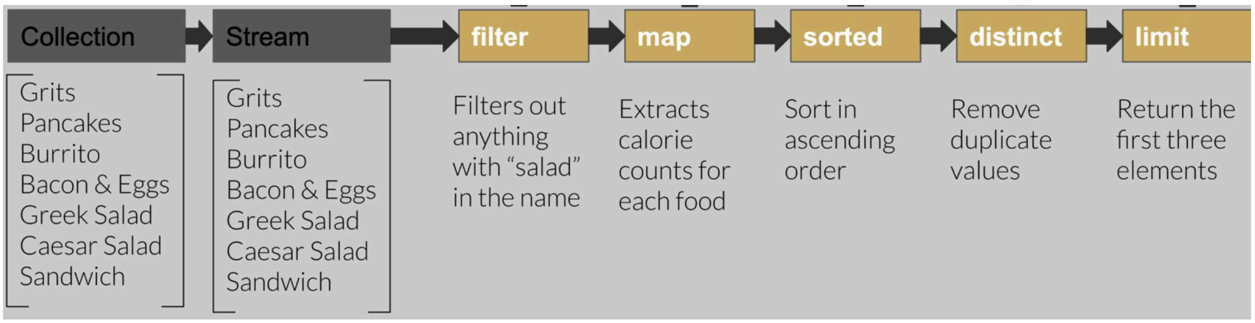
1. The diagram below shows a sequence of stream operations applied to a list of breakfast foods.



```
List<String> foods = List.of(
        "Grits",
        "Pancakes",
        "Burrito",
        "Bacon & Eggs",
        "Greek Salad",
        "Caesar Salad",
        "Sandwich"
);
```

//Write the required code using Stream API to simulate as shown in the above diagram.

---

2. You are given the following map of breakfast foods and their calorie counts:

```
    Map<String, Integer> calorieMap = Map.of(
        "Grits", 150,
        "Pancakes", 350,
        "Burrito", 400,
        "Bacon & Eggs", 300,
        "Greek Salad", 200,
        "Caesar Salad", 250,
        "Sandwich", 330
    );
```

Write a Java Stream expression to print the names and calorie counts of all foods that contain the word "Salad" in their name and have more than 200 calories. The output should be in the format:

Caesar Salad: 250

---

3. Suppose you are designing a payment system for an e-commerce platform. Create a Java interface named PaymentProcessor that meets the following requirements:

    - Contains one abstract method boolean processPayment(double amount) for processing a payment of a given amount.
    - Includes one default method void printReceipt(double amount) that prints a receipt message, calling a private helper method inside the interface to format the message.
    - Has one static method boolean validateCard(String cardNumber) that returns true if the given card number has exactly 16 digits.
    - Uses one private method String formatReceipt(double amount) that returns a formatted string, e.g., "Receipt: Paid $45.50".
    - Write a class CreditCardPayment that implements this interface and always returns true from processPayment.

---

4. You are building a notification service for a healthcare app. Users can receive notifications via SMS, Email, or Push Notification. The system should use a factory pattern for creating the appropriate notifier object.

    Requirements
    Create a Java interface Notifier that includes:

    - One abstract method:
        - void sendNotification(String recipient, String message);
    - One default method:
        - void notifySelf(String message)
            - (Sends a notification to a default recipient, "self@system.com", using the abstract method.)
    - One static method:
        - static boolean isValidMessage(String message)

        - (Returns true if the message is not null and less than 160 characters.)
- One private method:
    - String formatMessage(String recipient, String message)
        - (Returns a string in the format: "To [recipient]: [message]".)
- Implement three classes: SmsNotifier, EmailNotifier, and PushNotifier—each implements Notifier and prints out the formatted message when sending.
- Create a NotifierFactory class with a static method:
    - public static Notifier getNotifier(String type)
        - Returns the correct notifier (SmsNotifier, EmailNotifier, or PushNotifier) based on the type string ("sms", "email", or "push").
- Demonstrate how to send a notification via each method using the factory.

```
public class NotificationDemo {
   public static void main(String[] args) {
       // Using the factory to get each type of Notifier
       Notifier smsNotifier =
NotifierFactory.getNotifier("sms");
       Notifier emailNotifier =
NotifierFactory.getNotifier("email");
       Notifier pushNotifier =
NotifierFactory.getNotifier("push");

       // Send SMS notification
       smsNotifier.sendNotification("+1234567890", "Your
appointment is confirmed.");

       // Send Email notification
       emailNotifier.sendNotification("user@example.com",
"You have a new test result available.");

       // Send Push notification
       pushNotifier.sendNotification("user_device_token",
"Don't forget to take your medicine!");

       // Using default method: notifySelf
       emailNotifier.notifySelf("System maintenance
scheduled for tonight.");
```

```
        // Using default method: sendFormattedNotification
        pushNotifier.sendFormattedNotification("admin",
"Critical update required.");
    }
}
```

