→ class Product {

String sku, name;

double price, discount;

int stock;

👉 This defines a **Product** blueprint — each product has:

→ Product(String sku, String name, double price, double discount, int stock) {

this.sku = sku;

this.name = name;

this.price = price;

this.discount = discount;

this.stock = stock;

}

👉 A **constructor** initializes each product when created.

---

public String toString() {

return sku + " | " + name + " | Price: " + price + " | Disc: " + discount +

"% | Stock: " + stock;

}

👉 The toString() method gives a **readable format** when displaying product info in the text area.

Object already has a method named toString()

Part	Meaning
public	This method can be accessed from anywhere.
String	**Return type** → the method returns text (a String object).
toString()	The **method name**, defined originally in Java's built-in Object class.

⚙️ So in short:

- String → the **type of data returned** (text).
- toString() → built-in method from the Object class.
- @Override → means we are replacing the parent version with our own.
- It makes your object readable and meaningful when printed or displayed.

🧩 SECTION 2: MyArray — Custom Dynamic Array (1st Data Structure)

```
class MyArray {

    private Product[] arr = new Product[10];

    private int size = 0;
```

Why private?

- Makes arr accessible **only inside MyArray**.
- Prevents outside classes from changing it directly (encapsulation).

MyArray is a **custom container** for storing multiple Product objects.
MyArray acts like a **dynamic array**, which grows automatically when it's full.

MyArray (Custom Dynamic Array) Summary

- **Purpose:** Store Product objects dynamically without using ArrayList.
- **Fields:**
  - arr → internal array (initial capacity 10).

- size → tracks how many products are actually stored.

- **Methods:**
  1. add(Product p) → Adds a product.
     - If array is full, creates a **bigger array**, copies old products, then adds the new product.
  2. get(int i) → Returns the product at index i.
  3. remove(int index) → Removes product at given index, shifts remaining products left.
  4. size() → Returns number of products stored.
- **How it works:**
  - Think of arr as a **shelf with slots**.
  - size = how many slots are currently filled.
  - Adding fills the next empty slot; removing shifts products left.
  - When shelf is full, it **grows automatically** to hold more products.

```
void add(Product p) {
    if (size == arr.length) {
        Product[] newArr = new Product[arr.length * 2];
        System.arraycopy(arr, 0, newArr, 0, arr.length);
        arr = newArr;
    }
    arr[size++] = p;
}
```

🧠 What happens:

1. Adds Soap in slot 0.
2. Adds Shampoo in slot 1.
3. If we keep adding beyond 10 products,
   → it **creates a new array double in size (20)**,
   → **copies** all old products into it,
   → then adds the new product.

So it's like automatically adding more shelves when the old one is full.

```
Product get(int i) {
    return arr[i];
}
```

Key Point: `get(i)` returns the whole Product object at index i, so you can read its fields and methods, but you **cannot change the internal array itself**.

```
void remove(int index) {
    for (int i = index; i < size - 1; i++) arr[i] = arr[i + 1];
    size--;
}
```

- Shifts all products one step left from given index → removes it logically.
- remove() is used **only when a product's stock becomes 0** — meaning the **entire product is sold out**, not just one quantity.

🔢 Size

```
int size() { return size; }
```

- Returns current number of stored products.

🔄 Their Relationship in the App

Action	MyArray	MyQueue
Add new product	Adds product to inventory	—
Buy product	Finds and updates product stock	Enqueues sold product
Out of stock	Removes product from inventory	Product stays in sales queue
View sales	—	Queue shows recent sales

- Add products → goes into **MyArray** (inventory list).
- Sell products → they go through **MyQueue** (billing system), and dequeue() removes them once sold.

🧠 Concept Summary — Inventory & Sales System

MyArray.add(Product p) → 🏪 *Inventory Management*

- Adds a new product to the shop's inventory list.
- Used to **store and display** all products (SKU, name, price, stock, etc.).
- Think of it like **placing products on shelves** — they stay there until removed.
- 💡 Example: Adding "Apple" shows it in the inventory list.

