

DATA STRUCTURES AND ALGORITHMS

ASSIGNMENT 2 – LINKED LISTS

“Final Fantasy VII: Party, Inventory, and Materia Simulator”

Lecturer: Heri Wijayanto, ST., MT., Ph.D



Made by:

I NYOMAN WIDIYASA JAYANANDA
F1D02410053

**FACULTY OF ENGINEERING
DEPARTMENT OF INFORMATICS ENGINEERING
UNIVERSITY OF MATARAM
2025/2026**

A. Context or Scenario

This program demonstrates the concept of **Abstract Data Types (ADTs)** and their interactions through a **Final Fantasy VII-inspired scenario**. The narrative setting is a simplified adventure encounter where **party members, inventory, and materia** are modeled as different linked list structures, with each abstract parent class defining a contract for behavior.

Abstract Parent Classes

1. Nodes (**Node**)

- Represent the fundamental building blocks of any linked list.
- Abstract methods define how **next** (and optionally **prev**) links are managed.
- Different concrete implementations (**Singly_Node**, **Doubly_Node**) allow nodes to be chained in multiple styles.

2. Linked Lists (**Linked_List**)

- Represent the overarching container structure.
- Abstract methods like **add**, **remove**, and **find** define list operations.
- Concrete implementations include:
 - **Singly_Linked_List** – lightweight linear progression (used for inventory).
 - **Doubly_Linked_List** – supports forward/backward traversal (used for party).
 - **Circular_Linked_List** – loops continuously (used for materia ring).

Concrete Implementations (FF7 Scenario)

1. Party

- Models Cloud, Tifa, and Barrett as nodes in a **doubly linked list**.
- Allows flexible addition/removal of party members and bidirectional navigation.
- Example: Searching for *Tifa* demonstrates how the **find** method fulfills the contract defined by the abstract list.

2. Inventory

- Models items such as *Potion*, *Ether*, and *Hi-Potion*.
- Implemented as a **singly linked list**, emphasizing simple append and removal operations.
- Example: Removing *Ether* demonstrates how **remove** works with minimal references.

3. Materia Ring

- Models magical materia (*Fire*, *Cure*, *Restore*) arranged in a **circular linked list**.
- Simulates the rotation of materia slots; after reaching the end, traversal continues from the start.
- Example: Stepping through the ring multiple times demonstrates circular iteration.

Scenario Description

- The **party list** keeps track of current active characters.
- The **inventory list** stores consumable items the player can use or remove.
- The **materia ring** rotates to cycle through equipped spells.
- Demonstrations include adding/removing nodes, searching for members, and iterating through circular structures.

Educational Purpose

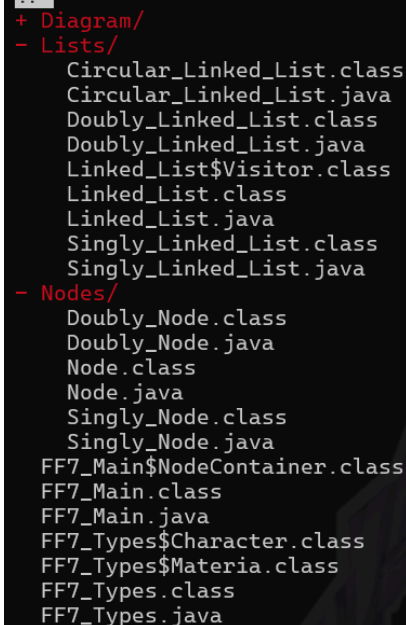
By framing the program in an **FF7 scenario**, students can better visualize abstraction, inheritance, and polymorphism concepts:

- **Abstract classes** (**Linked_List**, **Node**) define *contracts for data structures*.
- **Concrete child classes** (**Singly_Linked_List**, **Doubly_Linked_List**, **Circular_Linked_List**; **Singly_Node**, **Doubly_Node**) implement specific structural rules.
- **Interactions** show polymorphism in action: the same **add**, **remove**, and **find** methods behave differently depending on the list type.

This approach highlights **object-oriented design principles** with a sprinkle of engagement by mapping code elements to a familiar game world. The GitHub link for this assignment is <https://github.com/Schryzon/DSA-Assignments/tree/master/Second>.