---

5. You are given a Map<String, Integer> called stepCounts representing a person's daily step counts for July 2025. The key is the date (in "yyyy-MM-dd" format), and the value is the step count.

   Write Java Stream API code (using lambda expressions where appropriate) to answer the following:

   - Find the date(s) with the maximum and minimum step counts.
     - Print:
       - Date(s) with highest steps (value): date1, date2, …
       - Date(s) with lowest steps (value): date1, date2, ...
   - Calculate and print the total and average number of steps for the month.
     - Print:
       - Total steps: …
       - Average steps per day: ...
   - Find and print all dates where the step count was above the average.
     - Print:
     - Dates above average: date1 (value1), date2 (value2), ...

   Sample Input Data:

```
Map<String, Integer> stepCounts = Map.ofEntries(
        Map.entry("2024-06-01", 9000),
        Map.entry("2024-06-02", 10700),
        Map.entry("2024-06-03", 11500),
        Map.entry("2024-06-04", 9800),
        Map.entry("2024-06-05", 12500),
        Map.entry("2024-06-06", 13400),
        Map.entry("2024-06-07", 8200),
        Map.entry("2024-06-08", 15600),
        Map.entry("2024-06-09", 12900),
        Map.entry("2024-06-10", 9900),
        Map.entry("2024-06-11", 14700),
        Map.entry("2024-06-12", 8800),
        Map.entry("2024-06-13", 11400),
        Map.entry("2024-06-14", 14300),
        Map.entry("2024-06-15", 10200),
        Map.entry("2024-06-16", 15100),
        Map.entry("2024-06-17", 12700),
        Map.entry("2024-06-18", 10800),
        Map.entry("2024-06-19", 9700),
        Map.entry("2024-06-20", 16400),
        Map.entry("2024-06-21", 11900),
        Map.entry("2024-06-22", 10600),
        Map.entry("2024-06-23", 15500),
        Map.entry("2024-06-24", 13700),
        Map.entry("2024-06-25", 12100),
        Map.entry("2024-06-26", 11200),
        Map.entry("2024-06-27", 13200),
        Map.entry("2024-06-28", 8600),
        Map.entry("2024-06-29", 16800),
        Map.entry("2024-06-30", 9900)
);
```

Hint:
To create a stream from Map

```
     stepCounts.entrySet().stream()
```

---

6. You are given a List<Car> where each Car has a String brand and an int price. Use the Stream API and reduce to find the single most expensive car in the list. Suppose the data is as follows:

```
List<Car> cars = Arrays.asList(
        new Car("Skoda", 18544),
        new Car("Volvo", 22344),
        new Car("Fiat", 23650),
        new Car("Renault", 19700),
        new Car("Volvo", 24500),
        new Car("Fiat", 21700),
        new Car("Renault", 19600),
        new Car("Volvo", 18650),
        new Car("Fiat", 19800)
);
```

Sample Output:

    Car{brand='Volvo', price=24500}

---

7. Obtain the last element from a LinkedHashSet using reduce() of Java 8 Stream API.

```
LinkedHashSet<String> linkedSet = new LinkedHashSet<>();
linkedSet.add("Carrot");
linkedSet.add("Broccoli");
linkedSet.add("Spinach");
linkedSet.add("Tomato");
```

---

8. You are given a list of Car objects, where each car has a brand, model, price, and year:

```
List<Car> cars = Arrays.asList(
        new Car("Volvo", "XC40", 25000, 2020),
        new Car("Fiat", "500", 18000, 2022),
        new Car("Volvo", "XC60", 35000, 2022),
        new Car("Skoda", "Octavia", 22000, 2021),
        new Car("Fiat", "Panda", 15000, 2021),
        new Car("Renault", "Clio", 17000, 2020),
        new Car("Volvo", "XC40", 27000, 2022),
        new Car("Skoda", "Fabia", 17500, 2022)
);
```

Sort the list of cars by brand (alphabetically), then by model (alphabetically), then by price (descending), then by year (ascending). Print the sorted list, one car per line.