MyQueue.enqueue(Product p) → 💳 *Sales Queue*

- When a product is sold, it's **added to the queue** for processing.
- Works on **FIFO (First In, First Out)** — first sold, first processed.
- Think of it like **customers lining up at checkout**.
- 💡 Example: "Apple" is enqueued first → will be billed before "Banana".
- enqueue() adds sold products to a billing queue, managing the sales order. It ensures First-In-First-Out processing.

MyQueue.dequeue() → 🧾 *Billing / Checkout*

- Removes the **first product** from the queue after sale completion.
- Represents the **checkout process** — item sold, removed from line.
- 💡 Example: "Apple" is dequeued → sale completed, next product moves up.

```
void enqueue(Product p) {
    if (size == arr.length) {
        Product[] newArr = new Product[arr.length * 2];
        for (int i = 0; i < size; i++)
            newArr[i] = arr[(front + i) % arr.length];
        arr = newArr;
        front = 0;
        rear = size;
    }
    arr[rear] = p;
    rear = (rear + 1) % arr.length;
    size++;
}
```

1. **Check if full:**
   if (size == arr.length) → if queue is full, double its size.
   Because arrays in Java can't expand on their own.
2. **Create bigger array:**
   newArr → new array with double capacity.
3. **Copy old elements:**
   newArr[i] = arr[(front + i) % arr.length];
   Copies all products in the correct order (handles wrapping).
4. **Reset positions:**
   After copying, set front = 0 and rear = size.
5. **Add the new product:**
   arr[rear] = p; → places product at the rear end.
6. **Move rear forward:**
   (rear + 1) % arr.length → moves the rear pointer ahead (circularly).
7. **Increase size:**
   size++ → total products increased by 1.

```
Product dequeue() {
    if (size == 0) return null;
    Product p = arr[front];
    front = (front + 1) % arr.length;
    size--;
    return p;
}
```

Check if empty:
If no products (size == 0), return null.

Take out first product:
p = arr[front] → take the product at the front.

Move front ahead:
(front + 1) % arr.length → moves to next product (circular).

Decrease size:
One product removed → size--.

Return removed product.

```
boolean isEmpty() { return size == 0; }
```

Simple check — returns true if queue has no products.

📝 Why Array and Queue are Used

Array (Inventory):

- Stores all products in order.
- Easy to display list in UI(easy to bind with UI (e.g., JTextArea display).
- Fast access by index.
- Simple and memory-efficient.
  👉 Used for showing and updating product list.

Queue (Billing):

- Follows **FIFO** (First In, First Out).
- Works like real billing — first customer billed first.
- Easy to track recent sales.
  👉 Used to manage and record sales in order.

Not HashMap / LinkedList because:

- 🧠 If I used a **LinkedList**:
- Accessing the nth product would be **slow (O(n))**, not suitable for frequent lookups.
- Overhead of node objects adds unnecessary memory use.
- 🧠 If I used a **HashMap**:
- It would store key-value pairs, but my data is **not key-based lookup** heavy — it's sequential (display all products, not find by ID).
- HashMap doesn't maintain order, so displaying in order becomes harder.

---

static makes variables and methods shared by the entire program, so we can access them directly without creating objects. It's used here because the shop has one shared inventory, one billing record, and one output area."

```
public class Shopbill { // class name matches file name

    static MyArray products = new MyArray();

    static MyQueue recentSales = new MyQueue(); // second DS

    static JTextArea output;

    static String billHistory = "";

    static double totalSales = 0.0;

    static final double TAX_RATE = 0.05; // 5% tax
```

public class Shopbill

- This defines the **main class** of your program..
- It’s like the **controller** — it manages the entire billing and inventory system.
- Inside it, you’ll find everything: data structures, UI setup, and logic.
- **Variables** → store data (inventory, sales, totals).
- **Methods** → perform actions (run program, show products).

BorderLayout divides the window into parts,=(Defines how components (buttons, text areas, panels) will be arranged inside the window.)

