e.getGender() == MALE)
                    .toArray(Employee[]::new);


                                OR

            employees.stream().filter(e -> e.getGender() == MALE)
                    .toArray(len -> new Employee[]);
```

Infinite Streams

We already discussed, Streams can be derived from different sources:

- From array - Arrays.stream(T[])

- From known elements - Stream<String>.of("Stream", "is", "great")

- From file - Files.lines(Paths.get("myfile.txt"))

Please visit the Stream sources section for basics of stream sources. The streams generated from above sources are bounded streams where elements size is known. Stream interface supports two static methods `Stream.iterate()` and `Stream.generate` which returns infinitite streams that will produce unbounded stream. As generated stream will be unbounded , it's necessary to call limit(n) to convert stream into bounded.

Note: You can use *findAny* or *findFirst* terminal operations to terminate the stream if you assure required result is exist in the stream. Example: `Stream.<Integer>iterate(1, v -> v + 3).filter(i -> i % 5 == 0).findAny().get())` Here we are sure that there will be an element which will be divisible by 5 so you can use findAny to terminate the stream.

- **Stream.iterate:** : *Signature*: Stream<T> iterate(T seed, UnaryOperator<T> f)

  It returns an infinite sequential ordered Stream produced by iterative application of the given function. The function here is a UnaryOperator which uses the previous calculated result to produce next result. It also accepts a seed value that will be supplied to the UnaryOperator as initial value.

```
// Generating fibonacci numbers of a given length
Stream.iterate(new int[] { 0, 1 }, a -> {
    int next = a[0] + a[1];
    a[0] = a[1];
    a[1] = next;
    return a;
}).limit(10).map(a -> a[0]).forEach(System.out::println);

Output: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

- **Stream.generate:** : *Signature*: Stream<T> generate(Supplier<T> s)

    It returns an infinite sequential unordered stream where each element is generated by the provided Supplier. As we know Supplier doesn't accept any argument so the `generator` doesn't depend on previously calculated value. Below example generates UUID values of a given length.

    ```
    Stream.generate(UUID::randomUUID).limit(5).forEach(System.
    out::println)
    ```

Comparator

Comparator is there since jdk 1.2 and almost all of us know its usage and importance. Java-8 came up with couple of updates to the Comparator as given below.

- Additionl default and static methods added into `Comparator` interface to support pipeline of stream operations.

- `String` class added with `CaseInsensitiveComparator` to sort by ignoring the case.

- A new utility class `Comparators` is bundled with jdk-8 to support natural ordered sorting and handling `null` values while sorting.

Comparators

Comparators is a helper class that provides new Comparator implmentations in the following cases.

- To impose natural ordrered sorting on elements of Comparable types

- Sorting on the collections that mixed with null values.

Natural ordering:

```
NaturalOrderComparator implements Comparator<Comparable<Object>> {

    @Override
    public int compare(Comparable<Object> c1, Comparable<Object> c2) {
        return c1.compareTo(c2);
    }
}
```

`Comparator` interface contains a static method called `naturalOrder` which returns a `NaturalOrderComparator` that imposes sorting on elements implementing `Comparable`. As you know all wrapper classes for primitive types implements Comparable interface so this natural ordered sorting can be applicable to all of them.

Handling null elements:

Usually comparators throws `NullPointerException` if null elements found while performing sorting operation. `Comparator` contains two methods *nullsFirst* and *nullsLast* that takes a comparator as an argument and returns another null-friendly comparator by wrapping the given comparator. It will arrange null elements at the begining or end depending on the operation you called. For non-null elements it will sort them using the comparator passed initially.

Updates in Comparator

Comparator interface contains some of static methods that returns another comparator implementations described below.

- **comparing(Function<T,U> keyExtractor)** This method uses the given key extracting function that applies on T type elements to generate U type comparable sort keys. To compare two elements of type T, it first applies the key extracting function to both the elements and then performs the sorting operation on the resulted keys.

```
// Sorting words based on word lengths
Function<String, Integer> keyExtractor = str -> str.length();
Stream.of("grapes", "milk", "pineapple", "water-melon")
    .sorted(Comparator.comparing(keyExtractor))
    .forEach(System.out::println);
```

In the above code snippet a `Function<String, Integer> keyExtractor` object is passed to the `comparing` method that in turn will return a *Comparator* object. It first applied the function to string elements and generated string lengths then returned a comparator definition as given below.

```
Comparator<Integer> c = (s1, s2) -> keyExtractor.apply(s1).
compareTo(keyExtractor.apply(s2))
```

- **comparing(Function<T,U> keyExtractor, Comparator<U> keyComparator)** In the first `comparing` method, key extracting function returns sorting keys of `Comparable` type so it doesn't need additional Comparator object to perform sorting. But in this `comparing` function it first uses the key extracting function to generate key and then performs sorting based on the given comparator.

```
Stream.of("grapes", "milk", "pineapple", "water-melon")
    .sorted(Comparator.comparing(String::length, Comparator.reverseOrder()))
        .forEach(System.out::println);
```

- **comparingXXX(ToXXXFunction<T> keyExtractor)** Comparator interface provides three primitive comparing functions: *comparingInt, comparingDouble and comparingLong* to sort the elements based on the primitive keys. It accepts ToXXXFunction functional interface which returns primitive values that avoid unnecessary boxing-unboxing costs while doing sorting.

```
// Natural order sorting by ignoring the sign.
Stream.of(-10, 31, 16, -5, 2)
    .sorted(Comparator.comparingInt(i -> Math.abs(i)))
    .forEach(System.out::println);
```

- **thenComparing(Comparator<T> other)** It is very much possible that two elements will be equal according to the given comparator. In such cases the other comprator decides the sorting order. Below code snippet shows example of sorting employee objects based on employee's salary and then uses name if two salaries are equal.

```
List<Employee> employees = Application.getEmployees();
employees.stream()
```

```
    .sorted(Comparator.comparing(Employee::getSalary).
→thenComparing(Employee::getName))
    .forEach(System.out::println);
```

Collectors

In the Stream API section we saw some of the terminal operations: *toArray, reduce, forEach etc* used to calculate the end result from the pipeline of intermediate operations. Stream interface contains a most frequently used terminal operation `collect` that performs the reduction operation on the elements of the stream using Collector interface.

Collector is nothing but a mutable reduction operation that accumulates elements from the stream into a mutable container and finally it returns either the same result container or a different representation of the container dependening on the characteristics given. Reduction operations can be performed either sequentially or in parallel.

Examples of mutable reduction operations include: accumulating elements into a Collection; concatenating strings using a StringBuilder; computing summary information about elements such as sum, min, max, or average; collecting elements as groups etc.

How Collector works?

Collector divides the complete reduction process to four sub-tasks that best fits to any type of reduction operation. They are:

1. supplying a new empty result container at the begining

2. accumulating new data element into the result container

3. combining two result containers into one in case of parallelization

4. performing an optional final transform on the container

All of these sub-tasks may or may not be needed for every operation but these are the generalized form of the complete process. Collectors also have a set of characteristics, such as `Characteristics.CONCURRENT`, that provide hints to the reduction process to provide better performance. *Collector.Characteristics* enum contains three characteristics as:

- *UNORDERED:* Indicates that the collection operation does not commit to preserving the encounter order of input elements. This might be true if the result container has no intrinsic order, such as a Set.

- *CONCURRENT:* Indicates that this collector is concurrent, meaning that the result container can support the accumulator function being called concurrently with the same result container from multiple threads. Remember

marking CONCURRENT doesn't always execute concurrently, if not marked as UNORDERED or applied to an unordered data source like Set etc.

- *IDENTITY_FINISH:* Setting on this property returns the result container as the final result with out calling `Collector.finish()`.

Collector interface contains below five methods to support all of the above subtasks.

```
interface Collector<T, A, R> {
    Supplier<A> supplier();
    BiConsumer<A, T> accumulator();
    BinaryOperator<A> combiner();
    Function<A, R> finisher();
    Set<Characteristics> characteristics();
}
```

In this listing the following definitions apply:

- T denotes the generic type of the stream elements

- A represents the type of the supplier or type of the accumulator where the partial results will be accumulated

- R is the type of result to be returned at the end. If the IDENTITY_FINISH characteristic is given then both A and R will be of the same type.

Let's discuss the methods declared by the collctor interface:

- **Supplier<A> supplier()** The supplier method will return an empty result container whenever invoked. Remember, this method will be called only once if reduction operation is requested for sequential execution and multiple times if parallel execution.

- **BiConsumer<A, T> accumulator()** Accumulator will define the behaviour of the accumulation process. You might be already noticed that, though it is `BiConsumer` it takes partial result container and a new element as inputs and performs the configured task.

- **BinaryOperator<A> combiner()** Combiner defines what to be done if two partial results are provided. As we know in the parallelization case, the complete dataset will be splitted to multiple chunks and performed separately, so combiner will merge the two partial results into one. The `BinaryOperator's` functional decsriptor is exactly matching with this task: (partial1, partial2) -> partial1.merge(partial2)

- **Function<A, R> finisher()** This defines the final transformation to be done to the result container after all the elements are processed.

- **Set<Characteristics> characteristics()** Returns the immutable set of Characteristics, defining the behavior of the collector.

Implementing collectors

Now we have enough idea on what are the methods collector interface provides and how does they work. So let's implement our own collector that takes a set of *Employee* objects and generates a XML content.

```
public class Employee {
    public String name;
    public String empid;
    public String technology;
    public double salary;
}

public class ToXMLCollector implements Collector<Employee, StringBuffer, String> {
```

```java
    final String xmlstr = "\n   <employee eid='%s'>\n\t" + "<name>%s</name>\n\t"
            + "<tech>%s</tech>\n\t<salary>%s</salary>\n   </employee>";

    public Supplier<StringBuffer> supplier() {
        return StringBuffer::new;
    }

    public BiConsumer<StringBuffer, Employee> accumulator() {
        return (sb, e) -> sb.append(String.format(xmlstr, e.empid, e.name, e.
→technology, e.salary));
    }

    public BinaryOperator<StringBuffer> combiner() {
        return (sb1, sb2) -> sb1.append(sb2.toString());
    }

    public Function<StringBuffer, String> finisher() {
        return sb -> String.format("<employees> %s \n</employees>", sb.toString());
    }

    public Set<Characteristics> characteristics() {
        return EnumSet.of(CONCURRENT);
    }


    public static void main(String[] args) {
        Set<Employee> emps = Database.employees();
        String xmlstr = emps.parallelStream().collect(new ToXMLCollector());
        System.out.println(xmlstr);
    }
}


Output:
-------
<employees>
    <employee eid='E1001'>
        <name>Mr Bean</name>
        <tech>Cloud Computing</tech>
    </employee>
    <employee eid='E1002'>
            <name>J Smith</name>
            <tech>Java</tech>
    </employee>
</employees>
```

In this example we created a separate ToXMLCollector class by overriding all of the collector methods but `Collector` interface also has `Collector.of` static methods that accepts the collector behaviors and returns a anonymous Collector instance.