Project Directory Structure

The Java program is laid out as such:



```
+ Diagram/  
- Lists/  
  Circular_Linked_List.class  
  Circular_Linked_List.java  
  Doubly_Linked_List.class  
  Doubly_Linked_List.java  
  Linked_List$Visitor.class  
  Linked_List.class  
  Linked_List.java  
  Singly_Linked_List.class  
  Singly_Linked_List.java  
- Nodes/  
  Doubly_Node.class  
  Doubly_Node.java  
  Node.class  
  Node.java  
  Singly_Node.class  
  Singly_Node.java  
FF7_Main$NodeContainer.class  
FF7_Main.class  
FF7_Main.java  
FF7_Types$Character.class  
FF7_Types$Materia.class  
FF7_Types.class  
FF7_Types.java
```

Figure 1. Project Directory Structure

In text-based tree form:

```
SECOND/ → Root assignment directory (Shown as ".." in Figure 1).  
|  
├─ Diagram/ → Stores diagram figures for this assignment.  
|  
├─ Lists/ → Abstract parent class (Linked_List) and implementations.  
|   ├─ Circular_Linked_List.java  
|   ├─ Doubly_Linked_List.java  
|   ├─ Linked_List.java  
|   └─ Singly_Linked_List.java  
|
```

└─	Nodes/ → Abstract parent class (Node) and node implementations.
	└─ Doubly_Node.java
	└─ Node.java
	└─ Singly_Node.java
└─	FF7_Main.java → Main program file (entry point).
└─	FF7_Types.java → Contains classes for scenario objects.

Table 1. Project Tree Structure

B. Source Code and Explanation

```
package Nodes;

public abstract class Node<T>{ // Generic data, using getters & setters
    protected T data;
    public Node(T data){
        this.data = data;
    }

    public T get_data(){
        return this.data;
    }

    public void set_data(T data){
        this.data = data;
    }

    public static <T> boolean equals_safe(T a, T b){
        return (a == null && b == null) || (a != null && a.equals(b));
    }
    public abstract Node<T> get_next();
    public abstract void set_next(Node<T> next);
    public Node<T> get_prev(){
        // If not supported, give null
        return null;
    }
    public void set_prev(Node<T> prev){
        // No operation if not Doubly
    }
}
```

Table 2. Abstract Node Parent Class (Nodes/Node.java)

The file **Node.java** defines an **abstract generic class** that models the fundamental unit of any linked list structure within the Final Fantasy VII–inspired system. As an **Abstract Data Type (ADT)**, it specifies the universal state and behaviors every node must provide, while leaving structural details (such as next/previous linkage) to be implemented by concrete subclasses like **Singly_Node** or **Doubly_Node**. This abstraction ensures that all node types can interoperate under a common contract, supporting polymorphism across different list designs.

The class maintains a single protected field, **data**, which stores the node's payload in a **generic form (T)**, making it flexible enough to represent different scenario objects (e.g., party members, items, or materia). It provides general-purpose accessor methods (**get_data**, **set_data**) for controlled access to this payload.

Utility is extended through the static method **equals_safe**, which performs null-safe equality checks between two values. It's a crucial safeguard for list operations such as searching or removal where nulls may appear.

Most importantly, the class defines **abstract linkage methods** (**get_next**, **set_next**), compelling all subclasses to establish how nodes connect within their respective list types. Default implementations of **get_prev** and **set_prev** are also provided: they return/do nothing unless overridden, thereby supporting singly linked lists by default while allowing doubly linked nodes to enrich the contract.

Altogether, this file demonstrates the role of **node abstraction** in data structure design: it separates the universal concept of a "node" from the specific mechanics of how different linked list variants manage connectivity.

```
package Nodes;

public class Singly_Node<T> extends Node<T>{
    private Node<T> next;
    public Singly_Node(T data){
        super(data);
        this.next = null;
    }
    @Override
    public Node<T> get_next(){
        return this.next;
    }
    @Override
    public void set_next(Node<T> next){
        this.next = next;
    }
}
```

Table 3. Singly Linked List Node (Nodes/Singly_Node.java)

The file **Singly_Node.java** defines a concrete node type for singly linked lists by extending the abstract **Node<T>** class. It introduces a **next** reference to link to the following node and implements the required **get_next** and **set_next** methods.