---

9. You are given a list of Employee objects.

```
List<Employee> employees = Arrays.asList(
   new Employee("Alice", "HR", 28, 53000),
   new Employee("Bob", "Engineering", 35, 76000),
   new Employee("Charlie", "HR", 41, 68000),
   new Employee("David", "Engineering", 29, 82000),
   new Employee("Eva", "Sales", 30, 49000),
   new Employee("Frank", "Engineering", 45, 99000),
   new Employee("Grace", "Sales", 27, 51000),
   new Employee("Heidi", "Engineering", 32, 75000)
);
```

Using Java Stream API, calculate and print the following summary statistics for all employees:

- Total number of employees
- Average salary
- Highest salary
- Lowest salary
- Total salary (sum)
- Average age

---

10. You are given a list of Employee objects, where each employee has:

    String name
    String department
    Double salary (may be null for new hires not yet assigned a salary)
    String managerName (may be null for the top-level manager/CEO)

```
List<Employee> employees = Arrays.asList(
    new Employee("Alice", "HR", 53000.0, "Bob"),
    new Employee("Bob", "HR", 65000.0, null),
    new Employee("Charlie", "Engineering", null, "Diana"),
    new Employee("Diana", "Engineering", 92000.0, null),
    new Employee("Eva", "Sales", 47000.0, "Frank"),
    new Employee("Frank", "Sales", 69000.0, null),
    new Employee("Grace", "Sales", null, "Frank")
);
```

Print all employees, showing "No Manager" if managerName is missing, using Optional.ofNullable.
Print the names of employees whose salary is not yet assigned (salary is null), using Optional.ofNullable.
Find the average salary for each department, ignoring employees whose salary is missing (null).

---

11. You are given the following interface:

```
@FunctionalInterface
interface MessageFormatter {
   String format(String recipient, String message);
}
```

- Anonymous Inner Class:
    - Create an instance of MessageFormatter using an anonymous inner class that returns a string in the format:
        - "To <recipient>: <message>"
    - Store it in a variable and use it to format the message "Don't forget the meeting." for the recipient "Alice".
- Lambda and BiFunction:
    - Java's standard library has the functional interface BiFunction<T, U, R>.
    - Create a BiFunction<String, String, String> instance with the same logic, using a lambda expression.
    - Store it in a variable and trigger it for "Bob" and "Lunch at 1 PM.".
- Call Both:
    - Print the results of both steps.

Sample Output

```
To Alice: Don't forget the meeting.
To Bob: Lunch at 1 PM.
```

---

12. Imagine you're managing an automated chocolate dispenser at a theme park.

For quality control, only one instance of the chocolate dispenser should ever exist at a time—otherwise, chaos (and double dispensing!) ensues.

Every time the dispenser is initialized (constructed), it increments a counter and prints out which thread (worker) set up the machine.
You want to stress-test your system to ensure the dispenser is truly a singleton—even when multiple workers (threads) try to start it at the same time during a busy festival morning.

The Code
1. ChocolateDispenser Singleton with Counter

```
public class ChocolateDispenser {
    private static ChocolateDispenser instance;
    private static int counter = 0;

    private ChocolateDispenser() {
        counter++;
        System.out.println("🍫 ChocolateDispenser initialized by " +
            Thread.currentThread().getName() + " (Setup count: " + counter + ")");
    }

    public static ChocolateDispenser getInstance() {
        if (instance == null) {
            instance = new ChocolateDispenser();
        }
        return instance;
    }

    public static int getSetupCount() {
        return counter;
    }
}
```

2. Festival Worker Stress Test (using ExecutorService)

```
import java.util.Set;
import java.util.concurrent.*;

public class ThemeParkTest {
    public static void main(String[] args) throws InterruptedException {
        Set<ChocolateDispenser> dispensers = ConcurrentHashMap.newKeySet();
        ExecutorService festivalWorkers = Executors.newFixedThreadPool(10);

        Runnable setupTask = () -> {
            ChocolateDispenser dispenser = ChocolateDispenser.getInstance();
            dispensers.add(dispenser);
        };

        // 10 festival workers all try to set up the dispenser at once!
        for (int i = 0; i < 10; i++) {
            festivalWorkers.submit(setupTask);
        }

        festivalWorkers.shutdown();
        festivalWorkers.awaitTermination(5, TimeUnit.SECONDS);

        System.out.println("Unique ChocolateDispenser instances: " +
dispensers.size());
        System.out.println("Number of times dispenser setup happened: " +
ChocolateDispenser.getSetupCount());
        System.out.println("Did chaos happen (singleton broken)? " +
                (dispensers.size() > 1 || ChocolateDispenser.getSetupCount() > 1 ?
"Yes!" : "No. System safe."));
    }
}
```

