

JAVA / LESSON 30 OF 40

The Block Data Pipeline

Use Lambda and Stream to generate mining briefing

This chapter uses JDK 21 and does not require the preview function. | 100-120 minutes

Create mining statistics briefing MiningReport

The mine car brings a batch of records: screening, conversion, sorting, statistics, each through a section of pipeline. Organize the data with clear behavioral rules, and then calculate the answers by hand so that there is evidence for each number in the briefing.

Prerequisites: List, Map, record, method reference and loop. Today, let the mining data ride on the mine cart and pass through screening, processing, sorting and statistics stations in sequence.

Before you start: It is recommended to complete Lessons 24-29 first

Read the full lesson and copy the code:

<https://teacher-ma.de/minecraft-java-lesson.php?lesson=30&lang=en>

Long code lines may wrap at the > margin marker. The marker is not part of the code.

Parameter arrow expression writes temporary rules

Element	Explanation
Lambda	Passing behavior as value
Predicate	One parameter, returns true or false
BiFunction	Two parameters, return result

In this lesson

01. Before starting: predict first, then let the program answer
02. Lambda is a short piece of behavior
03. Stream pipeline is divided into three sections
04. filter only retains matching elements
05. map changes elements into another form
06. sorted control reporting order
07. Numerical flow is responsible for summation and averaging
08. distinct limit and skip
09. toList catches the final result
10. Stream can only be consumed once
11. Avoid modifying external collections
12. Complex Lambda is extracted as a method
13. Challenge Blueprint Mining Statistical Briefing
14. Mining Statistics Program
15. Expansion tasks and unit review
16. Own acceptance record

01 / 16

Before starting: predict first, then let the program answer

This chapter uses JDK 21 and does not require the preview function. The story is borrowed from the block world, and the example runs on the ordinary Java console; if the values and rules are course models, they will be clearly stated.

Each common example is a complete independent file, saved according to the file name marked on the box. Place each example in a different folder to avoid duplicate names of auxiliary classes. When the complete project is divided into sections, they are connected sequentially in the same file.

Terminal command: replace the Chinese placeholder with the actual file name and class name

```
javac -encoding UTF-8 ExampleFile.java
java ExampleClass
```

- Prediction: Write the expected output first, or draw out the changes in references and data.
- Verification: Check line by line after running, and then only change one condition.
- Explanation: Explain whether the results come from Java rules, course contracts, or input data.

"Intentional error" is used to recognize bugs and cannot be mixed with ordinary examples. Use a special temporary directory for file experiments; do not use real archives to test coverage and deletion.

02 / 16

Lambda is a short piece of behavior

Lambda is not a new method, but a shorter transitive behavior. `ore -> ore.length() >= 2` means receiving ore and then answering whether the condition is true.

Parameter arrow expression writes temporary rules

Element	Explanation
Lambda	Passing behavior as value
Predicate	One parameter, returns true or false
BiFunction	Two parameters, return result

Read the arrow as "Give me the parameters on the left and I will execute the rules on the right." It's usually easier to understand if you write the complete method first and then reduce it to a Lambda.

L30Example01.java · Complete independent example

```
public class L30Example01 {  
    public static void main(String[] args) {  
        java.util.function.Predicate<String> longName = name -> name.length() >= 2;  
        java.util.function.BiFunction<Integer, Integer, Integer> sum = (x, y) -> x + y;  
        System.out.println(longName.test("diamond"));  
        System.out.println(sum.apply(3, 4));  
    }  
}
```

Output

```
true  
7
```

Lambda requires a functional interface: it has only one abstract method. Here Predicate accepts one parameter and returns true or false, and BiFunction accepts two parameters and returns a result; methods such as default are not included in this abstract method.

03 / 16

Stream pipeline is divided into three sections

Stream describes the data processing method: source provides elements, intermediate operation building steps, and terminates the operation request result. It does not save a copy of the data that can be read over and over again like a List does.

The source passes through intermediate operations and is finally initiated by the termination operation

Step	Explanation
1. Source	List.stream provides elements
2. Intermediate operations	filter、map、sorted
3. Terminate operation	toList、sum、count

Terminate operation request calculation result. Implementations may omit steps that do not affect the answer, and do not rely on side effects in filters or maps for logging or counting.

L30Example02.java · Complete independent example

```
import java.util.*;
public class L30Example02 {
    public static void main(String[] args) {
        List<String> ores = List.of("Coal", "diamond", "Iron ore");
        var pipeline = ores.stream().filter(name -> name.length() >= 2);
        System.out.println(pipeline.toList());
    }
}
```

Output

```
[Coal, diamond, Iron ore]
```

Intermediate operations are lazy; terminating operations request results. The implementation can also optimize and skip steps that do not affect the results. Do not use the printing and counting side effects in filter or map to determine whether they must be executed one by one.

