

# Stream e Files

---

## 1. Creazione del Progetto

---

### 1.1. Avviare IntelliJ IDEA e Creare un Nuovo Progetto

- Apri IntelliJ IDEA.
- Seleziona **File** → **New** → **Project**.
- Nella finestra di dialogo:
  - Seleziona **Java** come tipo di progetto.
  - Seleziona **Maven** come build system.
- Premi **Create**.

### 1.2. Struttura del Progetto

La struttura finale sarà simile a:

```
it.epicode/  
├── src/  
│   ├── main/  
│   │   ├── java/  
│   │   │   ├── it/  
│   │   │   │   ├── epicode/  
│   │   │   │   │   ├── Main.java  
│   │   │   │   │   └── model/  
│   │   │   │   │       ├── Product.java  
│   │   │   │   │       └── Customer.java
```

```
|
|
|   └─ Order.java
|   └─ resources/
|   └─ test/
|   └─ java/
└─ pom.xml
```

## 2. Aggiunta delle Dipendenze Maven

---

Apri il file **pom.xml** e aggiungi (se non già presente) il tag `<dependencies>` con la dipendenza per Apache Commons IO:

```
<dependencies>
  <!-- Dipendenza per Apache Commons IO -->
  <dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.11.0</version>
  </dependency>
  <!-- Altre dipendenze se necessarie -->
</dependencies>
```

Salva il file per scaricare le dipendenze.

## 3. Creazione delle Classi del Progetto

---

### 3.1. Classe Product

```
package it.epicode.model;

public class Product {
    // Variabili d'istanza: private per garantire l'incapsulamento
    private String name;        // Nome del prodotto
    private String category;    // Categoria del prodotto
    private double price;       // Prezzo del prodotto

    // Costruttore della classe Product
    public Product(String name, String category, double price) {
        this.name = name;
        this.category = category;
        this.price = price;
    }

    // Getter e Setter per ogni attributo
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getCategory() {
        return category;
    }
    public void setCategory(String category) {
        this.category = category;
    }
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        this.price = price;
    }
}
```

```
}  
}
```

## 3.2. Classe Customer

```
package it.epicode.model;  
  
public class Customer {  
    // Attributi privati per l'incapsulamento dei dati  
    private int id;           // Identificativo univoco del cliente  
    private String name;      // Nome del cliente  
    private String email;     // Email del cliente  
  
    // Costruttore con parametri  
    public Customer(int id, String name, String email) {  
        this.id = id;  
        this.name = name;  
        this.email = email;  
    }  
  
    // Getter e Setter  
    public int getId() {  
        return id;  
    }  
    public void setId(int id) {  
        this.id = id;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public String getEmail() {
```

```

        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }

    // Metodo toString per una rappresentazione leggibile
    @Override
    public String toString() {
        return "Customer{" +
            "id=" + id +
            ", name='" + name + '\'' +
            ", email='" + email + '\'' +
            '}';
    }
}

```

### 3.3. Classe Order

```

package it.epicode.model;

public class Order {
    // Attributi privati per garantire l'incapsulamento
    private int orderId;          // Identificativo dell'ordine
    private Customer customer;    // Cliente che ha effettuato l'ordine
    private Product product;     // Prodotto ordinato
    private int quantity;        // Quantità ordinata
    private double amount;       // Importo totale (calcolato come prodotto * quantità)

    // Costruttore della classe Order
    public Order(int orderId, Customer customer, Product product, int quantity, double amount) {
        this.orderId = orderId;
        this.customer = customer;
        this.product = product;
    }
}

```

```
        this.quantity = quantity;
        this.amount = amount;
    }

    // Getter e Setter
    public int getOrderId() {
        return orderId;
    }
    public void setOrderId(int orderId) {
        this.orderId = orderId;
    }
    public Customer getCustomer() {
        return customer;
    }
    public void setCustomer(Customer customer) {
        this.customer = customer;
    }
    public Product getProduct() {
        return product;
    }
    public void setProduct(Product product) {
        this.product = product;
    }
    public int getQuantity() {
        return quantity;
    }
    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
    public double getAmount() {
        return amount;
    }
    public void setAmount(double amount) {
        this.amount = amount;
    }
}
```

```
// Metodo toString per visualizzare i dettagli dell'ordine
@Override
public String toString() {
    return "Order{" +
        "orderId=" + orderId +
        ", customer=" + customer +
        ", product=" + product +
        ", quantity=" + quantity +
        ", amount=" + amount +
        '}';
}
```

## 4. Salvataggio su File e Spiegazione di StringBuilder

---

### 4.1. Formato di Salvataggio File

Per l'esercizio, il formato scelto è:

`nomeProdotto1@categoriaProdotto1@prezzoProdotto1#nomeProdotto2@categoriaProdotto2@prezzoProdotto2`

**Motivazione:**

- Il simbolo `@` separa i campi di un record (nome, categoria, prezzo) rendendo semplice lo **split** della stringa.
- Il simbolo `#` separa i diversi record, facilitando il parsing in una lista di prodotti.