Expected Output (if not thread-safe):

```
ChocolateDispenser initialized by pool-1-thread-3 (Setup count: 1)
ChocolateDispenser initialized by pool-1-thread-7 (Setup count: 2)
ChocolateDispenser initialized by pool-1-thread-1 (Setup count: 3)
Unique ChocolateDispenser instances: 3
Number of times dispenser setup happened: 3
Did chaos happen (singleton broken)? Yes!
```

Why did chaos (multiple setups) occur?
How can you protect the chocolate (make the singleton thread-safe)?

What could happen in a real-world system (data corruption, wasted resources, etc.) if you don't enforce the singleton?

---

13. Your company is developing two different Java applications for its operations:

    Scenario 1: Real-Time Chat Server
    The chat server receives thousands of short messages per minute from users. Each message needs to be processed and broadcast to the intended recipients as quickly as possible. Most message-processing tasks are very brief (less than 100ms). During traffic spikes (e.g., lunchtime), the number of simultaneous messages can suddenly jump.

    Scenario 2: Image Processing Service
    The image service processes uploaded images (e.g., resizing, filtering) for a social media app. Each image-processing task can take several seconds or more. The number of image uploads at a time is typically limited and predictable. Your server has 8 CPU cores and you want to avoid overloading it.

    For each scenario, which thread pool implementation (Executors.newCachedThreadPool() or Executors.newFixedThreadPool(int n)) would you choose?

    Clearly state your choice for each application.
    Justify your answer with respect to performance, resource usage, and system stability.

---

14. Your team is designing a backend for a ride-sharing app. Two services require parallel processing:

    Service A: Driver Location Updates
    Thousands of drivers report their GPS location every few seconds. Each update is a tiny, fast operation (takes a few milliseconds to process). The number of incoming updates fluctuates a lot, especially during rush hour or special events.

Service B: Ride Fare Calculation
Each time a ride ends, a fare calculation job is triggered. Fare calculation involves fetching user data, promotions, map routes, and can take 2–4 seconds per ride. Your server has 16 CPU cores and is also running other backend services. You want to avoid excessive CPU contention and keep the backend responsive.

For each service above, would you choose a cached thread pool or a fixed thread pool?

Clearly state your choice for each service.
Explain your reasoning, focusing on resource management, latency, and scalability.

---

15. .
You are developing a banking application. At the end of the day, your system needs to process hundreds of deposits made by different customers to the same bank account.

- You create a shared BankAccount object with a balance. Each deposit is performed in its own thread (using executor.execute(Runnable)).
- You need to ensure that the final balance is correct after all deposits—i.e., all deposits are applied safely, with no lost updates due to concurrency.

- Implement a BankAccount class with methods to deposit money and get the balance.
- Launch N deposit tasks in parallel using executor.execute(), each depositing $100.
- Print the final balance after all deposits have completed.
- Ensure thread-safety so the final balance is correct (N * $100 + initial balance).

```
public class BankAccount {
   private int balance = 0;
```

```
    public void deposit(int amount) {
        balance += amount;
        System.out.println("Deposited $" + amount + " by " +
Thread.currentThread().getName() +
                           ", New Balance: " + balance);
    }

    public int getBalance() {
        return balance;
    }
}
```

---

16. You are building a warehouse management system that can store and retrieve items of different types (e.g., Book, Laptop, Furniture).

    You want to design a generic Warehouse<T> class that supports:

    - Adding items to the warehouse.
    - Retrieving the last item added (like a stack).
    - Getting the number of items in the warehouse.
    - Implement the generic class Warehouse<T>.
    - It should provide methods: addItem(T item), T retrieveLastItem(), and int getItemCount().
    - Create two types of items, Book and Laptop, with at least two fields each.
    - Create a Warehouse<Book>, add a few books, and retrieve the last one.
    - Create a Warehouse<Laptop>, add a few laptops, and retrieve the last one.
    - Why is it important that your Warehouse<T> is generic instead of using Object?
    - What compile-time advantages does Generics give you in this scenario?