Acceptance comparison table

Phase	Example	Effect
Source	ores.stream	Provide elements
Intermediate operations	filter map sorted	Description processing
Terminate operation	toList sum count	Produce results

04 / 16

filter only retains matching elements

filter expresses retention conditions: elements that meet the conditions enter the result. Filtering does not delete the original list members, nor should the source be secretly modified when judging conditions.

Predicate returns true records and continues to move forward

Element	Explanation
filter	Select elements by true or false
New results	Catch with termination operation
Original List	will not be deleted as a result

filter returns a new stream and does not delete elements from the original List. To save the results, you must catch the return value of the terminating operation.

L30Example03.java · Complete independent example

```
import java.util.*;
record MinedBlock(String ore, int amount, int value) {
    MinedBlock {
        if (ore == null || ore.isBlank() || amount < 0 || amount > 1_000_000
            || value < 0 || value > 1_000_000)
            throw new IllegalArgumentException("Check mineral name, quantity and unit price");
        ore = ore.strip();
    }
}
public class L30Example03 {
    public static void main(String[] args) {
        List<MinedBlock> blocks = List.of(new MinedBlock("Coal", 2, 1),
            new MinedBlock("diamond", 3, 10));
        var rare = blocks.stream().filter(b -> b.value() >= 5).toList();
        System.out.println(rare.size());
        System.out.println(blocks.size());
    }
}
```

Output

```
1
2
```

05 / 16

map changes elements into another form

map like workbench recipe. Each input element is converted once, the number usually remains the same, but the type can change.

Records can become names, tags or values

Element	Explanation
map	Convert elements into new forms
Return type	Can differ from source
HashMap	It is a key-value container, and the concept is different

Stream's map is a conversion operation and is not the same concept as HashMap. One is a verb and the other is a key-value container.

L30Example04.java · Complete independent example

```
import java.util.*;
public class L30Example04 {
    public static void main(String[] args) {
        List<String> ores = List.of("diamond", "iron");
        System.out.println(ores.stream().map(s -> s.toUpperCase(Locale.ROOT)).toList());
        System.out.println(ores.stream().map(String::length).toList());
    }
}
```

Output

```
[DIAMOND, IRON]
[7, 4]
```

Locale.ROOT makes the capitalization of English identifiers independent of the computer language. String.length counts UTF-16 code units, which is not equivalent to the number of visible characters in all text; there is no such difference in English in this example.

06 / 16

sorted control reporting order

Natural ordering is suitable for strings and numbers. Complex records should be clearly sorted by which field, and a second rule can be added when the scores are the same.

Comparator describes comparison rules

Element	Explanation
Sorting field	Whether to specify unit price or subtotal
Second rule	Same points according to mineral name
Original sequence	sorted does not modify the source

Stream sorted does not change the original list. It's perfect when you need new ordinal results, and it's easier to compare before and after data.

L30Example05.java · Complete independent example

```
import java.util.*;
record MinedBlock(String ore, int amount, int value) {
    MinedBlock {
        if (ore == null || ore.isBlank() || amount < 0 || amount > 1_000_000
            || value < 0 || value > 1_000_000)
            throw new IllegalArgumentException("Check mineral name, quantity and unit price");
        ore = ore.strip();
    }
}
public class L30Example05 {
    public static void main(String[] args) {
        List<MinedBlock> blocks = List.of(new MinedBlock("B", 1, 5),
            new MinedBlock("A", 1, 5), new MinedBlock("C", 1, 10));
        var names = blocks.stream()
            .sorted(Comparator.comparingInt(MinedBlock::value).reversed()
                .thenComparing(MinedBlock::ore))
            .map(MinedBlock::ore).toList();
        System.out.println(names);
        System.out.println(blocks.get(0).ore());
    }
}
```

Output

```
[C, A, B]
B
```

07 / 16

Numerical flow is responsible for summation and averaging

The object stream must first be mapped into a numerical stream before it can be directly summed. You can use `mapToInt` for quantities, and `mapToLong` for potentially large values; determine the value range first every time.

`mapToInt` converts the object into an integer stream

Element	Explanation
long multiplication	Convert first and then calculate
sum	The empty value stream is 0
average	There may be no result, use <code>orElse</code>

Numeric streams provide summing and averaging. The value in this chapter uses `mapToLong` and is converted before multiplication; the sum of the empty stream is 0, and the expression of average and max using `Optional` may have no result.