- **Collector<T, A, R> of(Supplier<A> supplier, BiConsumer<A, T> accumulator,** BinaryOperator<A> combiner, Function<A, R> finisher, Characteristics... characteristics)

- **Collector<T, A, R> of(Supplier<A> supplier, BiConsumer<A, T> accumulator,** BinaryOperator<A> combiner, Characteristics... characteristics)

Using these helper method our `ToXMLCollector` can also be implemented as:

```
Collector.<Employee, StringBuffer, String>of(StringBuffer::new,
  (sb, e) -> sb.append(String.format(xmlstr, e.empid, e.name, e.technology)),
    (sb1, sb2) -> sb1.append(sb2.toString()),
      sb -> sb.insert(0, "<employees>").append("\n</employees>").toString(),
        Collections.emptySet());
```

Predefined Collectors

In the previous chapter you got an overal idea on how does collector works and how to implement custom collectors. Java-8 has introduced `java.util.stream.Collectors` class containing many factory methods that provides most commonly used `Collector` implementations. Collectors mainly offers following functionalities:

- Collecting elements to a *java.util.Collection*

- Joining String elements to a single String

- Grouping elements by custom grouping key

- Partitioning elements into TRUE FALSE group

- Reducing operations

- Summerizing elements

These factory methods can also be combined to generate nested Collector that we will see while moving deeper.

Collecting as collections

Collecting stream elements to a *java.util.Collection* is the most widely used operation. Collectors class provide couple of methods that returns a collector which will then collect stream elements to a specific collection container.

Collector<T, ?, List<T>> toList()
Collector<T, ?, Set<T>> toSet()
Collector<T, ?, C> toCollection(Supplier<C> collectionFactory)
Collector<T, ?, Map<K,U>> toMap(Function<T, K> keyMapper, Function<T, U> valueMapper)

- **toList():** Returns a Collector that will accumulate stream elements into ArrayList in the encountered order.

```
List<String> list = Stream.of("java", ".net", "python")
            .map(String::toUpperCase).collect(Collectors.toList());
```

- **toSet():** Returns a Collector that will accumulate stream elements into HashSet object.

```
Set<String> set = Stream.of("java", ".net", "python")
            .map(String::toUpperCase).collect(Collectors.toSet());
```

- **toCollection(Supplier<C> collectionFactory):** The first two methods returns collectors using ArrayList and HashSet as the container, but in case you need some other Collection implementations then *toCollection* method can be helpful which accept a supplier representing the type of the container to be used for the accumulation process.

```
TreeSet<String> set = Stream.of("java", ".net", "python").
→map(String::toUpperCase)
            .collect(Collectors.toCollection(TreeSet::new));
```

- **toMap(Function<T, K> keyMapper, Function<T, U> valueMapper):** Returns a Collector that accumulates elements into a Map whose keys are derived from keyMapper function and values are from valueMapper function.

```
Map<String, Integer> result = Stream.of("java", ".net", "python")
        .collect(Collectors.toMap(String::toUpperCase, String::length));

Output: {JAVA=4, .NET=4, PYTHON=6}
```

Sometime it is very obvious that the keyMapper function will derive duplicate key either by same element in the stream or the mapper function is responsible for that. In such situtaion *toMap* will throw `java.lang.IllegalStateException: Duplicate key`. Collectors class has another overloaded method that takes a merge function to decide the action to be taken if duplicate key is found.

```
toMap(Function<T, K> keyMapper, Function<T, U> valueMapper,
BinaryOperator<U> mergeFunction)
```

```
Map<String, Integer> result = Stream.of("java", ".net", "python", "jAvA")
        .collect(Collectors.toMap(String::toUpperCase, String::length, (key1,␣
→key2) -> key1));

Output: {JAVA=4, .NET=4, PYTHON=6}
```

Here we are passing a merge function that says "consider the first key if two keys are duplicates". You can also provide some other merge function that will generate a composite key using both keys. The first two `toMap` methods will use *HashMap* as the accumulator container. Collectors has also a 4-arg overloaded toMap method that takes a supplier to define the *Map* type will be used for accumulation.

```
toMap(Function<T, K> km, Function<T, U> vm, BinaryOperator<U> mf,
Supplier<M> mapSupplier)
```

Strings joining

Collectors utility class provides some of overloaded methods that concatenates stream elements into a single string either by separating them with a delimiter if provided.

Collector<CharSequence, ?, String> joining()
joining(CharSequence delimiter)
joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix)

The default delimiter for the no argument `joining` method is an empty string. The three argument *joining* method takes prefix and suffix which will be joined in the front and rear end of the final concatenated string result.

```
Stream.of("java", ".net", "python").collect(joining(", ", "Joined String[ ", " ]"));

Output: Joined String[ java, .net, python ]
```

Grouping elements

A common database operation is to group records based on one or multiple columns similarly Collectors also provide factory method that accepts a classification function and returns a Collector implementing a "group by" operation on stream elements T.

The classification function derives grouping keys of type K from stream elements. The collector produces a Map<K, List<T>> whose keys are the values resulting from applying the classification function to the input elements, and values are Lists containing the input elements which map to the associated key under the classification function.

Below is the entity class definition and the data we will be using through out the collector examples.

```java
public class Trade {

    private String tradeId;
    private String trader;
    private double notional;
    private String currency;
    private String region;

    // getters and setters
}
```

Table 9.1: Trade deals

Trade Id	Trader	Notional	Currency	Region
T1001	John	540000	USD	NA
T1002	Mark	10000	SGD	APAC
T1003	David	120000	USD	NA
T1004	Peter	4000	USD	NA
T1005	Mark	300000	SGD	APAC
T1006	Mark	25000	CAD	NA
T1007	Lizza	285000	EUR	EMEA
T1008	Maria	89000	JPY	EMEA
T1009	Sanit	1000000	INR	APAC

Now let's group the trade deals according to country region.

```java
Map<String, List<Trade>> map =  trades.stream()
    .collect(Collectors.groupingBy(Trade::getRegion));


Output:
{
   APAC: [T1002, T1005, T1009],
   EMEA: [T1007, T1008],
   NA: [T1001, T1003, T1004, T1006]
}
```

In the above example we passed `Trade.getRegion()` as the classification function. `grouping` method will

apply the given classification function to every element T to derive key K and then it will place the stream element into the corresponding map bucket. The grouping operation we just perfomed is very simple and straight-forward example but Collectors also support overloaded factory methods for multi-level grouping such as grouping trade detals according to region and currency.

groupingBy(Function<T, K> classifier, Collector<T, A, D> downstream): This overloaded method accepts an additional downstream collector to which value associated with a key will be supplied for further reduction. The classification function maps elements T to some key type K and generates groups of List<T>. The downstream collector will then operates on each group of elements of type T and produces a result of type D, at last collector will produces a result of Map<K, D>.

Below example is grouping trade deals according to region and currency. The end result from this example will be Map<Region, Map<Currency, List<Trade>>>.

```
Map<String, Map<String, List<Trade>>> map2 = trades.stream()
    .collect(Collectors.groupingBy(Trade::getRegion,
                Collectors.groupingBy(Trade::getCurrency)));
System.out.println(map2);


Output:
{
    NA={CAD=[T1006], USD=[T1001, T1003, T1004]},
    EMEA={EUR=[T1007], JPY=[T1008]},
    APAC={SGD=[T1002, T1005], INR=[T1009]}
}
```

There is no limit on grouping, you can call nested grouping any times you want. Now let's look into the `groupingBy` method signature once again. Does this method only meant for multi-level grouping? No. The method accepts a `Collector` as a second argument and we can do much more by passing different Collector implementations. Below example demonstrates counting number of deals in each region.

```
Map<String, Long> map2 = trades.stream()
    .collect(Collectors.groupingBy(Trade::getRegion, Collectors.counting()));

Output:
{NA=4, EMEA=2, APAC=3}
```

groupingBy(Function<T,K> f, Supplier<M> mapFactory, Collector<T, A, D> dc): Just like *toCollection* method we saw in the begining, this method also facilitates to pass a map factory to decide the group container type. The default map object type is `Hashmap` so you can use this method if some other map type required.

See also:

All these grouping collectors doesn't no guarantee on the thread-safety of the Map returned, so check `Collectors.groupingByConcurrent` methods for thread-safety operations.

Partitioning elements

Partitioning a special type of grouping but it will always contain two groups: FALSE and TRUE. It returns a Collector which partitions the input elements according to a Predicate supplied, and organizes them into a Map<Boolean, List<T>>. Following example shows partitioning deals to USD and no USD deals.

```
Map<Boolean, List<Trade>> map2 = trades.stream()
    .collect(Collectors.partitioningBy(t -> "USD".equals(t.getCurrency())));
System.out.println(map2);
```

```
Output:
{
    false=[T1002, T1005, T1006, T1007, T1008, T1009],
    true=[T1001, T1003, T1004]
}
```

Reducing collectors

Like `java.util.stream.Stream`, Collectors class also provides some overloaded reducing methods. To perform simple reduction operation on a stream, `Stream.reduce(Object, BinaryOperator)` methods can be used. The purpose of reducing() collectors are mostly for multi-level reduction operations. Following are list of overloaded reducing collectors given by Collectors class.

reducing(T identity, BinaryOperator<T> op)
reducing(BinaryOperator<T> op)
reducing(U identity, Function<T,U> mapper, BinaryOperator<U> op)

Collectors reducing methods are similar to *Stream.reduce* operation. If you haven't checked them, then see the Stream API section.

Arithmetic & Summerizing

Collectors also has some of methods that returns collector to perform arithmetic operations like finding max, min, sum and average. Below are the method defined in Collectors utility class.

Collector<T, ?, Optional<T>> minBy(Comparator<T> comparator)
Collector<T, ?, Optional<T>> maxBy(Comparator<T> comparator)
Collector<T, ?, XXX> summingXXX(ToXXXFunction<T> mapper)
Collector<T, ?, XXX> averagingXXX(ToXXXFunction<T> mapper)

I don't have to explain what these method do, they are self explanatory. Collectors has individual `summing` and `averaging` methods for these three primitive types: int, double and long. As like reduction operations, arithmetic fuctions are also available in IntStream, DoubleStream and LongStream interfaces that can be used for simple stream reduction. These arithmetic collectors will be helpful for nested reduction operations through other collectors.

Apart from individual arithmetic operations, Collectors has also `summarizingXXX` factory methods that will perform all of these arithmetic operations all togather. The collector produced by summerizing function will return `XXXSummaryStatistics` class which is a container for holding results calculated for these arithmetic operations.