Because it doesn't override **get_prev** or **set_prev**, singly nodes remain forward-only, distinguishing them from doubly or circular variants. This keeps the structure lightweight while still conforming to the common **Node<T>** abstraction, ensuring polymorphic compatibility across different list types.

```
package Nodes;

public class Doubly_Node<T> extends Node<T>{
    private Node<T> next;
    private Node<T> prev;
    public Doubly_Node(T data){
        super(data);
        this.next = null;
        this.prev = null;
    }
    @Override
    public Node<T> get_next(){
        return this.next;
    }
    @Override
    public void set_next(Node<T> next){
        this.next = next;
    }
    @Override
    public Node<T> get_prev(){
        return this.prev;
    }
    @Override
    public void set_prev(Node<T> prev){
        this.prev = prev;
    }
}
```

Table 4. Doubly Linked List Node (Nodes/Doubly_Node.java)

The file **Doubly_Node.java** provides a concrete implementation of a node designed for **doubly linked lists**, extending the shared **Node<T>** abstraction. Unlike the singly variant, it maintains two references: **next** for the forward link and **prev** for the backward link.

It overrides both sets of navigation methods: **get_next** / **set_next** and **get_prev** / **set_prev**, ensuring that movement in both directions is supported. This allows for more

flexible list operations, such as reverse traversal or efficient insertion and removal at both ends.

By staying within the same abstract contract as `Node<T>`, this class remains interchangeable with other node types, but it clearly establishes the richer connectivity needed for list types that demand **bidirectional access and manipulation**.

```
package Lists;
import Nodes.*;

public abstract class Linked_List<T>{
    protected Node<T> head;
    protected Node<T> tail;
    protected int size;
    public Linked_List(){
        this.head = null;
        this.tail = null;
        this.size = 0;
    }
    // Can be append or push
    public abstract void add(T data);

    // Remove by first occurrence
    public abstract boolean remove(T data);

    // Find first equal
    public abstract Node<T> find(T data);
    public int get_size(){
        return this.size;
    }
    public boolean is_empty(){
        return this.size == 0;
    }
    // Iterate and apply a visitor lambda
    public void for_each(Visitor<T> visitor){
        Node<T> current = this.head;
        int iter_count = 0;
        while(current != null && iter_count < this.size){
            visitor.visit(current.get_data());
            current = current.get_next();
            iter_count++;
        }
    }
}
```

```

    }
    public Node<T> get_head(){
        return this.head;
    }
    public interface Visitor<T>{
        void visit(T item);
    }
}

```

Table 5. Abstract Linked List Parent Class (Lists/Linked_List.java)

The file **Linked_List.java** defines the **core abstraction** for all list structures in the system. It provides the common state: **head**, **tail**, and **size**, as well as essential behaviors that every linked list type (singly, doubly, or circular) will share.

At its heart, this abstract class enforces three key operations via abstract methods:

- **add(T data)**: handles insertion, leaving flexibility for subclasses to decide whether it behaves like an append, push, or other variant.
- **remove(T data)**: specifies removal by first occurrence, ensuring consistent semantics across implementations.
- **find(T data)**: allows node search based on equality, tying directly into the **equals_safe** utility from the **Node** base class.

On top of this, it defines concrete helpers:

- **get_size()** and **is_empty()** expose size tracking.
- **for_each(Visitor<T> visitor)** introduces a functional-style traversal pattern.

The inclusion of the **Visitor interface** here is significant; it separates iteration from operation, enabling clients to apply behaviors (like printing, aggregation, or combat simulation in the FF7 setting) without changing the underlying structure. The **iter_count** safeguard ensures protection against infinite loops, which is especially critical when dealing with **circular lists**.

This class is the architectural **spine of the linked list hierarchy**, balancing shared invariants with open points for customization. It ensures that any specialized list still adheres to the same interface while benefiting from the reusable traversal and size-management logic.

```

package Lists;
import Nodes.*;

public class Singly_Linked_List<T> extends Linked_List<T>{
    public Singly_Linked_List(){
        super();
    }
}

```



```

@Override
public void add(T data){
    Node<T> node = new Singly_Node<>(data);
    if(this.head == null){
        this.head = node;
        this.tail = node;

    }else{
        this.tail.set_next(node);
        this.tail = node;
    }
    this.size++;
}

@Override
public boolean remove(T data){
    Node<T> prev = null;
    Node<T> current = this.head;
    while(current != null){
        if(Node.equals_safe(current.get_data(), data)){
            if(prev == null){
                this.head = current.get_next();
                if(this.head == null){
                    this.tail = null;
                }
            }else{
                prev.set_next(current.get_next());
                if(current == this.tail){
                    this.tail = prev;
                }
            }
            this.size--;
            return true;
        }
        prev = current;
        current = current.get_next();
    }
    return false;
}