JTextArea displays all messages, and JScrollPane lets it scroll — all placed in the center of the window.

```
public static void main(String[] args) {

    JFrame f = new JFrame("🛒 Billing and Inventory Management System");

    f.setSize(800, 600);

    f.setLayout(new BorderLayout());


    // Output area

    output = new JTextArea();

    output.setEditable(false);

    output.setFont(new Font("Monospaced", Font.PLAIN, 14));

    f.add(new JScrollPane(output), BorderLayout.CENTER);
```

JFrame f = new JFrame("Title") → Creates main app window (object in heap).
f.setSize(800,600) → Sets window size.
f.setLayout(new BorderLayout()) → Divides window into 5 parts (N, S, E, W, Center).
output = new JTextArea() → Multi-line display box for bills/inventory.
output.setEditable(false) → User can’t type, only program displays data.
output.setFont(...) → Makes text neat and readable.
f.add(new JScrollPane(output), BorderLayout.CENTER) → Adds scrollable text area to center of window.

```
JPanel addPanel = new JPanel(new GridLayout(6, 2));

    addPanel.setBorder(BorderFactory.createTitledBorder("Add Product"));
```

```
JTextField sku = new JTextField();

JTextField name = new JTextField();

JTextField price = new JTextField();

JTextField disc = new JTextField();

JTextField stock = new JTextField();


addPanel.add(new JLabel("SKU:"));

addPanel.add(sku);

addPanel.add(new JLabel("Name:"));

addPanel.add(name);

addPanel.add(new JLabel("Price:"));

addPanel.add(price);

addPanel.add(new JLabel("Discount %:"));

addPanel.add(disc);

addPanel.add(new JLabel("Stock:"));

addPanel.add(stock);


JButton addBtn = new JButton("Add Product");

addPanel.add(addBtn);
```

JPanel (addPanel, buyPanel, south)

- **Purpose:** Container to hold related components (labels, text fields, buttons).
- **Example:** JPanel addPanel = new JPanel(new GridLayout(6,2));
- **Layout:** GridLayout(rows, columns) arranges components in a table-like structure.

JLabel

- **Purpose:** Display text labels (static, not editable).
- **Example:** new JLabel("SKU:") → shows “SKU:” next to input box.
- **Used For:** Describing each input field.

JTextField

- **Purpose:** Input boxes where user types information (SKU, Name, Price, etc.).

- **Example:** JTextField sku = new JTextField();

JButton

- **Purpose:** Clickable button to perform actions.
- **Example:** new JButton("Add Product") → triggers ActionListener.
- **Used For:** Add product, Buy product, Show bills, Export bills.

BorderFactory

- **Purpose:** Adds a border with a title around a panel.
- **Example:** addPanel.setBorder(BorderFactory.createTitledBorder("Add Product"));
- **Benefit:** Makes the GUI organized and visually clear.

GridLayout

- **Purpose:** Organizes components in rows and columns.
- **Example:** new GridLayout(6,2) → 6 rows, 2 columns.
- **Benefit:** Ensures labels and text fields line up neatly.

Each component (JLabel, JTextField, JButton) is an **object stored in heap memory**.

Heap memory is the part of your computer's memory (RAM) where **objects are stored at runtime** in Java.

JPanel holds references to these objects; everything is visible inside the frame (JFrame).

```
JPanel buyPanel = new JPanel(new GridLayout(7, 2));
    buyPanel.setBorder(BorderFactory.createTitledBorder("Buy Product"));

    JTextField buySku = new JTextField();
    JTextField qty = new JTextField();
    JTextField phone = new JTextField();
    JCheckBox isMember = new JCheckBox("Is Member?");
    JButton buyBtn = new JButton("Buy");
    JButton historyBtn = new JButton("Show All Bills");
    JButton exportBtn = new JButton("Export Bills");
```

```
buyPanel.add(new JLabel("Enter SKU:"));
buyPanel.add(buySku);
buyPanel.add(new JLabel("Quantity:"));
buyPanel.add(qty);
buyPanel.add(new JLabel("Phone (if not member):"));
buyPanel.add(phone);
buyPanel.add(isMember);
buyPanel.add(buyBtn);
buyPanel.add(historyBtn);
buyPanel.add(exportBtn);

JPanel south = new JPanel(new GridLayout(1, 2));
south.add(addPanel);
south.add(buyPanel);

f.add(south, BorderLayout.SOUTH);
```