Method signature

Collector<T, ?, DoubleSummaryStatistics> summarizingDouble(ToDoubleFunction<T> mapper)

The *summarizingDouble* method accepts a `ToDoubleFunction` that will apply on the stream elements of type T to generate double type values on which summarization functionality will be executed. Below example demonstrates the usage of `summarizingDouble` method.

```
Map<String, DoubleSummaryStatistics> map = trades.stream()
        .collect(Collectors.groupingBy(Trade::getRegion,
            Collectors.summarizingDouble(Trade::getNotional)));

DoubleSummaryStatistics naData = map.get("NA");
```

```
System.out.printf("No of deals: %d\nLargest deal: %f\nAverage deal cost: %f\nTotal␣
↪traded amt: %f",
    naData.getCount(), naData.getMax(), naData.getAverage(), naData.getSum());



Output:
No of deals: 4
Largest deal: 540000
Average deal cost: 172250
Total traded amt: 689000
```

Miscellaneous

We saw *grouping* and *partitioning* functions that accepts another downstream collector used for nesting operations.
Collectors class also provides two additional methods mostly used for such nested complex situations.

Collector<T,A,RR> collectingAndThen(Collector<T,A,R> c, Function<R,RR> f)
Collector<T, ?, R> mapping(Function<T,U> mapper, Collector<U, A, R> c)

1. **collectingAndThen(Collector<T,A,R> downstream, Function<R,RR> finisher)**

   It will return a collector that will additionally perform a finishing transformation after the downstream
   collector collected elements. We will see few examples with explanation to get more clarity on the usage.

   ```
   Set<Trade> set = trades.stream().collect(collectingAndThen(toSet(),
   Collections::unmodifiableSet))
   ```

   In this example *toSet* collector will first collect elements to a set and then the resulting set will be applied
   to the finisher function to return a unmodifiable set. This is the simplest usage of *collectingAndThen*
   method and it has more meaning when used with nested collectors. Below code snippet demonstrates an
   advanced usage of the method that is finding maximum valued deal in each region.

   ```
   Map<String, Optional<Trade>> map1 = trades.stream()   // Solution-1
       .collect(groupingBy(Trade::getRegion,␣
   ↪maxBy(comparing(Trade::getNotional))));



   Map<String, Trade> map2 = trades.stream()              // Solution-2
           .collect(groupingBy(Trade::getRegion,
                   collectingAndThen(maxBy(comparing(Trade::getNotional)),␣
   ↪Optional::get)));
   ```

   We already know that `Collectors.maxBy` produces values of *Optional* types but actually we were
   expecting for Trade typed values. The `collectingAndThen` is first calculating the maximum valued
   deal wrapped with *java.util.Optional* and then passes to the finisher function to call `Optional.get()`
   which will then extract Trade object out of it.

2. **mapping(Function<T,U> mapper, Collector<U, A, R> downstream)**

   `collectingAndThen()` resulting collector first collect elements and then applies the transformation
   function but the `mapping` collector applies the function before collecting elements. It returns a collector
   which applies the mapping function to the input elements and provides the mapped results to the down-
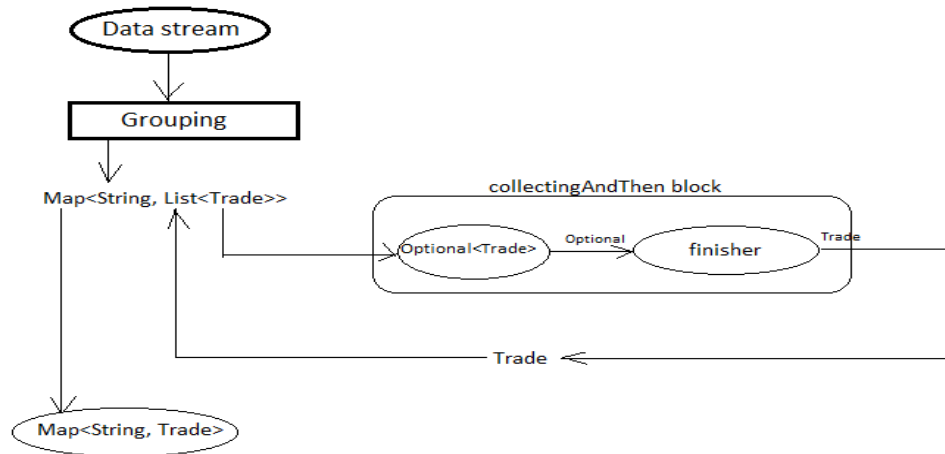   stream collector. As like *collectingAndThen*, the mapping() collectors are most useful when used in a
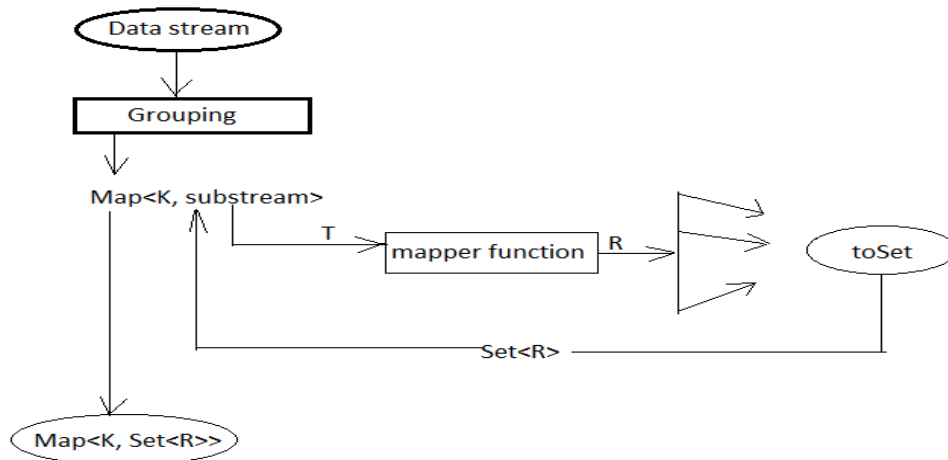
Fig. 9.1: **Solution-2 flow diagram**

multi-level reduction, such as downstream of a groupingBy or partitioningBy. For example, accumulate the set of trade ids in each region.

```
Map<String, Set<String>> map = trades.stream()
        .collect(groupingBy(Trade::getRegion, mapping(Trade::getTradeId,␣
↪toSet())));
System.out.println(map);
```

Handling nulls with Optional

Tony Hoare-one of the giants of computer science once told, "*I call it my* **billion-dollar mistake**. *It was the invention of the null reference in 1965. I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement.*" The null reference is the source of many problems because it is often used to denote the absence of a value. As a java developer you would have felt the pain of getting NullPointerException. Millions of projects are running using Java and guess the total amount of dollars spent on fixing those issues.

Imagine a job portal maintaining candidate database with the following nested object structure.

```
class Candidate{
    String name;
    String spouse;

    jobProfile: class Job{
        int yearsOfExpr;

        uiExperience: class Framework{
            String name;
            String proficiency;

            certification: class Certification{
                String type;
                double score;
            }
        }
    }
}
```

The structure looks pretty reasonable, but a candidate can be a recent college passout for whom job-profile could be missing, or for a candidate doing job might not have worked on any UI (User Interface) frameworks or even worked on might not be certified. All of these scenarios are proven to generate NullPointerException if not carefully handled in the code.

The simplest solutions depelopers are adopting is the defensive approach of having `null` checks.

```
String uiProficiency(Candidate candidate){
    String proficiency = null;
    Job job = candidate.getJobProfile();
    if(job != null){
        if(job.getUiExperience() != null){
            String expr = job.getUiExperience().proficiency;
            proficiency = min(expr, "Intermediet");
            Certification cert = ui.certification;
            if(cert != null && cert.score >= 60){
                proficiency = "Expert";
            }
        }
    }
    return proficiency;
}
```

No doubt that code will run without any issues but it still has following problems:

- It looks wierd and looses the readability.

- Repeated low levels codes such as `if` conditions.

- Maintainability will be difficult as nesting levels goes on.

Java8 has introduced `java.util.Optional` singleton class to deal with such problems and additionally provides some utility methods that can be used in certain common scnearios. *Optional* is a container or a wrapper class that represents value might or might not exist for a variable. When value present you can use `get` method to fetch the value or on absent it just behaves as an empty container. We get exception when we directly operate on the null instances so Optional promotes to use its utility methods to perform operations. With keeping things in mind we can reimplement the original 'Candidate' model class given below.

```
class Candidate {
    String name;
    Optional<String> spouce;
    Optional<Job> jobProfile;
}

class Job {
    int yearsOfExpr;
    Optional<Framework> ui;
}

class Framework {
    String name;
    String proficiency;
    Optional<Certification> certification;
}

class Certification {
    String type;
    double score;
}
```

The benifits over using Optional are:

- No need to document separetely to represent nullable members, model class Optional types are self documentary. As an example *spouse* and *jobProfile* clearly mentions that they can be null.

- No need to write null checks explicitely, operations will be performed only if value is present.

Optional Construction

Creating optional objects are damn easy, it provides following factory methods to create Optionals.

- **Empty Optional:**

  `Optional.empty()` gets you an hold of empty optional object. The default values for the nullable members of an object can be of this type which passed to some other code won't through NullPointerException and will supress any operation performed on it. Even though `Option.empty() == Option.empty()` returns true, Optional promotos to use `isPresent` method to perform the equility operation.

  `Optional<Job> optJob = Optional.empty();`

- **Optional from nullable value:**

  You can create an optional object from a nullable value using the static factoy method `Optional.ofNullable`. The advantage over using this method is if the given value is null then it returns an empty optional and rest of the operations performed on it will be supressed.

  `Optional<Job> optJob = Optional.ofNullable(candidate.getJobProfile());`

- **Optional from non-null value:**

  You can also create an optional object from a non-null value using the static factoy method `Optional.of`. Use this method if you are sure given value is not null otherwise it will immidietely throw NullPointerException.

  `Optional<Job> optJob = Optional.of(candidate.getJobProfile());`

  There is no other difference in using `Optional.of` or `Optional.ofNullable` except *of()* methods creates the perception that given value is mandatory field and passing null is the unaccepted criteria.

  Note: Most of languages has concept of missing values and they handle it in different ways. Scala has a safe way to navigate through values, Google's Guava library and Groovy language has same construct as Java Optional, so we can say java Optional can be inspired from them.

Operating on Optionals

Optional provides three basic methods: *map, flatMap* and *filter* to perform any kind of common task. Like Streams these operations can also be chained togather to perform composite tasks.

- **map(Function<T, U> mapper):**

  If a value is present, apply the provided mapping function to it, and return an Optional describing the result, otherwise return an empty Optional. Similar to *Stream.map* method, this is also commonly used as transformation function.