```

```

@Override
public Node<T> find(T data){
    Node<T> current = this.head;
    while(current != null){
        if(Node.equals_safe(current.get_data(), data)){
            return current;
        }
        current = current.get_next();
    }
    return null;
}
}

```

Table 6. Singly Linked List (Lists/Singly_Linked_List.java)

The file **Singly_Linked_List.java** provides a concrete implementation of the abstract **Linked_List** blueprint using singly linked nodes. Its constructor simply delegates to the parent, keeping initialization consistent across all list types. The **add** method appends elements to the end of the list by creating a new **Singly_Node**. If the list is empty, both head and tail are set to the new node; otherwise, the current tail links forward to it, and the tail reference updates. Each insertion increments the size counter, ensuring proper tracking.

Removal is handled by traversing the list with a **prev** pointer. When the first matching data is found, the method splices out the corresponding node. Special care is taken if the node to remove is the head, in which case the head reference is updated. If the removal affects the tail, the tail reference is also corrected. This guarantees structural integrity even as elements are removed dynamically.

The **find** method provides linear search by walking through the nodes until a data match is found, using the **equals_safe** helper to avoid null pitfalls. If a match is discovered, the node itself is returned; if not, the method yields null.

Altogether, this class represents the simplest linked list form: a forward-only chain. It demonstrates the value of the abstract **Linked_List** contract while showing how the basic mechanics of adding, removing, and searching can be realized in a minimal but reliable manner.

```

package Lists;
import Nodes.*;

public class Doubly_Linked_List<T> extends Linked_List<T>{
    public Doubly_Linked_List(){
        super();
    }

    @Override

```

```

public void add(T data){
    Doubly_Node<T> node = new Doubly_Node<>(data);
    if(this.head == null){
        this.head = node;
        this.tail = node;
    }else{
        this.tail.set_next(node);
        node.set_prev(this.tail);
        this.tail = node;
    }
    this.size++;
}

@Override
public boolean remove(T data){
    Node<T> current = this.head;
    while(current != null){
        if(Node.equals_safe(current.get_data(), data)){
            Node<T> prev = current.get_prev();
            Node<T> next = current.get_next();
            if(prev != null){
                prev.set_next(next);
            }else{
                this.head = next;
            }

            if(next != null){
                next.set_prev(prev);
            }else{
                this.tail = prev;
            }
            this.size--;
            return true;
        }
        current = current.get_next();
    }
    return false;
}

@Override
public Node<T> find(T data){

```

```

        Node<T> current = this.head;
        while(current != null){
            if(Node.equals_safe(current.get_data(), data)){
                return current;
            }
            current = current.get_next();
        }
        return null;
    }
}

```

Table 7. Doubly Linked List (Lists/Doubly_Linked_List.java)

The file **Doubly_Linked_List.java** extends the abstract **Linked_List** framework with a full **bidirectional chain** implementation. Unlike the singly variant, it relies on **Doubly_Node** objects, which maintain both **next** and **prev** links, allowing traversal and modification from either direction.

The **add** method creates a new node and attaches it at the tail. If the list is empty, both head and tail are initialized. Otherwise, the existing tail's **next** is set to the new node, the new node's **prev** points back to the tail, and the tail reference is shifted forward. This symmetry ensures a consistent two-way linkage while keeping insertion efficient at the end.

Removal carefully re-links both sides of the target node. On finding a match, it retrieves **prev** and **next** references and reconnects them around the removed element. Special handling ensures the head and tail pointers update properly if the first or last node is deleted. By maintaining both forward and backward pointers, the method avoids unnecessary traversal overhead.

The **find** method mirrors its singly-linked counterpart, walking from the head until a match is located or the chain ends. Although traversal remains forward by default, the doubly-linked design enables potential enhancements such as reverse searching from the tail if desired.