Convert the operand to long before multiplication. Do int multiplication first and then convert to long, the error has occurred. Only when both the single record and the total number of records have an upper limit can it be proved that the accumulation will not overflow.

L30Example06.java · Complete independent example

```
import java.util.*;
record MinedBlock(String ore, int amount, int value) {
    MinedBlock {
        if (ore == null || ore.isBlank() || amount < 0 || amount > 1_000_000
            || value < 0 || value > 1_000_000)
            throw new IllegalArgumentException("Check mineral name, quantity and unit price");
        ore = ore.strip();
    }
}
public class L30Example06 {
    public static void main(String[] args) {
        List<MinedBlock> blocks = List.of(new MinedBlock("diamond", 3, 10),
            new MinedBlock("Iron", 7, 3));
        long total = blocks.stream().mapToLong(b -> (long) b.amount() * b.value()).sum();
        System.out.println(total);
        System.out.println(blocks.stream().mapToInt(MinedBlock::amount).average().orElse(0));
        System.out.println(List.<MinedBlock>of().stream()
            .mapToLong(MinedBlock::amount).sum());
    }
}
```

Output

```
51
5.0
0
```

08 / 16

distinct limit and skip

Sort first and then limit to get the top three; first limit and then sort will only sort the first three of the original list. Pipeline order is algorithmic order.

Deduplication and interception shaping final report

Element	Explanation
distinct	Deduplication according to equality rules
sorted and limit	Change the answer in sequence
skip	Skip the specified amount

Run the same data after exchanging sorted and limit, and record the output differences. Don't think of piping steps as if they were just randomly arranged decorations.

L30Example07.java · Complete independent example

```
import java.util.*;
public class L30Example07 {
    public static void main(String[] args) {
        List<Integer> values = List.of(3, 1, 3, 2);
        System.out.println(values.stream().distinct().sorted().limit(2).toList());
        System.out.println(values.stream().limit(2).sorted().toList());
        System.out.println(values.stream().distinct().sorted().skip(1).toList());
    }
}
```

Output

```
[1, 2]
[1, 3]
[2, 3]
```

09 / 16

toList catches the final result

Java 21's `Stream.toList` returns an unmodifiable list. It can be read, but adding or removing it will fail.

The pipeline result is usually a new unmodifiable list

Element	Explanation
toList	Return to unmodifiable list
Variable copy	new ArrayList<>(result)
Element status	Container read-only does not mean depth is immutable

Do not assume that all methods that return a List allow modification. Read method contracts and create explicit mutable copies if necessary.

Fault experiment: toList returns an unmodifiable list, add throws UnsupportedOperationException. Errors are intentionally left below, please anticipate them first and then try to fix them.

An immutable list does not mean that the elements are not mutable. If the elements are mutable objects, their fields may still change; the MinedBlock in this chapter contains only Strings and ints, so the values themselves remain unchanged.

L30Example08.java · Complete independent example

```
import java.util.*;
public class L30Example08 {
    public static void main(String[] args) {
        List<String> result = List.of("Coal", "diamond").stream().toList();
        List<String> editable = new ArrayList<>(result);
        editable.add("Iron");
        System.out.println(result);
        System.out.println(editable);
    }
}
```

Output

```
[Coal, diamond]
[Coal, diamond, Iron]
```

BadToList.java · Intentional error example

```
import java.util.*;
public class BadToList {
    public static void main(String[] args) {
        var result = List.of("Coal").stream().toList();
        result.add("Iron");
    }
}
```

10 / 16

Stream can only be consumed once

Stream is like a one-time minecart route. After reaching the end point, you cannot go back to the starting point and run again.

The stream has been closed after the termination operation.

Element	Explanation
Disposable Stream	Can no longer be consumed after termination
Retain source	List can repeatedly create new streams
Report again	Restart from stream()

Usually data sources such as List are saved instead of putting Stream into fields for a long time. Each report builds a new pipeline from the source.

Fault experiment: The stream has been consumed, and an `IllegalStateException` is thrown when the operation is terminated again. Errors are intentionally left below, please anticipate them first and then try to fix them.

BadStreamReuse.java · Intentional error example

```
import java.util.*;
public class BadStreamReuse {
    public static void main(String[] args) {
        var stream = List.of("Coal", "Iron").stream();
        stream.count();
        stream.toList();
    }
}
```

L30Example09.java · Complete independent example

```
import java.util.*;
public class L30Example09 {
    public static void main(String[] args) {
        List<String> ores = List.of("Coal", "Iron");
        System.out.println(ores.stream().count());
        System.out.println(ores.stream().toList());
    }
}
```

Output

```
2  
[Coal, Iron]
```

11 / 16

Avoid modifying external collections

If the external List is secretly modified in `forEach`, the data flow direction is divided into two. Returning results is easier to read and test than hiding side effects.