### 4.2. Cos'è uno StringBuilder e Perché Si Usa

Lo **StringBuilder** è una classe Java utilizzata per costruire e modificare stringhe in maniera efficiente.

**Perché usarlo?**

- È più performante rispetto alla concatenazione diretta con l'operatore `+`, soprattutto in cicli, poiché riduce la creazione di oggetti intermedi.
- Permette di appendere, modificare o cancellare parti della stringa senza creare nuove istanze.

## 5. Classe Main con Esempi sugli Stream e Controllo Scanner

---

La seguente classe Main include:

- Creazione di istanze di **Customer**, **Product** e **Order**.
- Esempi di stream con operazioni di raggruppamento e statistiche.
- Un ciclo interattivo con controllo errori per leggere l'input tramite Scanner.

```
package it.epicode;

import it.epicode.model.Customer;
import it.epicode.model.Order;
import it.epicode.model.Product;
import java.util.*;
import java.util.stream.Collectors;

public class Main {

    public static void main(String[] args) {
        // Creazione di alcune istanze di esempio
        Customer customer1 = new Customer(1, "Mario Rossi", "mario.rossi@example.com");
        Customer customer2 = new Customer(2, "Luigi Bianchi", "luigi.bianchi@example.com");

        Product product1 = new Product("Prodotto A", "Elettronica", 99.99);
        Product product2 = new Product("Prodotto B", "Casa", 49.99);
```



```

Product product3 = new Product("Prodotto C", "Elettronica", 149.99);

Order order1 = new Order(101, customer1, product1, 2, 2 * product1.getPrice());
Order order2 = new Order(102, customer1, product2, 1, product2.getPrice());
Order order3 = new Order(103, customer2, product3, 3, 3 * product3.getPrice());
Order order4 = new Order(104, customer2, product1, 1, product1.getPrice());

List<Product> products = new ArrayList<>(Arrays.asList(product1, product2, product3));
List<Order> orders = new ArrayList<>(Arrays.asList(order1, order2, order3, order4));

// Esempi interattivi con Scanner e controllo errori
Scanner scanner = new Scanner(System.in);
boolean exit = false;
while (!exit) {
    System.out.println("\n--- Menu Operazioni Stream ---");
    System.out.println("1. Raggruppa ordini per cliente");
    System.out.println("2. Calcola totale vendite per cliente");
    System.out.println("3. Trova il prodotto più costoso");
    System.out.println("4. Calcola media importi degli ordini");
    System.out.println("5. Raggruppa prodotti per categoria e somma prezzi");
    System.out.println("6. Esempi aggiuntivi di operazioni stream");
    System.out.println("7. Esci");
    System.out.print("Scelta: ");
    String scelta = scanner.nextLine();

    switch (scelta) {
        case "1":
            // Raggruppa gli ordini per cliente
            Map<Customer, List<Order>> ordersByCustomer = orders.stream()
                .collect(Collectors.groupingBy(Order::getCustomer));
            ordersByCustomer.forEach((cust, ordList) ->
                System.out.println(cust.getName() + " → " + ordList));
            break;
        case "2":
            // Calcola il totale delle vendite per cliente
            Map<Customer, Double> totalSalesByCustomer = orders.stream()

```

```

        .collect(Collectors.groupingBy(Order::getCustomer,
            Collectors.summingDouble(Order::getAmount)));
totalSalesByCustomer.forEach((cust, total) ->
    System.out.println(cust.getName() + " → " + total));
break;
case "3":
    // Trova il prodotto più costoso
    Optional<Product> mostExpensiveProduct = products.stream()
        .max(Comparator.comparingDouble(Product::getPrice));
    System.out.println("Prodotto più costoso: " +
        mostExpensiveProduct.map(Product::getName).orElse("Nessuno"));
    break;
case "4":
    // Calcola la media degli importi degli ordini
    Double averageOrderAmount = orders.stream()
        .collect(Collectors.averagingDouble(Order::getAmount));
    System.out.println("Media importi ordini: " + averageOrderAmount);
    break;
case "5":
    // Raggruppa i prodotti per categoria e somma i prezzi
    Map<String, Double> totalByCategory = products.stream()
        .collect(Collectors.groupingBy(Product::getCategory,
            Collectors.summingDouble(Product::getPrice)));
    totalByCategory.forEach((cat, total) ->
        System.out.println(cat + " → " + total));
    break;
case "6":
    // Esempi aggiuntivi sugli stream:
    // a) Raggruppa per cliente e conta il numero di ordini
    Map<Customer, Long> ordersCountByCustomer = orders.stream()
        .collect(Collectors.groupingBy(Order::getCustomer, Collectors.counting()));
    ordersCountByCustomer.forEach((cust, count) ->
        System.out.println(cust.getName() + " ha effettuato " + count + " ordini."));

    // b) Calcola il totale importi per prodotto
    Map<String, Double> totalSalesByProduct = orders.stream()