  This method supports post-processing on optional values, without the need to explicitly check for a return status. For example, the following code snippet traverses a stream of trades, selects first APAC trade encountered, and then returns the trade id, returning an Optional<String>:

```
Optional<String> opt = trades.stream()
    .filter(trade -> "APAC".equals(trade.getRegion()))
    .findFirst()
    .map(Trade:::getTradeId);
```

- **flatMap(Function<T, Optional<U>> mapper):**

  Before getting through *flatMap* method let's try an example to find the UI certification done by a candidate who is having a job.

  ```
  String certificationName = candidate.getJobProfile()
      .map(Job::getFramework)
      .map(Framework::getCertification)
      .orElse(null);
  ```

  Yeah pretty easy, but unfortunately this code will not compile at all. The reason is `Job.getFramework()` returns `Optional<Framework>` type so the first *map* method will produce Optional<Optional<Framework>>. This return value will again input for the second map method and we know that `getCertification` method exist in 'Framework' class not in 'Optional<Framework>'.
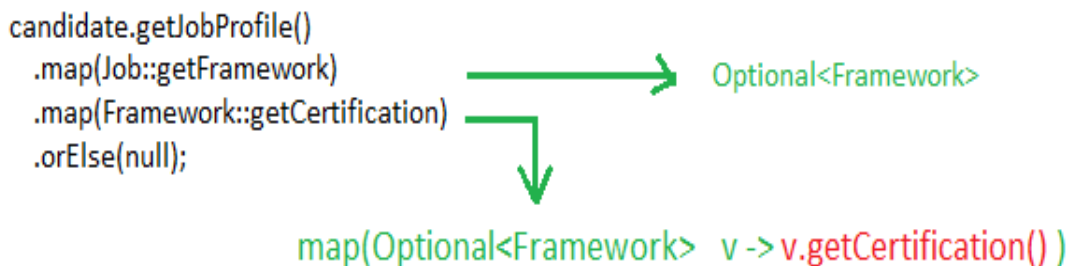
  The solution here is to replace the map methods with *flatMap* method. This method is similar to map method but flatMap expects the mapper function return type already to be Optional which will be directly returned as the final result. If you notice map & flatMap methods internal implementations, map method will wrap mapper calculated result inside a Optional object where as flatMap method will not.

  ```
  public<U> Optional<U> map(Function<? super T, ? extends U> mapper) {
      return Optional.ofNullable(mapper.apply(value));
  }

  public<U> Optional<U> flatMap(Function<? super T, Optional<U>> mapper) {
      return Objects.requireNonNull(mapper.apply(value));
  }
  ```

  Below snippet is the correct solution for our earlier example.

  ```
  String certificationName = candidate.getJobProfile()
      .flatMap(Job::getFramework)
      .flatMap(Framework::getCertification)
      .orElse(null);
  ```

  We saw map and flatMap methods in details, now I will show you a nice usage by combining both methods that will be used often. Imagine there is a external service which calculates the reimbursement amount.

```
double calculate(Optional<Framework> optFrm, Optional<Certification>␣
→optCert) {
    return optFrm.flatMap(framwork ->
        optCert.map(certification -> reimburse(framwork, certification)))
        .get();
}
```

Here the map method is called inside flatMap just for the availability of framework value to invoke *reimburse*. Originally reimbursement will be executed by map method and flatMap will just return calculated result.

- **filter(Predicate<T> predicate):**

If the value matches the given predicate, then the Optional containing the value will be returned, otherwise an empty Optional.

```
boolean isCertified = candidate.getJobProfile()
    .flatMap(Job::getFramework)
    .flatMap(Framework::getCertification)
    .filter(certification -> certification.score >= 60)
    .isPresent();
```

Retrieving from Optionals

Optional provides following methods to retrive values from optional object.

Method	Description

get()	Returns the value wrapped by the Optional or throws NoSuchElementException if doesn't contain data. Use this method if you are sure optinal holding data.
	`int years = Optional.of(job).`
	`map(Job::getYearsOfExpr).get()`

orElse(default_value)	Return the value if present, otherwise default_value. This method is the safest way to get the value.
	`String spouse = Optional.`
	`of(candidate).map(Candidate::getSpouse).`
	`orElse(null)`

orElseGet(Supplier<T> other)	Return the value if present, otherwise retrived from supplier. This is the lazy way to retrive value. As an example call an external service in case value not exist in optional. If you use *orElse(external_service)* then the service will be executed irrespective of the original value exist which can impose additional cost.
	`Optional.of(trade).map(Trade::getId).`
	`orElseGet(UUID::randomUUID)`

orElseThrow(Supplier<Throwable> exception)	Return the value if present, otherwise throw supplied exception. Using *get* method will always return NoSuchElementException but custom exceptions could be returned using this.
	`String spouse = Optional.`
	`of(candidate).orElse(YourException::new)`

isPresent()	Returns true if value present Otherwise false.
	`boolean cond = Optional.of(job).`
	`isPresent()`

ifPresent(Consumer<T> consumer)	If a value is present, invoke the specified consumer with the value, otherwise do nothing.
	`Optional.of(job).ifPresent(System.`
	`out::println)`

Miscellaneous

Primitive Optionals:

As like streams, Optionals also have primitive flavours- OptionalInt, OptionalLong and OptionalDouble. These primitives should be used when operating on streams otherwise its usage is discouraged. Stream can contain huge number of elements where using primitives can save time as well as space but an Optional can have at most single value and primitive optional will not employ much difference in fact it will stop you to use common methods like map, filter etc.

Optionals can't be serialized:

Optionals were designed to handle missing values. These were not intended for use as a field type so it doesn't implement Serializable. In case you need to have a serializable domain model, implement getter methods returning optionals given below.

```
class Candidate {
   String name;
   String spouce;

   public Optional<String> getSpouse(){
      return Optional.ofNullable(spouse);
```

```
    }
}
```

Default and Static methods

Prior to java8 interfaces were containing only abstract methods but this time you will be able to provide concrete implementations inside interface. Usually interfaces are contracts that defines the set of operations to be supported for a usecase and all of its implementing classes should provide implementations to those abstract methods. When there is a need to provide some basic common functionalities to all implementating classes, the general approach was taken to introduce an abstract class which was inherited by them rather than directly implementing interface. Just to take an example think about Servlet case.
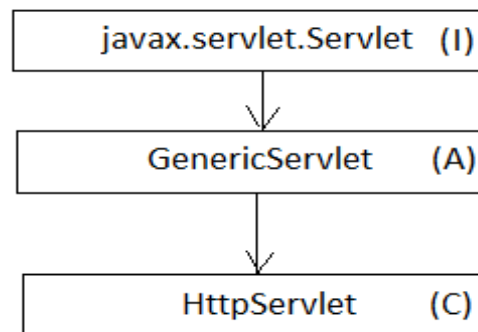
Servlet defines the contract and GenericServlet were introduced just to provide common functionalties to implementing classes like HttpServlet. This was not the bug but an approach taken in older days. Now java has been evolved a lot to remove this intermedite classes and common operations could be reside inside interfaces.

Java8 has introduced many new methods on existing interfaces such as the `sort` method in *List* interface, `stream` method in *Collection* etc. Java had always argued that its implementing classes must provide concrete implementation to all non-concrete methods of interface. There are millions of libraries and applications running on java and imagine the problem would have happened with directly adding methods inside interface. Java people had tough time to resolve this issue and finally came up with the solutions to add methods using `default` keyword.

Default methods

Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces. They provide a default implementation for methods. As a result, existing classes implementing an interface will automatically inherit the default implementations. You specify that a method definition in an interface is a default method with the default keyword at the beginning of the method signature. All method declarations in an interface, including default methods, are implicitly public, so you can omit the public modifier.

To get more clear picture let's discuss the `stream` method added in *Collection* interface.

```
default Stream<E> stream() {
    return StreamSupport.stream(spliterator(), false);
}
```

The stream method is required in all List and Set implementations so added in their super interface i.e. `Collection`. Doing this, stream method will now be directly available to all their implementing classes ArrayList, TreeSet. The default method is not only restricted to java people but you can also add default methods to your own interfaces.

```java
interface Vehicle {

   default void applyBreak() {
      System.out.println("Applying break.");
   }

   void transport();
}

class GoodsVehicle implements Vehicle {

   @Override
   public void transport() {
      System.out.println("Transporting goods.");
      applyBreak();
   }
}

class PublicTransport implements Vehicle {

   @Override
   public void transport() {
      System.out.println("Transporting people.");
      applyBreak();
   }

}
```

Multiple inheritance

You might have heard of the diamond problem in C++ where a class can inherit two methods of the same signature from two different classes. This is the reason that java adopted multiple inheritance from very begining. But introducing default methods it again opened the gate for the same issue. A class is able to implement multiple interfaces even if they contain abstract method with the same name.

```
public class SampleClass implements A, B {

   @Override
   public void print() {
      System.out.println("SampleClass");
   }

   public static void main(String[] args) {
      A a = new SampleClass();
      a.print();

      B b = new SampleClass();
      b.print();
   }
}

interface A {
   void print();
}

interface B {
   void print();
}
```

This was possible because the method is called on a single interface reference and both the interfaces are not interfering each other, they are just contracts. But now though interfaces can contain concrete methods, there is the possibility of a class inheriting more than one method with the same signature. Java 8 acknowledges this conflict with three basic principles.

1. A method declaration in the class or a superclass wins the priority over any default method declared in the interface.

```
interface A {
    default String print() {
        return "A";
    }
}

class MyClass {
    public String print() {
        return "MyClass";
    }
}

public class DefaultTest extends MyClass implements A {

    public static void main(String[] args) {
        System.out.println(new DefaultTest().print());
    }
}

Output: MyClass
```

Here *print* method is ingerited by both MyClass and interface A, but MyClass print method has taken into consideration.

2. The method with the same signature in the most specific default-providing interface will take the priority.

```
interface A {
    default String print() {
        return "A";
    }
}

interface B extends A {
    default String print() {
        return "B";
    }
}

public class DefaultTest implements A, B {

    public static void main(String[] args) {
        System.out.println(new DefaultTest().print());
    }
}

Output: B
```

Here *print* method is inherited by both interfaces but interface A extending B so B will be consider most specific or closer and will be considered.

3. In case choices are still ambiguous, the class inheriting from multiple interfaces has to override the default method and then it can provide its own implementation or can inherit any one. To call the super interface method `super` keyword is used.

```
interface A {
    default String print() {
        return "A";
    }
}

interface B {
    default String print() {
        return "B";
    }
}

public class DefaultTest implements A, B {

    public String print() {
        return A.super.print();
    }

    public static void main(String[] args) {
        System.out.println(new DefaultTest().print());
    }
}

Output: A
```

Here the `DefaultTest` class is choosing interface A prividing method with the help of super keyword.