🛒 Buy Panel – Quick Notes

- **Panel & Layout:** JPanel buyPanel = new JPanel(new GridLayout(7,2)) → 7 rows × 2 columns; titled border “Buy Product”.
- **Inputs:** SKU, Quantity, Phone (if non-member) → JTextFields.
- **Options & Buttons:** Member checkbox (JCheckBox), Buy (JButton), Show All Bills, Export Bills.
- **Adding Components:** Each field/label/button added in order; GridLayout places them row by row.
- **Combining Panels:** JPanel south = new JPanel(new GridLayout(1,2)) → side-by-side: Add Panel (left) + Buy Panel (right).
- **Frame Placement:** f.add(south, BorderLayout.SOUTH) → bottom section of JFrame.
- **Purpose:** Handles purchase input, updates inventory (MyArray) & sales (MyQueue), shows bills, and manages history/export.

✅ **Memory:** All components are objects in heap; JPanel holds references; JFrame displays visually.

// Add Product Action

```
addBtn.addActionListener(e -> {
        try {
            String newSku = sku.getText();
            String newName = name.getText();
            double newPrice = Double.parseDouble(price.getText());
            double newDisc = Double.parseDouble(disc.getText());
            int newStock = Integer.parseInt(stock.getText());

            // Check for existing SKU
            for (int i = 0; i < products.size(); i++) {
                Product existing = products.get(i);
                if (existing.sku.equalsIgnoreCase(newSku)) {
                    if (existing.price != newPrice || existing.discount != newDisc) {
                        JOptionPane.showMessageDialog(f,
                                "SKU already exists with different price/discount!");
                    } else {
                        JOptionPane.showMessageDialog(f,
                                "Product already exists in the store!");
                    }
                    sku.setText("");
                    name.setText("");
                    price.setText("");
                    disc.setText("");
                    stock.setText("");
                    return;
                }
```

```
            }

            Product p = new Product(newSku, newName, newPrice, newDisc, newStock);
            products.add(p);
            showProducts();

            sku.setText("");
            name.setText("");
            price.setText("");
            disc.setText("");
            stock.setText("");

        } catch (Exception ex) {
            JOptionPane.showMessageDialog(f, "Invalid input, please check fields!");
        }
    });
addBtn.addActionListener(e -> {
    try {
        String newSku = sku.getText();
        String newName = name.getText();
        double newPrice = Double.parseDouble(price.getText());
        double newDisc = Double.parseDouble(disc.getText());
        int newStock = Integer.parseInt(stock.getText());
addBtn.addActionListener(e -> { ... })
```

- This attaches an **event handler** to the “Add Product” button.
- When the button is clicked, the code inside { ... } runs.
- e is the event object representing the button click. Using a **lambda** makes this shorter than creating a full ActionListener class.

```
Product p = new Product(newSku, newName, newPrice, newDisc, newStock);
```

1. new Product(...)

- Creates a new Product object with the entered details.

2. products.add(p);

- Adds the new product to the inventory (MyArray), dynamically resizing if needed.

- When a product is **bought**, a **bill receipt** string is created.
- JOptionPane.showMessageDialog(f, bill, "Bill Receipt", JOptionPane.INFORMATION_MESSAGE);
  → This pops up the **bill** in a dialog window.
- Similarly, when you click **Show All Bills**, the accumulated billHistory string is shown using:
  JOptionPane.showMessageDialog(f, billHistory + "\n💰 Daily Total Sales: " + totalSales, ...)
  → This displays the **history of all bills** in a popup.

So, JOptionPane acts as the **visual popup layer** for bills and history.

- The e -> {} is a **lambda expression**, shorthand for ActionListener.