Pipeline steps should return results based on input

Element	Explanation
Avoid hidden side effects	Return result expression collection
Do not modify the source	The traversal contract requires no interference
<code>forEach</code>	Suitable for actions that must be performed

Let the terminating operation explicitly return final data. Use `forEach` only when you really need to perform an action.

L30Example10.java · Complete independent example

```
import java.util.*;  
public class L30Example10 {  
    public static void main(String[] args) {  
        List<String> ores = List.of("Coal", "diamond", "Iron ore");  
        List<String> rare = ores.stream().filter(s -> s.length() >= 2).toList();  
        System.out.println(rare);  
    }  
}
```

Output

```
[Coal, diamond, Iron ore]
```

It is not a syntax error to add a sequential flow's `forEach` to a standalone mutable container, but the side effect makes refactoring and parallelization harder. Use return values to express collection results, and do not modify the source collection during traversal.

12 / 16

Complex Lambda is extracted as a method

After Lambda exceeds one or two lines, extract it into a named method. The main process is clearer and the rules can be tested individually.

Name Description Rules Tests can be run individually

Element	Explanation
Naming method	Explain complex selection rules
Method reference	Put named actions into pipes
Independent testing	Can be verified without the entire report

The pipeline shows the processing sequence, and the naming method explains each complex rule. Each of the two layers of information does one thing.

L30Example11.java · Complete independent example

```
import java.util.*;
record MinedBlock(String ore, int amount, int value) {
    MinedBlock {
        if (ore == null || ore.isBlank() || amount < 0 || amount > 1_000_000
            || value < 0 || value > 1_000_000)
            throw new IllegalArgumentException("Check mineral name, quantity and unit price");
        ore = ore.strip();
    }
}
public class L30Example11 {
    static boolean isRare(MinedBlock b) {
        return b != null && b.value() >= 5;
    }
    public static void main(String[] args) {
        List<MinedBlock> blocks = List.of(new MinedBlock("Coal", 2, 1),
            new MinedBlock("diamond", 3, 10));
        System.out.println(blocks.stream().filter(L30Example11::isRare)
            .map(MinedBlock::ore).toList());
    }
}
```

Output

```
[diamond]
```

Challenge Blueprint Mining Statistical Briefing

The challenge program is called MiningReport. Each record saves the mineral name, quantity, and single block value, and the Stream generates the total blocks, rare list, total value, and highest value mineral.

Filter, Transform, Sort and Aggregate to Compose Report

Element	Explanation
Total quantity	$14+7+4+3+20=48$
Total value	$14+21+20+30+0=85$
Highest subtotal	Diamond record is 30

Calculate the total quantity and total value first, and then run the program. What statistical programs fear most is that the output looks reasonable but no one knows the correct answer.

Acceptance conditions

- The total number of blocks is 48.
- Rare minerals are arranged from high to low in terms of single piece value.
- The total value is calculated by multiplying the quantity by the unit price.
- The highest value mineral is selected based on the total value of the entire record.
- When the list is empty, the highest value is displayed as None.

Acceptance comparison table

Minerals	Quantity	Unit price	Subtotal
Coal	14	1	14
Iron ore	7	3	21
Gold ore	4	5	20
diamond	3	10	30
Cobblestone	20	0	0

14 / 16

Mining Statistics Program

Create a new MiningReport.java, compile and run using Java 21.

The Stream pipeline generates four verifiable results

Step	Explanation
1. Clean up the source first	Reject null lists and filter empty records
2. Screen rare	The unit price is at least 5
3. Final stable sorting	Unit price in descending order, then by name

An empty list results in a total of 0, an empty rare list, and the highest value "None". Add records with the same unit price and subtotal, and check whether the name as the second sorting rule is stable.