This implementation shows how the general linked list abstraction can be enriched with bidirectional structure, giving faster operations for certain use cases while staying faithful to the unified contract established in **Linked_List**.

```

package Lists;
import Nodes.*;

public class Circular_Linked_List<T> extends Linked_List<T>{
    public Circular_Linked_List(){
        super();
    }

    @Override

```

```

public void add(T data){
    Node<T> node = new Singly_Node<>(data);
    if(this.head == null){
        this.head = node;
        this.tail = node;
        this.tail.set_next(this.head); // Circular link
    }else{
        this.tail.set_next(node);
        this.tail = node;
        this.tail.set_next(this.head); // Keep circular
    }
    this.size++;
}

@Override
public boolean remove(T data){
    if(this.head == null){
        return false;
    }
    Node<T> current = this.head;
    Node<T> prev = this.tail;
    int iter_count = 0;
    while(iter_count < this.size){
        if(Node.equals_safe(current.get_data(), data)){
            if(current == this.head){
                this.head = this.head.get_next();
                this.tail.set_next(this.head);
                if(this.size == 1){
                    this.head = null;
                    this.tail = null;
                }
            }else{
                prev.set_next(current.get_next());
                if(current == this.tail){
                    this.tail = prev;
                }
            }
            this.size--;
            return true;
        }
        prev = current;
    }
}

```

```

        current = current.get_next();
        iter_count++;
    }
    return false;
}

@Override
public Node<T> find(T data){
    if(this.head == null){
        return null;
    }
    Node<T> current = this.head;
    int iter_count = 0;
    while(iter_count < this.size){
        if(Node.equals_safe(current.get_data(), data)){
            return current;
        }
        current = current.get_next();
        iter_count++;
    }
    return null;
}

@Override
public void for_each(Visitor<T> visitor){
    Node<T> current = this.head;
    int iter_count = 0;
    while(current != null && iter_count < this.size){
        visitor.visit(current.get_data());
        current = current.get_next();
        iter_count++;
    }
}
}

```

Table 8. Circular Linked List (Lists/Circular_Linked_List.java)

The file **Circular_Linked_List.java** builds on the **Linked_List** abstraction by creating a structure where the tail always links back to the head, forming a closed loop. This design makes traversal naturally cyclical, useful in applications like round-robin scheduling or game turn systems.

The **add** method either initializes the list with a single node pointing to itself, or, if the list is non-empty, attaches the new node at the tail and reconnects the tail's **next** back to the head to preserve circularity.

The **remove** method has extra considerations compared to singly or doubly lists. Since there is no natural end marker, it tracks both the current node and its predecessor while iterating up to **size** times. If the target is the head, it advances the head reference and updates the tail's link. If the list shrinks to a single element, both head and tail are nulled. For other nodes, the predecessor bypasses the removed node.

The **find** method follows a similar bounded traversal strategy, cycling through the loop at most **size** steps. This ensures it won't get stuck in infinite iteration, a risk unique to circular designs.

Finally, the **for_each** override maintains the same traversal discipline, applying a visitor action exactly **size** times regardless of circular continuity.

Through this file, the linked list abstraction is reimaged into a cyclic structure that emphasizes control over traversal limits, showing how circular connectivity changes list dynamics while preserving the shared ADT contract.

```
/**
 * FF7 flavored data classes. Simple POJOs used as payloads in nodes.
 */
public class FF7_Types {
    public static class Character {
        public String name;
        public int hp;
        public int level;

        public Character(String name, int hp, int level) {
            this.name = name;
            this.hp = hp;
            this.level = level;
        }

        @Override
        public String toString() {
            return String.format("%s (Lv %d, HP %d)", this.name, this.level,
this.hp);
        }

        @Override
        public boolean equals(Object o) {
            if (o == null || !(o instanceof Character)) {
```

```

        return false;
    }
    Character other = (Character) o;
    return this.name.equals(other.name);
}
}

public static class Materia {
    public String name;
    public int grade;

    public Materia(String name, int grade) {
        this.name = name;
        this.grade = grade;
    }

    @Override
    public String toString() {
        return String.format("%s (G%d)", this.name, this.grade);
    }

    @Override
    public boolean equals(Object o) {
        if (o == null || !(o instanceof Materia)) {
            return false;
        }
        Materia other = (Materia) o;
        return this.name.equals(other.name) && this.grade ==
other.grade;
    }
}
}

```

Table 9. Final Fantasy VII Object Types (FF7_Types.java)

The file **FF7_Types.java** provides the concrete data objects that serve as the **payloads** stored inside nodes of the various linked list implementations. These are deliberately kept simple, acting as plain old Java objects (POJOs) that carry game-relevant information for the Final Fantasy VII–flavored scenario.

The nested **Character** class models a party member with attributes for **name**, **hp**, and **level**. Its **toString** method presents a compact status summary, while the **equals** method treats two characters as identical if their names match, regardless of stats. This reflects how characters in the game are identified by identity rather than current health or progression.