17. What code change is needed to make the method compile?

```
public static T identity(T t) {
        return t;
```

}
A. Add <T> after the public keyword.
B. Add <T> after the static keyword.
C. Add <T> after T.
D. Add <?> after the public keyword.
E. Add <?> after the static keyword.
F. No change required. The code already compiles.

---

18. Which of the following statements are true? (Choose all that apply.)
    A. Comparable is in the java.util package.
    B. Comparator is in the java.util package.
    C. compare() is in the Comparable interface.
    D. compare() is in the Comparator interface.
    E. compare() takes one method parameter.
    F. compare() takes two method parameters.

---

19. Which of the following compiles and print outs the entire set? (Choose all that apply.)
    Set<String> s = new HashSet<>();
    s.add("lion");
    s.add("tiger");
    s.add("bear");
    s.forEach(____________________);

    A. () -> System.out.println(s)
    B. s -> System.out.println(s)
    C. (s) -> System.out.println(s)
    D. System.out.println(s)
    E. System::out::println
    F. System.out::println

---

20. You are developing a Word Chain Game app. In each round, players take turns submitting words. At the end of the round, you want to highlight the longest word submitted as the "Word Chain Champion."

    You use this BiFunction:

```
BiFunction<String, String, String> mapper = (v1, v2) -> v1.length() > v2.length() ?
v1 : v2;
```

Explain how this BiFunction helps to pick the longest word from the list of words submitted in a round.

Given this list of words:

```
List<String> words = Arrays.asList(
    "apple",
    "rhinoceros",
    "cat",
    "strawberry",
    "dog"
);
```

Write Java code using streams and mapper to find the longest word.

Suppose you want to change the rule for the next round: now, the "champion" word is the one with the highest number of vowels (a, e, i, o, u). How would you write a new BiFunction and use it with streams to pick the winner from the same list?

If two words have the same number of vowels, how would you break the tie by picking the word that comes last alphabetically?

---

21. You are developing a utility method for an e-commerce site that needs to help customers quickly find an affordable, in-stock product from a given list of products.

```
public class Product {
    private final String name;
    private final boolean inStock;
    private final double price;
    private final double rating;

    public Product(String name, boolean inStock, double price, double rating) {
        this.name = name;
        this.inStock = inStock;
        this.price = price;
        this.rating = rating;
    }

    public String getName() { return name; }
    public boolean isInStock() { return inStock; }
    public double getPrice() { return price; }
    public double getRating() { return rating; }
}
```

Write a method called findFirstAffordableInStock that accepts a List<Product> and returns an Optional<Product> representing the first product (in the original list order) that is in stock and costs less than $500. Use a sequential stream and the findFirst() method.

Method signature:
public Optional<Product> findFirstAffordableInStock(List<Product> products)

Write unit tests for your method using JUnit framework:

- Test with a list containing several products:
  - Some in stock, some not; some under $500, some above.
  - Verify that the returned product is the first in the list matching the criteria.
- Test with a list where no products match:
  - All are either out of stock or too expensive.
  - Verify that the result is empty.
- Test with an empty list:
  - The result should also be empty.
- Test the case where multiple products match—ensure only the first one is returned.

22. .

Your application tracks employee records for a company.

```
import java.time.LocalDate;

import java.time.LocalDate;

public class Employee {
    private final int employeeId;
    private final String name;
    private final LocalDate dateOfJoining;

    public Employee(int employeeId, String name, LocalDate dateOfJoining) {
        this.employeeId = employeeId;
        this.name = name;
        this.dateOfJoining = dateOfJoining;
    }

    public int getEmployeeId() { return employeeId; }
    public String getName() { return name; }
    public LocalDate getDateOfJoining() { return dateOfJoining; }
}
```

Write a method called getEmployeesSortedByJoiningDate that returns a list sorted:

- By dateOfJoining (earliest first)
- Then by name (A–Z)
- If two employees have the same name and joining date, sort by a unique employee ID (ascending).

Use the Java Stream API for sorting.

Method Signature:
public List<Employee> getEmployeesSortedByJoiningDate(List<Employee> employees)

- Test different joining dates (should sort by date).
- Test same date, different names (should sort by name).
- Test same date and name, different IDs (should sort by employeeId).
- Test with an empty list.