MiningReport.java · Paragraph 1/2, sequentially connected to the same file

```
import java.util.*;
record MinedBlock(String ore, int amount, int value) {
    MinedBlock {
        if (ore == null || ore.isBlank() || amount < 0 || amount > 1_000_000
            || value < 0 || value > 1_000_000)
            throw new IllegalArgumentException("Check mineral name, quantity and unit price");
        ore = ore.strip();
    }
}
public class MiningReport {
    static boolean isRare(MinedBlock block) {
        return block != null && block.value() >= 5;
    }
    static long totalValue(MinedBlock block) {
        return (long) block.amount() * block.value();
    }
    static List<MinedBlock> clean(List<MinedBlock> blocks) {
        if (blocks == null || blocks.size() > 1_000_000)
            throw new IllegalArgumentException("The record table cannot be null And at most 100 Ten
> thousand");
        return blocks.stream().filter(Objects::nonNull).toList();
    }
    static long totalBlocks(List<MinedBlock> blocks) {
        return clean(blocks).stream().mapToLong(MinedBlock::amount).sum();
    }
    static long valueOfAll(List<MinedBlock> blocks) {
        return clean(blocks).stream().mapToLong(MiningReport::totalValue).sum();
    }
    static List<String> rareLabels(List<MinedBlock> blocks) {
```

MiningReport.java · Paragraph 2/2, sequentially connected to the same file

```
        return clean(blocks).stream().filter(MiningReport::isRare)
            .sorted(Comparator.comparingInt(MinedBlock::value).reversed())
            .thenComparing(MinedBlock::ore))
            .map(block -> block.ore() + "x" + block.amount()).toList();
    }
    static String bestOre(List<MinedBlock> blocks) {
        return clean(blocks).stream()
            .sorted(Comparator.comparingLong(MiningReport::totalValue).reversed())
            .thenComparing(MinedBlock::ore))
            .map(MinedBlock::ore).findFirst().orElse("None");
    }
    public static void main(String[] args) {
        List<MinedBlock> blocks = List.of(
            new MinedBlock("Coal", 14, 1), new MinedBlock("Iron ore", 7, 3),
            new MinedBlock("Gold ore", 4, 5), new MinedBlock("diamond", 3, 10),
            new MinedBlock("Cobblestone", 20, 0));
        System.out.println("Total blocks=" + totalBlocks(blocks));
        System.out.println("Rare=" + rareLabels(blocks));
        System.out.println("Total value=" + valueOfAll(blocks));
        System.out.println("Highest value=" + bestOre(blocks));
    }
}
```

Output

```
Total blocks=48
Rare=[diamondx3, Gold orex4]
Total value=85
Highest value=diamond
```

Here "rare" means unit price ≥ 5 , and "highest value" is compared by the quantity of each record $\times$ unit price, not by unit price, nor by the total amount after combining the same minerals. The maximum number of records is 1 million, the quantity and unit price of each record are at most 1 million, the total value does not exceed 10^{18} , and long can be accommodated. The total number of empty lists is 0, the rare list is empty, and the highest value is "none"; empty records are filtered, and null source lists are rejected.

15 / 16

Expansion tasks and unit review

Lessons 26 to 30 completed key value statistics, collection deduplication, random simulation, exception recovery, file archiving and data pipeline.

The third-level task continues to expand the data pipeline

Element	Explanation
Pipeline explanation	Explain the input and output of each step
Empty result	The total number is 0, the highest is none
Boundary rules	Quantity, unit price and total number of records

Lesson 31 Go through the Paper portal, create a real server plug-in, and have the first welcome message appear when the server loads.

Third gear operation

- Wooden Pickaxe: Filter a list of integers for even numbers and sum them.
- Iron Pickaxe: Sort items by name length, if the length is the same, then sort items in alphabetical order.
- Diamond Pickaxe: Generate top five, average points and underachievement lists from player records.

Acceptance comparison table

Self-evaluation	Can	Still need to practice
I can read Stream pipe	☐	☐
I can choose filter and map	☐	☐
I will handle empty results	☐	☐

16 / 16

Own acceptance record

Beautiful output is the starting point, and being able to explain boundaries is considered mastery. Select one case each from normal, borderline and error, write down your expectations before running, and leave evidence after running.

Leave a basis for judgment

Category	The operation you selected	Check key points
Normal	A common input or a successful action	Is the result consistent line by line?
Border	Range endpoints and adjacent values	Is there one more step or one less thing?
Error	Illegal quantity, missing object or bad data	Whether it is explicitly rejected and whether the status is maintained

- Bug records: triggering conditions, actual phenomena, causes, and minimum repairs.
- Illustrated record: describe the starting point, end point and meaning of each arrow.
- Adaptation record: only change one rule, and modify expectations and tests at the same time.

Technical check: Java SE 21 language specification and standard library. See the language specification for class, interface, type and expression rules; see the corresponding API for the behavior of collections, Random, Files and Stream.

<https://docs.oracle.com/javase/specs/jls/se21/html/index.html>

<https://docs.oracle.com/en/java/javase/21/docs/api/index.html>