```

```

        .collect(Collectors.groupingBy(o -> o.getProduct().getName(),
            Collectors.summingDouble(Order::getAmount)));
totalSalesByProduct.forEach((prod, total) ->
    System.out.println(prod + " → " + total));

// c) Filtra ordini sopra un certo importo e ordina per quantità
List<Order> filteredOrders = orders.stream()
    .filter(o -> o.getAmount() > 100)
    .sorted(Comparator.comparingInt(Order::getQuantity))
    .collect(Collectors.toList());
System.out.println("Ordini con importo > 100: " + filteredOrders);

// Spiegazione: l'operatore "::" è una _method reference_,
// es. Order::getCustomer equivale a lambda o -> o.getCustomer()
break;
case "7":
    exit = true;
    break;
default:
    System.out.println("Scelta non valida, riprova.");
}
}
scanner.close();
}
}

```

### Nota sugli Stream e Method References:

- **Order::getCustomer** è una *method reference* che richiama il metodo `getCustomer()` per ogni oggetto `Order`.
- Questo approccio riduce il codice e rende le espressioni lambda più concise rispetto a:  
`order -> order.getCustomer()`.

## 6. Salvataggio e Lettura dei Prodotti su Disco con Apache Commons IO

---

### 6.1. Metodo per Salvare i Prodotti

```
import org.apache.commons.io.FileUtils;
import java.io.File;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.util.List;

public void salvaProdottiSuDisco(List<Product> products, String filePath) throws IOException {
    // Utilizzo dello StringBuilder per costruire la stringa finale in maniera efficiente
    StringBuilder sb = new StringBuilder();
    for (Product p : products) {
        // Formato: nome@categoria@prezzo#
        sb.append(p.getName()).append("@")
          .append(p.getCategory()).append("@")
          .append(p.getPrice()).append("#");
    }
    // Rimuove l'ultimo '#' se presente
    if (sb.length() > 0) {
        sb.setLength(sb.length() - 1);
    }
    FileUtils.writeStringToFile(new File(filePath), sb.toString(), StandardCharsets.UTF_8);
}
```

### 6.2. Metodo per Leggere i Prodotti da Disco

```
import org.apache.commons.io.FileUtils;
import java.io.File;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
```

```

import java.util.ArrayList;
import java.util.List;

public List<Product> leggiProdottiDaDisco(String filePath) throws IOException {
    String content = FileUtils.readFileToString(new File(filePath), StandardCharsets.UTF_8);
    List<Product> productList = new ArrayList<>();
    String[] records = content.split("#");
    for (String record : records) {
        String[] parts = record.split("@");
        if (parts.length == 3) {
            String name = parts[0];
            String category = parts[1];
            double price = Double.parseDouble(parts[2]);
            productList.add(new Product(name, category, price));
        }
    }
    return productList;
}

```

## 8. Tabella Riassuntiva dei Metodi di Collectors

Metodo Collector	Descrizione
<code>toList()</code>	Colleziona gli elementi in una <b>List</b> .
<code>toSet()</code>	Colleziona gli elementi in un <b>Set</b> (elementi unici).
<code>toMap(keyMapper, valueMapper)</code>	Colleziona gli elementi in una <b>Map</b> usando le funzioni per mappare chiave e valore.
<code>groupingBy(classifier)</code>	Raggruppa gli elementi in una <b>Map</b> dove la chiave è il risultato della funzione <i>classifier</i> .

Metodo Collector	Descrizione
<code>groupBy(classifier, downstream)</code>	Raggruppa gli elementi e applica un operatore <i>downstream</i> (es. summing, counting).
<code>partitioningBy(predicate)</code>	Partiziona gli elementi in due gruppi (true/false) in una <b>Map&lt;Boolean, List&gt;</b> .
<code>counting()</code>	Conta il numero di elementi presenti nello stream.
<code>summingInt()/summingDouble()/summingLong()</code>	Somma i valori numerici estratti da ogni elemento.
<code>averagingInt()/averagingDouble()/averagingLong()</code>	Calcola la media dei valori numerici estratti da ogni elemento.
<code>joining()</code>	Concatena gli elementi (solitamente stringhe) in un'unica stringa.

**Spiegazione degli Utilizzi:**

- *groupBy*: Organizza gli elementi in base a una chiave (es. raggruppare ordini per cliente).
- *counting*: Utile per contare quanti elementi sono presenti in ogni gruppo.
- *summing* e *averaging*: Permettono di aggregare valori numerici per ottenere totali o medie.
- *joining*: Consente di concatenare stringhe (es. per generare un output formattato).