The nested **Materia** class represents magical or support orbs that characters equip. It records both the **name** and a **grade** value, with the **toString** producing a short display like “Fire (G2).” Equality here is stricter than in **Character**, requiring both the name and grade to match, since a level 1 Materia and a level 3 Materia are meaningfully different items.

By isolating these two types in their own utility container, the program separates **data modeling** from **list infrastructure**. This allows lists to remain generic and reusable, while the FF7 domain logic is captured neatly within these self-contained classes.

```
/**
 * Interactive FFVII linked list manager.
 *
 * Provides:
 * - Party (Doubly Linked List) CRUD
 * - Inventory (Singly Linked List) CRUD
 * - Materia ring (Circular Linked List) CRUD + rotate simulation
 *
 * Compile:
 * javac Nodes\*.java Lists\*.java FF7_Types.java FF7_Main.java
 * Run:
 * java -cp . FF7_MainInteractive
 */
import java.util.Scanner;
import Nodes.*;
import Lists.*;

public class FF7_Main{
    private static final Scanner input = new Scanner(System.in);
    public static void main(String[] args) {
        System.out.println("=== FINAL FANTASY VII ===");

        // Lists (use your concrete list implementations)
        Doubly_Linked_List<FF7_Types.Character> party = new
Doubly_Linked_List<>();
        Singly_Linked_List<String> inventory = new Singly_Linked_List<>();
        Circular_Linked_List<FF7_Types.Materia> materia_ring = new
Circular_Linked_List<>();

        // Seed some entries (optional)
        party.add(new FF7_Types.Character("Cloud", 1200, 50));
        party.add(new FF7_Types.Character("Tifa", 1100, 48));
        inventory.add("Potion");
    }
}
```

```

inventory.add("Ether");
materia_ring.add(new FF7_Types.Materia("Fire", 3));
materia_ring.add(new FF7_Types.Materia("Cure", 2));

main_loop:
while (true) {
    System.out.println("\nMain Menu:");
    System.out.println(" 1) Party (manage characters)");
    System.out.println(" 2) Inventory (items)");
    System.out.println(" 3) Materia Ring");
    System.out.println(" 4) Show summary");
    System.out.println(" 0) Exit");
    System.out.print("> ");

    String choice = input.nextLine().trim();
    clear_screen();
    switch (choice) {
        case "1":
            party_menu(party);
            break;
        case "2":
            inventory_menu(inventory);
            break;
        case "3":
            materia_menu(materia_ring);
            break;
        case "4":
            show_summary(party, inventory, materia_ring);
            break;
        case "0":
            System.out.println("Program exited, until then!");
            break main_loop;
        default:
            System.out.println("Invalid option, try again.");
    }
    wait_for_enter();
    clear_screen();
}
input.close();
}

```

```

/**
 * Party submenu: add, remove, list, find, edit hp/level.
 */
private static void party_menu(Doubly_Linked_List<FF7_Types.Character>
party) {
    while (true) {
        System.out.println("\nParty Menu:");
        System.out.println(" 1) List party members");
        System.out.println(" 2) Add member");
        System.out.println(" 3) Remove member (by name)");
        System.out.println(" 4) Find member (by name)");
        System.out.println(" 5) Edit member (hp / level)");
        System.out.println(" 0) Back");
        System.out.print("> ");

        String c = input.nextLine().trim();
        switch (c) {
            case "1":
                System.out.println("\n-- Party members --");
                party.for_each(ch -> System.out.println(" - " + ch));
                System.out.println("-- END --");
                break;
            case "2":
                System.out.print("Name: ");
                String name = input.nextLine().trim();
                int hp = prompt_int("HP: ", 1, Integer.MAX_VALUE);
                int level = prompt_int("Level: ", 1, 999);
                party.add(new FF7_Types.Character(name, hp, level));
                System.out.println("Added " + name);
                break;
            case "3":
                System.out.print("Name to remove: ");
                String rem_name = input.nextLine().trim();
                boolean removed = party.remove(new
FF7_Types.Character(rem_name, 0, 0));
                System.out.println("Removed? " + removed);
                break;
            case "4":
                System.out.print("Name to find: ");
                String find_name = input.nextLine().trim();
                Node<FF7_Types.Character> found = party.find(new