Static methods

In addition to default methods, you can define static methods in interfaces. (A static method is a method that is associated with the class in which it is defined rather than with any object. Every instance of the class shares its static methods.) This makes it easier for you to organize helper methods in your libraries; you can keep static methods specific to an interface in the same interface rather than in a separate class.

Like static methods in classes, you specify that a method definition in an interface is a static method with the static keyword at the beginning of the method signature. All method declarations in an interface, including static methods, are implicitly public, so you can omit the public modifier. Through out the tutorial you have seen lot of example of interface static method like `Stream.of`, `Comparator.naturalOrder`, `Comparator.comparing` etc.

ForkJoinPool

In the begining of the tutorial we said- parallelization is almost free, with a small change you can enjoy the benifit of parallel stream processing. The complete parallel stream processing is based on ForkJoinPool concept. ForkJoinPool was delivered with jdk7 which provides a highly specialized `ExecutorService`. If you can recall, we submit multiple indepenedent tasks to ExecutorService which are then executed by thread-pool threads. In case of parallel stream we don't have multiple tasks where as we have a single complex task of larger size. To execute this big task we have to perform following actions explicitly.

- Divide the big task into smaller sub tasks

- Process all sub tasks independently

- Join the partial results from each sub task

ForkJoinPool internally does all these steps for you. We will typically submit a single task to ForkJoinPool and awaits its completion. The ForkJoinPool and the task itself work togather to divide and conquer the problem. Any problems that can be recursively divided can be a candidature for Fork-Join.

ForkJoinPool creation

Creating ForkJoinPool is simple, call its no-arg constructor to create an instance that will internally use `Runtime.availableProcessors()` method to determine number of worker threads (ForkJoinWorkerThread) to be used by the pool. It also provides an overloaded *ForkJoinPool(int parallelism)* constructor that allows user to override the number of threads to be created. Usually you shouldn't override the number of threads unless you have a strong reason to do it.

Though it internally uses the number of processors (cores) available to create the worker threads, we should always create a single instance of ForkJoinPool through out the application and different kinds of tasks should be submitted to the same pool. Its implementation restricts the maximum number of running threads to 32767 and attempting to create pools with greater than this size will result to *IllegalArgumentException*.

Note: The level of parallelism can also be controlled globally by setting `java.util.concurrent.ForkJoinPool.common.parallelism` system property which will affect every ForkJoinPool creation in your

application.

ForkJoinTask

As like Callable and Runnable in ThreadPoolExecutor, fork-join accepts a type of `ForkJoinTask` instance for the execution. The abstract base class ForkJoinTask is an implementation of `java.util.concurrecnt.Future` that provides common functionalities to its subclasses. It again offers two more abstract subclasses: *RecursiveAction* and *RecursiveTask* which has only one abstract method called "compute()". There is no difference between these two classes except `RecursiveTask` can return result where as `RecursiveAction` can not. You can assume RecursiveTask is an example of finding largest number from an array where as RecursiveAction is to sort an array which doesn't require to return any result.

ForkJoinTask has large set of methods but the methods we will be using most of the time are: **fork(), compute(), join()**. The compute method will contain the original task to be performed by the worker threads. The following is the pseudo code of the compute method.

```
if(Task is small) {
    Execute the task

} else {
    //Split the task into smaller chunks
    ForkJoinTask first = getFirstHalfTask();
    first.fork();
    ForkJoinTask second = getSecondHalfTask();
    second.compute();
    first.join();
}
```

The idea behind the fork-join is to divide the task into multiple smaller chunks and execute them independently. The compute method is responsible to split the task if it is not small enough to execute. In the pseudo code we have split the task into two but it can be split into more also. When you call *fork* on the first task it will be pushed into the queue and may be executed by some other thread, then you call compute method on second. After the competion of second task we will call join on the first task to wait for its completion. We will quickly see a complete example which demonstrates finding the largest element in an array.

```java
public class ForkJoinPoolTest {

    public static void main(String[] args) {
        int[] array = yourMethodToGetData();

        ForkJoinPool pool = new ForkJoinPool();
        Integer max = pool.invoke(new FindMaxTask(array, 0, array.length));
        System.out.println(max);
    }

    static class FindMaxTask extends RecursiveTask<Integer> {

        private int[] array;
        private int start, end;

        public FindMaxTask(int[] array, int start, int end) {
            this.array = array;
            this.start = start;
            this.end = end;
        }
```

```
21
22          @Override
23          protected Integer compute() {
24              if (end - start <= 3000) {
25                  int max = -99;
26                  for (int i = start; i < end; i++) {
27                      max = Integer.max(max, array[i]);
28                  }
29                  return max;
30
31              } else {
32                  int mid = (end - start) / 2 + start;
33                  FindMaxTask left = new FindMaxTask(array, start, mid);
34                  FindMaxTask right = new FindMaxTask(array, mid + 1, end);
35
36                  ForkJoinTask.invokeAll(right, left);
37                  int leftRes = left.getRawResult();
38                  int rightRes = right.getRawResult();
39
40                  return Integer.max(leftRes, rightRes);
41              }
42          } //end of compute
43
44      }
45  }
```

Here rather than calling *fork, compute* and *join* separately, we used `invokeAll` method which internally performs the same. There is no rule to define what is the size of the smaller chunk task, but the task should not be very small that it will loose the benifit of parallelism.

How fork-join works?

ForkJoinPool has array of DEqueues (WorkerQueue) which will be shared by all the worker threads. You can assume it is a single shared task queue that is usually used in normal ExecutorServices. Each DEQueue is belongs to one worker thread who will be the owner for that queue. Every time `fork` is called on a task will be pushed into its own queue. Each thread repeatedly removes a task from its own DEQueue and runs it. DEQueue support three functions: *push*, *pop* and *poll* where push and pop methods will be called by owner thread only and poll will be called by other threads. If a thread discovers its queue is empty then it becomes a theaf: it chooses a victive thread at random and calls that queue's poll method to steal a task for itself. This process is called *work stealing*.

Initially , only a single thread in a ForkJoinPool will be busy when you submit a task. The thread will begin to subdivide the larger task into smaller tasks. Each time a task is divided into two or more tasks, we fork the every new subtask except the last one we compute. After the computation we invoke join to wait for the forked tasks to complete. This divide-and-conquer process continues untill all the tasks are executed, and all the queues become empty. Mote generally this work stealing algorithm is used to redistribute and balance the tasks among the worker threads in the pool. Below figure shows how this process occurs.
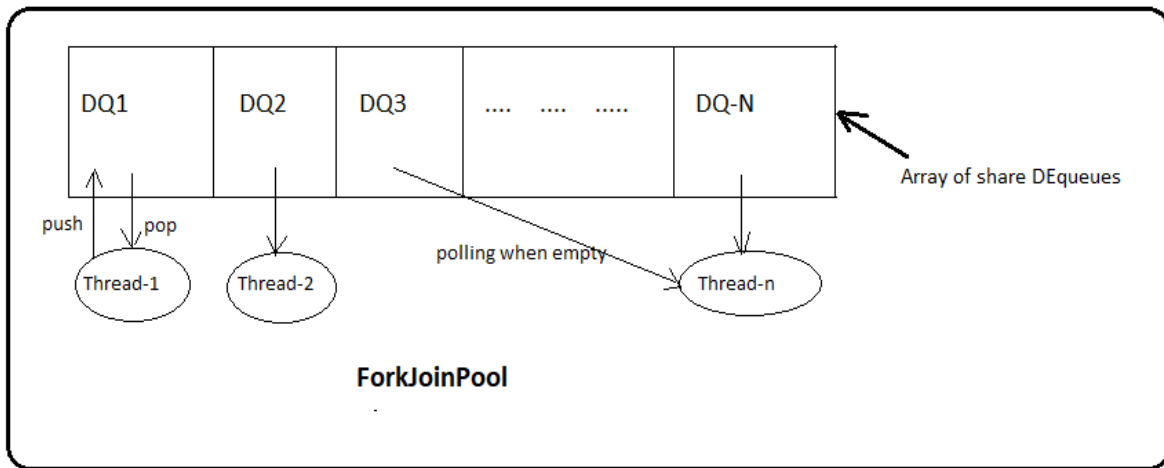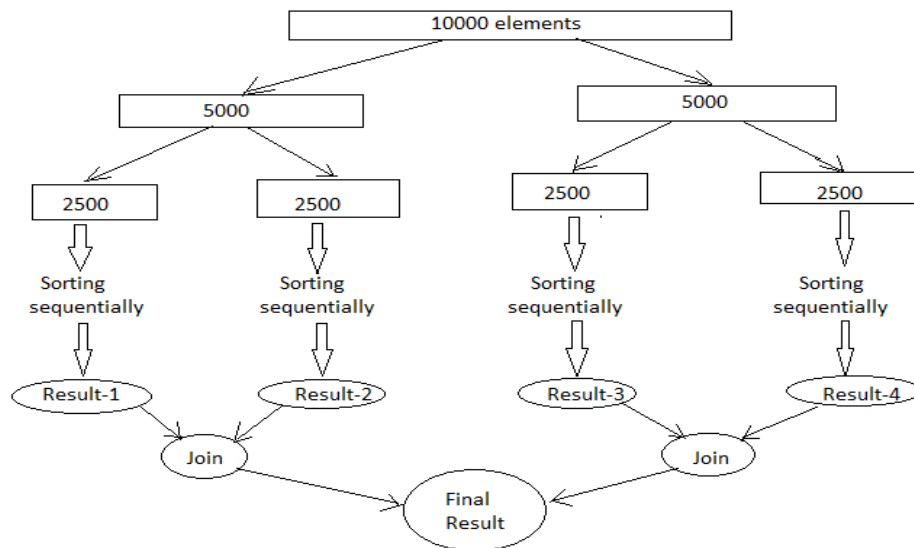
Fig. 12.1: **Internals of ForkJoinPool**

Fig. 12.2: **Splitting of tasks**

Parallel Data Processing

Compare to earlier days, cost of the hardwares have been reduced and the number of processors (cores) in modern computers are also increased. Each core has ability to perform operations independently. Before Java 7, processing a collection of data in parallel was extreamly cumbersome. First you have to split the complete data set into sub parts and asks the threads to execute them parallelly. In the last chapter we saw how ForkJoinPool perform these operations more consistently and in less error-prone way. To gain better understanding of prallel processing it is important to know how parallel stream works internally. I will strongly recommend you to go through fork-join-pool chapter if you have missed it.

Parallel Streams

A *parallel* Stream is a stream that splits its elements into multiple chunks, process each chunk with different thread. Thus you can automatically partition the workload of a given operation on all the cores of your multicore processor and keep all of them equally busy. Getting parallel stream is very easy, calling `parallelStream()` method on collection classes or `parallel()` method on sequential stream returns a parallel stream as demonstrated below.

```
List<String> list = getDataSet();
list.parallelStream().forEach(System.out::println);

int[] array = {1, 2, 3, 4, 5};
int sum = Arrays.stream(arr).parallel().sum();
```

Similarly stream also has `sequential()` method that converts parallel stream into sequential stream. In reality stream class maintains an internal boolean state to identify the stream is a parallel stream. Due to this calling *parallel()* and *sequential()* methods multiple times on a stream will not throw any exception. In the below example the last call parallel() will win the priority.

```
stream.parallel()
   .filter(...)
   .sequential()
   .map(...)
   .parallel()
   .sum();
```

By now you already have idea that tasks are divided and processed individually in parallel stream. Now let's see how parallel stream internally works. To understand it better we will see following example to find largest element in an array.

```
int max = numbers.parallelStream().reduce(0, Integer::max, Integer::max);
System.out.println("Parallel: " + max);
```

Here to the reduce method we are passing a BiFunction (2nd argument) which denominates the task to be performed when the task become too small and can be executed without splitting again. The last argument is a BinaryOperator shows the action should taken on the two partial results collected from sub tasks. If you want to know about *Stream.reduce* method please refer the Stream chapter. Below is the call stack of parallelStream() method.

parallelStream()
     StreamSupport.stream(spliterator(), true);
        ArrayList.spliterator()
           ArrayListSpliterator<>();

Parallelstream() calls `spliterator()` on the collection object which returns a Spliterator implementation that provides the logic of splitting a task. Every source or collection has their own spliterator implementations. Using these spliterators, parallel stream splits the task as long as possible and finally when the task becomes too small it executes it sequentially and merges partial results from all the sub tasks.

Spliterator

Spliterator is the new interface introduced in jdk8 that traverses and partitions elements of a source. The name itself suggests that, these are the iterators that can be splitted as and when require. As like Iterator, Spliterator is also used for traversing elements but meant to be used within stream only. Spliterator has defined some important methods that drives both sequential and parallel stream processing.

```
public interface Spliterator<T> {

   boolean tryAdvance(Consumer<T> action);
   default void forEachRemaining(Consumer<T> action);
   Spliterator<T> trySplit();
   long estimateSize();
   int characteristics();
}
```

- **tryAdvance** method is used to consume an element of the spliterator. This method returns either true indicating still more elements exist for processing otherwise false to signify all the elements of the spliterator is processed and can be exited.

- **forEachRemaining** is a default method available in Spliterator interface that indicates the spliterator to take certain action when no more splitting require. Basically this performs the given action for each remaining element, sequentially in the current thread, until all elements have been processed.

  ```
  default void forEachRemaining(Consumer<T> action) {
     do {
  ```

```
        } while (tryAdvance(action));
    }
```

If you see the `forEachRemaining` method default implementation, it repeatedly calls the *tryAdvance* method to process the spliterator elements sequentially. While splitting task when a spliterator finds itself to be small enough that it can be executed sequentially then it calls *forEachRemaining* method on its elements.

- **trySplit** is used to partition off some of its elements to second spliterator allowing both of them to process parallelly. The idea behind this splitting is to allow balanced parallel computation on a data structure. These spliterators repeatedly calls *trySplit* method unless spliterator returns null indiacating end of splitting process.

- **estimateSize** returns an estimate of the number of elements available in spliterator. Usually this method is called by some forkjoin tasks like *AbstractTask* to check size before calling trySplit.

- **characteristics** method reports a set of characteristics of its structure, source, and elements from among OR-DERED, DISTINCT, SORTED, SIZED, NONNULL, IMMUTABLE, CONCURRENT, and SUBSIZED. These helps the Spliterator clients to control, specialize or simplify computation. For example, a Spliterator for a Collection would report SIZED, a Spliterator for a Set would report DISTINCT, and a Spliterator for a SortedSet would also report SORTED.

You saw detailed descriptions on spliterator defined methods, now we will see a complete example that will deliver more context on how does they work.

```java
 1  public class SpliteratorTest {
 2
 3      public static void main(String[] args) {
 4          Random random = new Random(100);
 5          int[] array = IntStream.rangeClosed(1, 1_000_000).map(random::nextInt)
 6                              .map(i -> i * i + i).skip(20).toArray();
 7          int max = StreamSupport.stream(new FindMaxSpliterator(array, 0, array.length -
     →1), true)
 8                              .reduce(0, Integer::max, Integer::max);
 9          System.out.println(max);
10      }
11
12      private static class FindMaxSpliterator implements Spliterator<Integer> {
13          int start, end;
14          int[] arr;
15
16          public FindMaxSpliterator(int[] arr, int start, int end) {
17              this.arr = arr;
18              this.start = start;
19              this.end = end;
20          }
21
22          @Override
23          public boolean tryAdvance(Consumer<? super Integer> action) {
24              if (start <= end) {
25                  action.accept(arr[start]);
26                  start++;
27                  return true;
28              }
29              return false;
30          }
31
32          @Override
33          public Spliterator<Integer> trySplit() {
34              if (end - start < 1000) {
```

```
35              return null;
36          }
37
38          int mid = (start + end) / 2;
39          int oldstart = start;
40          start = mid + 1;
41          return new FindMaxSpliterator(arr, oldstart, mid);
42      }
43
44      @Override
45      public long estimateSize() {
46          return end - start;
47      }
48
49      @Override
50      public int characteristics() {
51          return ORDERED | SIZED | IMMUTABLE | SUBSIZED;
52      }
53   }
54 }
```

The FindMaxSpliterator is trying to find out the largest element in an array. Every time *trySplit* method checks the remaining size of the elements in current spliterator and creates a second spliterator if size is more than 100. Once the elements size reaches under 1000, it calls *tryAdvance* method repeatedly on those 1000 (may be less) elements.
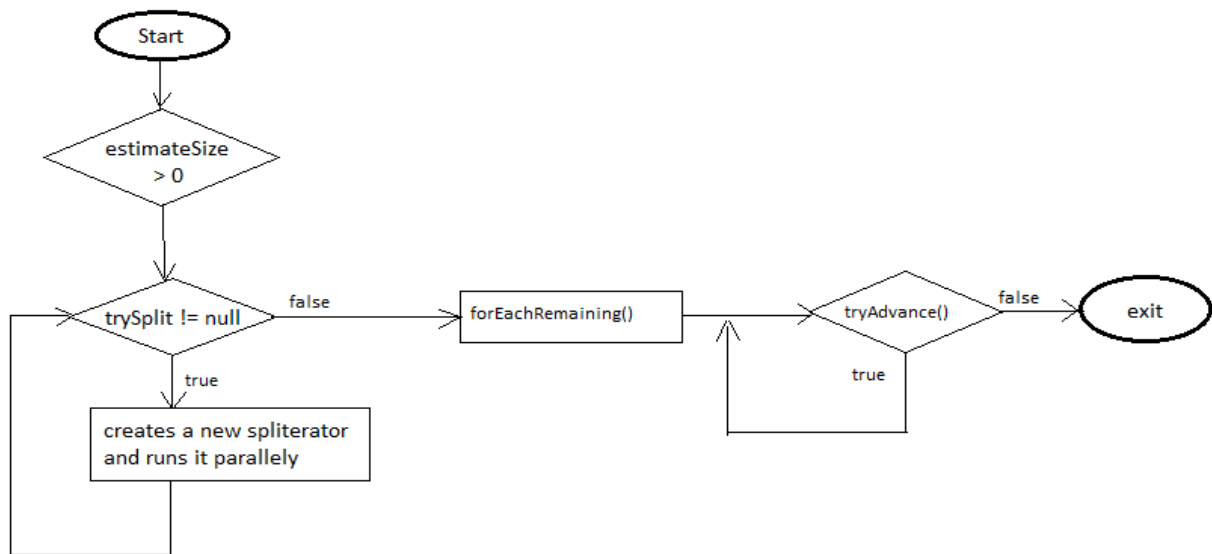

Fig. 13.1: **Spliterator Workflow**

Conclusion

Parallel stream make use of both ForkJoinPool and Spliterator to process elements parallelly. It is not the wise decision to use parallel stream all the time without comparing running time between sequential and parallel processing. In the above example we have considered 1_000_000 number of elements which is quite huge and can make sense if

executing in parallel, but suppose there were only 5000 elements then parallel stream will give you higher running time compared to sequential because it also includes the time taken for spitting and merging the partial results.

Evolution of date time API

Working with dates in Java was challenging tasks from the day one. Java 1.0 started with `java.util.Date` class to support date functionality but it had several problems and limitations. Despite its name, this class doesn't represent a date but a specific instant in time with millisecond precision. Its hazy design decision of using offsets: the year starts from 1900 and months are zero index based were misleading to the users. As an example if you want to represent March 18, 2014, we have to create instance of Date as follows.

Date date = new Date(114, 2, 18);

Here in the year field we have to pass year as 114 (2014-1900) and 2 as 3rd month which are quite confusing. Date had some of getXXX, setXXX methods to interpret dates as year, month, day, hour, minute, and second values and a `Date(String)` for parsing of date strings. Unfortunately, the API for these functions were not easy for internationalization. Another problem could be the arguments given to the Date API methods don't fall within any specific ranges; for example, a date may be specified as January 32 and is interpreted as meaning February 1. There is no explicit control over it.

To overcome all these limitations many of Date class methods were deprecated and `java.util.Calendar` class was introduced in Java 1.1, but it still couldn't meet the expectations. It solved some of Date class issues; internally handling offset values: passing 2014 as year rather than passing 114, daling with localization etc, but it has introduced some other problems. Calendar has similar problems and design flaws given below that lead to the error prone code.

- Constants were added in Calendar class but still month is zero index based.

- Calendar class is mutable so thread safety is always a question for it.

- It is very complicated to do date calculations. In fact, there is no simple, efficient way to calculate the days between two dates.

- `java.text.DateFormat` were introduced for the purpose of parsing of date strings but it isn't thread-safe. Following example shows the serious problem can occure when DateFormat is used in multi threaded scenarios.

```java
SimpleDateFormat sdf = new SimpleDateFormat("ddMMyyyy");
ExecutorService es = Executors.newFixedThreadPool(5);
for (int i = 0; i < 10; i++) {
    es.submit(() -> {
        try {
            System.out.println(sdf.parse("15081947"));
```

```
        } catch (ParseException e) {
            e.printStackTrace();
        }
    });
}
es.shutdown();


Output:
-------
Fri Aug 15 00:00:00 IST 1947
Mon Aug 11 00:00:00 IST 1947
Fri Aug 15 00:00:00 IST 1947
Fri Aug 15 00:00:00 IST 1947
```

If you run the above code multiple times then you will see unexpected behaviors. The existing Java date and time classes are poor, mutable, and have unpredictable performance. Some of the third-party libraries, such as *Joda-Time* showed his interest to overcome the issues with both Date and Calendar classes and it become so popular that it won the attention of Java core development team to include similar features to the Java core API.

JSR 310 defines the specifications for new Date and Time API to tackle the problem of a complete date and time model, including dates and times (with and without time zones), durations and time periods, intervals, formatting and parsing. Project ThreeTen was created to integrate JSR 310 into JDK 8. The goals of new Date Time API are:

- Support standard time concepts including date, time, instant, and time-zone.

- Immutable implementations for thread-safety.

- Provide an effective API suitable for developer usability.

- Provide a limited set of calendar systems and be extensible to others in future.

Java 8 introduced a new package java.time to provide a high quality date and time support in the native Java API.

java.time package

java.time package contains many classes to represent basic date-time concepts: instants, durations, dates, times, time-zones and periods based on ISO calendar system. All the classes are immutable and thread-safe. Following are nested packages available in java.time package.

| Package	Description
java.time.temporal	Each date time instance is composed of fields. This package contains lower level access to those fields.
java.time.format	Provides classes to print and parse dates and times. Instances are generally obtained from DateTimeFormatter, however DateTimeFormatterBuilder can be used if more power is needed.
java.time.chrono	This is intended for use by applications that need to use localized calendars. It contains the calendar neutral API ChronoLocalDate, ChronoLocalDateTime, ChronoZonedDateTime and Era. Actually the main API is build on ISO-8601 calendar system. However, there are other calendar systems: Hijrah Calendar, Japanese Calendar, Minguo Calendar, Thai Buddhist Calendar also exist for which this package provide support.
java.time.zone	This package provides support for time-zones, their rules and the resulting gaps and overlaps in the local time-line typically caused by Daylight Saving Time.

Common methods

Java 8 includes a large number of classes representing different aspects of dates like LocalDate, LocalTime, Local-DateTime, Instant, Duration and Period. These classes provides wide set of methods that will serve most of date time usecases. You will find many similar method prefixes to maintain the consistency and easy to remember. For example:

Method	Description	Example
of	It is a static factory method to create instance using the required individual field values.	LocalDate.of(year, month, day)
from	Static factory method to create instance from another date-time aspect. It will throw `DateTimeException` if unable to create instance.	LocalDate.from(LocalDateTime.now())
to	converts this object to another type	LocalDateTime.toLocalDate() Instant.toEpochMilli()
parse	Static factory method to create instance from string.	LocalDate.parse("2016-07-12")
get	gets the value of something.	Period.get(ChronoUnit.YEARS)
with	the immutable equivalent of a setter.	**LocalDateTime.now()** .withYear(2016) .withDayOfMonth(20);
plus	adds an amount to an object	duration.plusHours(5);
minus	subtracts an amount from an object	localdate.minusDays(2) instant.minusMillis(1000)

LoalDate, Time, Instant

Following diagram represents the class heirerchy for LocalDate, Time, Instant classes. *TemporalAccesssor* is the base interface defines the read-only access to a temporal object, such as a date, time, offset or some combination of these. *Temporal* interface defines the write access that will manipulate objects using plus and minus operations. We will gradually explore different temporal implementations individually.
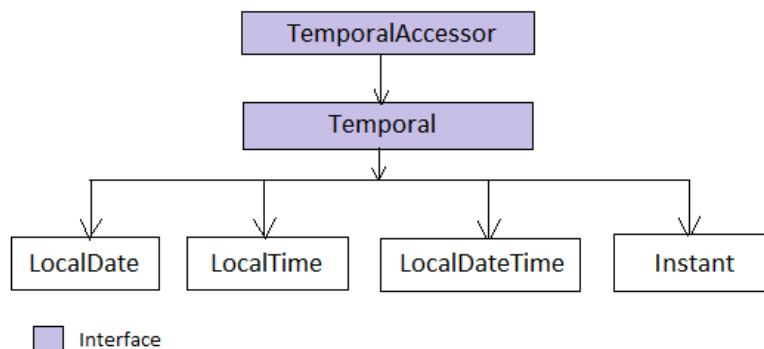
LocalDate LocalDate is an immutable object that represents a plain date with out time of day. It doesn't carry any information about the offset or time zone. It stores the date in YYYY-MM-DD format, for example '2014-03-18'. As I mentioned in the *Common methods* section, LocalDate instance can be created in many ways.

```
LocalDate.of(2015, 03, 18);      -- When individual values know
LocalDate.parse("2015-03-18");   -- Creating from date string

LocalDate.now();                 -- To get the current date.
LocalDate.now(ZoneId.of("America/Chicago"));
```

It also provides additional methods to retrieve its field informations such as Day, Month, Year, Era etc as shown in below example.

```
1. LocalDate date = LocalDate.now();
2. date.getMonth();
3. date.getDayOfYear();
4. date.get(ChronoField.YEAR);
```

If you see into line #4, it contains a generic `get` method that accepts *TemporalField* type and returns the field value. TemporalField is an interface and java 8 has `ChronoField` enum class to hold available temporal field types.

LocalTime Similar to LocalDate class, LocalTime represents only time of the day. It also doesn't hold time zone details. It stores the time in HH:mm:ss.nano_seconds format, for example '04:30:15.123456789'. This class also contain similar set of methods including accessing field values such as `getHour`, `getMinute`.

```
LocalTime.of(4, 30, 15);
LocalTime.parse("04:30:15.12345");

LocalTime.now();
LocalTime.now(ZoneId.of("America/Chicago"));

date.getMinute();
date.getNano();
date.get(ChronoField.HOUR_OF_DAY);
```

LocalDateTime LocalDateTime is the combination of LocalDate and LocalTime that holds both date and time parts with out time zone details. The format of stored data is 2007-12-03T10:15:30 whete 'T' is the delimiter between date and time values. Most of the LocalDate and LocalTime methods are applicable to LocalDateTime class. It also contains methods to get LocalDate and LocalTime instances.

```
LocalDateTime.now();
LocalDateTime.getDayOfWeek();
LocalDateTime.parse("2007-12-03T10:15:30");

date.toLocalDate();
date.toLocalTime();
```

Instant Instant is a point on a continuous time line or scale. Basically this represents the number of seconds passed since the Epoch time 1970-01-01T00:00:00Z. Internally Instant stores two values, one long value representing epoch-seconds and an int representing nanosecond-of-second, which will always be between 0 and 999,999,999. Any date-time after 1970-01-01T00:00:00Z will return positive value and before will be negative value.

```
1. Instant.now();
2. Instant.now().getEpochSecond();

3. Instant.parse("1969-01-01T00:00:00.00Z").getEpochSecond();  --> -31,536,000
4. Instant.parse("1971-01-01T00:00:00.00Z").getEpochSecond();  --> 31,536,000
```

Here in line #3 we have supplied one year before epoch time so it is returning a negative long value (1*365*24*60*60 = 31,536,000 secs). Similarly in line #4, given date-time is next year of the epoch time

so the result is a positive long value.

Duration & Period

In the previous section you saw, LocalDate, LocalTime used to work with date and time aspects. Beyond dates and times, the API also allows the storage of periods and durations of time. With the Date and Calendar class it is complicated to do date calculation like days between two dates so duration and period provide solutions for these kind of usecases.

Both Duration and Period class implements `TemporalAmount`. It is the base interface to represent amount of time. This is different from a date or time-of-day in that it is not tied with any point on time-line or scale, it is as simple as amount of time, such as "6 hours", "8 days" or "2 years and 3 months". As like TemporalField, Java API also provides `TemporalUnit` interface to measure time in units of years, months, days, hours, minutes and seconds. `ChronoUnit` is the enum that implements TemporalUnit interface which will be used by the end users.

Duration Duration holds quantity or amount of time in terms of seconds and nanoseconds. Along with these two, it provides some `toXXX` methods to access other fields: hours, minutes, millis, days. It also provides a highly used utility method `between` to calculate duration among two temporal objects.

```
1  LocalDateTime d1 = LocalDateTime.parse("2014-12-03T10:15:30");
2  LocalDateTime d2 = LocalDateTime.parse("2016-03-05T23:15:00");
3  Duration duration = Duration.between(d1, d2);
4  duration.toHours();
5  duration.toDays();
6
7  Duration.between(d1.toLocalTime(), d2).toHours();   -> 12
8  Duration.between(d1, d2.toLocalTime()).toHours();   -> DateTimeException
9
10 Duration.between(d1.toLocalDate(), d2.toLocalDate());  -> DateTimeException
```

If you have marked line #8 is throwing DateTimeException. The reason is when two different temporal objects are passed then the duration is calculated based on the first temporal object. Here the socond argument Local-Time tries to be coverted into LocalDateTime and the convertion failed. One another characteristic of between method is to accept temporal object that supports seconds or nanoseconds due to which line #10 will also throw DateTimeException.

Period Period represents amount of time in terms of years, months and days. It provides some `getXXX` methods to access these fields. Along with field accessing methods it also provides similar methods contained in Duration class.

```
LocalDate date1 = LocalDate.parse("2010-01-15");
LocalDate date2 = LocalDate.parse("2011-03-18");

Period period = Period.between(date1, date2);
period.getYears();      -> 1
period.getMonths();     -> 2
period.getDays();       -> 3
```

Important point to notice here is getMonths and getDays method doesn't return the number of months or days between these two dates, it is just the numeric value difference between two months and two days. If you want total number of days or months between these dates then use `LocalDate.until(temporal, unit)`.

Example: date1.until(date2, ChronoUnit.DAYS)

TemporalAdjusters

New Date Time API provides numerous methods: plusHour, minusWeek, withYear, withDays to manipulate temporal objects. Sometime we need to perform advanced operations such as finding next working day for a software firm considering its holiday calendar. One solution is to write temporal object modification logic wherever require in your code but this will cause code repeatation. To help with these scenarios Java 8 provides an interface `TemporalAdjuster` to externalize temporal adjustment logic. It has only one abstract method `Temporal adjustInto(Temporal)` that takes an existing temporal object and returns a manipulated temporal. Java recommends not to alter the original input temporal object for the thread safety.

The framework interface *Temporal* defines an overloaded version of `with(TemporalAdjuster)` method that takes *TemporalAdjuster* as input and returns a new temporal object.

```
default Temporal with(TemporalAdjuster adjuster) {
    return adjuster.adjustInto(this);
}
```

Remember we can directly call `adjuster.adjustInto(temporal)` but is recommended by Java core development team to use the first approach for the sake of maintaining code readability. Java 8 also provides a utility class `TemporalAdjusters` that defines most of common adjustment implementations. Suppose to find out the next sunday after the java 8 release date.

```
LocalDate date = LocalDate.parse("2014-03-18");
TemporalAdjuster adjuster = TemporalAdjusters.nextOrSame(DayOfWeek.SUNDAY);
System.out.println(date.with(adjuster));
```

Below table shows the API provided temporal adjusters. For all these adjusters we will use `LocalDate.parse("2014-03-18")` for demonstrating examples.

Method	Description & Example
dayOfWeekInMonth	Returns an adjuster representing temporal instance of the given dayOfWeek that is the nth occurance in the month. // 4th monday in the month (2014-03-24) date.with(dayOfWeekInMonth(4, DayOfWeek.MONDAY)); // 2nd Sunday in the month (2014-03-09) date.with(dayOfWeekInMonth(2, DayOfWeek.SUNDAY)); // 8th Friday in the month (2014-04-25) date.with(dayOfWeekInMonth(8, DayOfWeek.FRIDAY)); It is not possible to have 8th Friday in any of the month, so here next subsequent months will also be considered.
firstDayOfMonth	Returns the adjuster that in turn returns temporal object representing first day of the month. date.with(firstDayOfMonth()); => 2014-03-01
firstDayOfNextMonth	Returns the adjuster that in turn returns temporal object representing first day of the next month. LocalDate date = LocalDate.parse("2014-12-03"); date.with(firstDayOfNextMonth()); => 2015-01-01
firstDayOfNextYear	Adjuster to return temporal object representing first day of the next year. date.with(firstDayOfNextYear()) => 2015-01-01
firstDayOfYear	Adjuster to return temporal object representing first day of the given date year. date.with(firstDayOfYear()) => 2014-01-01
firstInMonth	Adjuster to return temporal object representing first occurance of given day in the month. date.with(firstInMonth(DayOfWeek.MONDAY)) => 2014-08-04
lastDayOfMonth	Returns the adjuster that in turn returns temporal object representing last day of the month. date.with(lastDayOfMonth()) => 2014-08-31
lastDayOfYear	Adjuster to return temporal object representing last day of the given date year. date.with(lastDayOfYear()) => 2014-12-31
lastInMonth	Adjuster to return temporal object representing last occurance of given day in the month. date.with(lastInMonth(DayOfWeek.MONDAY)) => 2014-08-25
next	Adjuster to return next occurance of given day. date.with(next(DayOfWeek.FRIDAY)) => 2014-08-08
nextOrSame	Returns the next-or-same day-of-week adjuster, which adjusts the date to the first occurrence of the specified day-of-week after the date being adjusted unless it is already on that day in which case the same object is returned. date.with(lastInMonth(DayOfWeek.SUNDAY)) => 2014-08-03 "2014-08-03" is a SUNDAY, so returned the same date.
previous	Adjuster to return previous occurance of given day. date.with(previous(DayOfWeek.MONDAY)) => 2014-07-28
previousOrSame	Same as previous method but considers current given date also. date.with(previousOrSame(DayOfWeek.SUNDAY)) => 2014-08-25

14.5. TemporalAdjusters

Apart from above methods, TemporalAdjusters also contains a generic method `ofDateAdjuster(UnaryOperator<LocalDate> adjuster)` to hold the custom logic. User can pass a lambda by wrapping their own date manipulation logic. Below example shows a custom TemporalAdjuster implementation for finding next working day.

```java
TemporalAdjuster nextWorkingday = temporal -> {
    LocalDate date = (LocalDate) temporal;
    DayOfWeek day = date.getDayOfWeek();
    if (DayOfWeek.FRIDAY.equals(day) || DayOfWeek.SATURDAY.equals(day)) {
        return date.with(next(DayOfWeek.MONDAY));
    } else {
        return date.plusDays(1);
    }
};

System.out.println(LocalDate.now().with(nextWorkingday));
```

Formatting & parsing

Formatting and parsing are must required features of date time API that does the convertion between string and date. In the begining we saw one of the major issue with the old DateFormat class is the thread safety. The Date Time API has introduced a new package *java.time* to support parsing and formatting with new thread safe date time classes. This package has two basic classes DateTimeFormatter and DateTimeFormatterBuilder where most of the time we will be using DateTimeFormatter class.

DateTimeFormatter: This class is the replacement for java.text.DateFormat which provides two main methods; format(temporal) to convert temporal object to string and parse(string) to create a temporal object from the given date string. Creating DateTimeFormatter instance is easy, it provides overloaded `ofPatttern` methods to create it instances.

```java
DateTimeFormatter f1 = DateTimeFormatter.ofPattern("dd-MMM-yyyy");
LocalDate date = f1.parse("18-Mar-2014");
f1.format(LocalDate.of(2014, 3, 18));  =>  18-Mar-2014

//For localization
DateTimeFormatter f2 = DateTimeFormatter.ofPattern("dd-MMM-yyyy", Locale.FRENCH);
f2.format(LocalDate.of(2014, 3, 18));  =>  18-mars-2014
```

DateTimeFormatter class also contains many of its own instances like ISO_LOCAL_DATE, ISO_LOCAL_DATE_TIME, BASIC_ISO_DATE etc that can be used for our general usecases.

DateTimeFormatterBuilder: This class is used to create DateTimeFormatters. If you hook into DateTimeFormatter source code you will see ultimately they are created using the builder class. This class will be rarely used in case of complex needs so we will not focus much on this. Below code snippet taken from the java source code to show the implementation of ISO_LOCAL_DATE instance.

```java
ISO_LOCAL_DATE = new DateTimeFormatterBuilder()
            .appendValue(YEAR, 4, 10, SignStyle.EXCEEDS_PAD)
            .appendLiteral('-')
            .appendValue(MONTH_OF_YEAR, 2)
            .appendLiteral('-')
            .appendValue(DAY_OF_MONTH, 2)
            .toFormatter(ResolverStyle.STRICT, IsoChronology.INSTANCE);
```

Working with time zones

One of the confusing aspects of date time is working with time zones. Till Java 7 java.util.TimeZone can be used togather with Calendar class but JDK 8 now introduced quite few classes to simplify the usage and gives better options.

Class	Description
ZoneID	Defines a unique id for a region and city combination. For example Asia/Kolkata
ZoneOffset	Represents timezone with an offset from Greenwich/UTC, such as +05:30.
ZonedDate-Time	Represents a date time in the ISO-8601 calendar system with time zone such as 2007-12-03T10:15:30+01:00 Europe/Paris
OffsetDate-Time	A date-time with an offset from UTC/Greenwich in the ISO-8601 calendar system, such as 2007-12-03T10:15:30+01:00.
OffsetTime	A time with an offset from UTC/Greenwich in the ISO-8601 calendar system, such as 10:15:30+01:00.
ZoneRule-sProvider	Provides time zone rules.

Time zones and Offsets:

Java uses the Internet Assigned Numbers Authority (IANA) public domain database of time zones, which keeps a record of all known time zones around the world and is updated periodically to reflect changes made by political bodies to time zone boundaries, UTC offsets, and daylight-saving rules. You can find a nice video on DayLight Saving here. java.time.ZoneId represents a time zone with a unique id identified by continent or ocean and then by the name of the location, which is typically the largest city within the region. For example, America/New_York represents most of the US eastern time zone.

There are basically three types of zone ids.

- The first type is the offset from UTC/GMT time. They are represented by the class ZoneOffset and they consist of digits starting with + or -, for example, +05:30 giving hints that perticular time zone is 5:30 hours ahead of GMT.

- The next type ids are also offsets but they started with some recognised prefixes: 'UTC', 'GMT' and 'UT'. Same with first one they also represented by ZoneOffset class.

- The third type is region based. These are in the format area/city, for example, Asia/Kolkata.

You can create a ZoneId instance using `ZoneId.of(String zoneId)` factory method. Usually this returns its subclass instance ZoneOffset or ZoneRegion depending upon the input given. ZoneOffset class has factory method that can directly create its instance from the offset value.

```
ZoneId zone = ZoneId.of("Asia/Kolkata");
ZoneOffset zone2 = ZoneOffset.of("+05:30");
```

ZoneId consists of ZoneRules that defines rules for that time zoneIt is not recommended to use ZoneOffset as they don't contain daylight saving details if a country or city supporting it. ZoneId class also provides a method `ZoneId.getAvailableZoneIds()` that returns all available time zones. These time zones are usually supplied by ZoneRulesProvider class. You can register your own time zones by registering a custom provider.

```
public class MyZoneRulesProvider extends ZoneRulesProvider {

    @Override
    protected Set<String> provideZoneIds() {
        Set<String> set = new HashSet<>();
        set.add("India/Delhi");
        set.add("India/Mumbai");
```

```
        set.add("India/Chennai");
        return set;
    }

    public static void main(String[] args) {
        ZoneRulesProvider.registerProvider(new MyZoneRulesProvider());
        ZoneId.getAvailableZoneIds().stream().forEach(System.out::println);
    }
}
```

Each ZoneId consists of ZoneRules that defines rules for that time zone. *ZoneId.getRules()* will returns the rules.

ZonedDateTime: As like LocalDateTime, ZonedDateTime stores date and time fields, but additionally contains time zone information. You can combine ZoneId with temporal objects to transform it into ZonedDateTime or can use overloaded `of` methods to create its instance.

```
1  ZoneId zone = ZoneId.of("Asia/Kolkata");
2
3  LocalDateTime dateTime = LocalDateTime.parse("2014-12-03T10:15:30");
4  ZonedDateTime z11 = dateTime.atZone(zone);
5  ZonedDateTime z12 = ZonedDateTime.of(dateTime, zone);
6
7  LocalDate date = LocalDate.of(2014, 3, 18);
8  ZonedDateTime z21 = date.atStartOfDay(zone);
9  ZonedDateTime z22 = ZonedDateTime.of(date, LocalTime.now(), zone);
10
11 Instant instant = Instant.now();
12 ZonedDateTime z31 = instant.atZone(zone);
13 ZonedDateTime z32 = ZonedDateTime.ofInstant(instant, zone);
```

OffsetDateTime As we saw time zones are also represented by an offset value from UTC, OffsetDateTime represents an object with date/time information and an offset, for example, 2014-12-03T11:30-06:00. Instant, OffsetDateTime and ZonedDateTime are very much looks similar but there are key differences exists. Instance represents a point in time in UTC on a continuous time line, OffsetDateTime maintains time zone with an offset compared to UTC and ZonedDateTime contains time zone information along with Day-Light-saving rules. It is always better to use ZonedDateTime or Instant for simple usages. As like other temporal instances, this also has standard method patterns to create its instances.

```
ZoneOffset offset = ZoneOffset.of("-2");

OffsetDateTime.now(offset);
OffsetDateTime.of(LocalDateTime.now(), offset);
OffsetDateTime.of(LocalDate.now(), LocalTime.now(), offset);
OffsetDateTime.ofInstant(Instant.now(), offset);
```

Similar to OffsetDateTime, Java 8 also provides an OffsetTime class that contains time with an offset from UTC/Greenwich, such as 10:15:30+01:00.

CHAPTER 15

Indices and tables

- genindex
- modindex
- search