```

```

FF7_Types.Character(find_name, 0, 0));
    if (found != null) {
        System.out.println("Found -> " + found.get_data());
    } else {
        System.out.println("Not found.");
    }
    break;
case "5":
    System.out.print("Name to edit: ");
    String edit_name = input.nextLine().trim();
    Node<FF7_Types.Character> node = party.find(new
FF7_Types.Character(edit_name, 0, 0));
    if (node == null) {
        System.out.println("No such member.");
        break;
    }
    FF7_Types.Character ch = node.get_data();
    System.out.println("Editing " + ch);
    int new_hp = prompt_int("New HP (" + ch.hp + "): ", 1,
Integer.MAX_VALUE);
    int new_level = prompt_int("New Level (" + ch.level +
"): ", 1, 999);

    ch.hp = new_hp;
    ch.level = new_level;
    System.out.println("Updated -> " + ch);
    break;
case "0":
    return;
default:
    System.out.println("Invalid option.");
}
wait_for_enter();
clear_screen();
}
}

/**
 * Inventory submenu: add, remove, list.
 */
private static void inventory_menu(Singly_Linked_List<String> inventory)
{

```

```

while (true) {
    System.out.println("\nInventory Menu:");
    System.out.println(" 1) List items");
    System.out.println(" 2) Add item");
    System.out.println(" 3) Remove item (by name)");
    System.out.println(" 0) Back");
    System.out.print("> ");

    String c = input.nextLine().trim();
    switch (c) {
        case "1":
            System.out.println("\n-- Inventory --");
            inventory.forEach(it -> System.out.println(" * " +
it));

            System.out.println("-- END --");
            break;
        case "2":
            System.out.print("Item name: ");
            String item = input.nextLine().trim();
            inventory.add(item);
            System.out.println("Added " + item);
            break;
        case "3":
            System.out.print("Item to remove: ");
            String rem = input.nextLine().trim();
            boolean ok = inventory.remove(rem);
            System.out.println("Removed? " + ok);
            break;
        case "0":
            return;
        default:
            System.out.println("Invalid option.");
    }
    wait_for_enter();
    clear_screen();
}

/**
 * Materia submenu: add, remove, list, rotate simulation.
 */

```

```

    private static void materia_menu(Circular_Linked_List<FF7_Types.Materia>
ring) {
    while (true) {
        System.out.println("\nMateria Menu:");
        System.out.println(" 1) List materia");
        System.out.println(" 2) Add materia");
        System.out.println(" 3) Remove materia (by name+grade)");
        System.out.println(" 4) Rotate (simulate advancing slots)");
        System.out.println(" 0) Back");
        System.out.print("> ");

        String c = input.nextLine().trim();
        switch (c) {
            case "1":
                System.out.println("\n-- Materia ring --");
                ring.for_each(m -> System.out.println(" ~ " + m));
                System.out.println("-- END --");
                break;
            case "2":
                System.out.print("Materia name: ");
                String name = input.nextLine().trim();
                int grade = prompt_int("Grade: ", 1, 9);
                ring.add(new FF7_Types.Materia(name, grade));
                System.out.println("Added " + name);
                break;
            case "3":
                System.out.print("Materia name to remove: ");
                String rname = input.nextLine().trim();
                int rgrade = prompt_int("Grade: ", 1, 9);
                boolean r = ring.remove(new FF7_Types.Materia(rname,
rgrade));

                System.out.println("Removed? " + r);
                break;
            case "4":
                rotate_materia_sim(ring);
                break;
            case "0":
                return;
            default:
                System.out.println("Invalid option.");
        }
    }
}

```

```

        wait_for_enter();
        clear_screen();
    }
}

/**
 * Simulate rotating the circular materia ring starting from
 * either a named materia or the first present entry.
 *
 * Implementation note: cannot directly access protected head,
 * so use find() and for_each() to obtain a starting Node reference.
 */
private static void
rotate_materia_sim(Circular_Linked_List<FF7_Types.Materia> ring) {
    int size = ring.get_size();
    if (size == 0) {
        System.out.println("Ring is empty.");
        return;
    }

    System.out.print("Start from materia name (leave empty to use
first): ");
    String start_name = input.nextLine().trim();

    Node<FF7_Types.Materia> start_node = null;

    if (start_name.length() > 0) {
        // Build a query materia with same name (grade unknown -> try several)
        // Attempt to find by name by scanning and comparing name field.
        // Because find() compares objects via equals, and
FF7_Types.Materia.equals requires exact grade,
        // Fallback to scanning with a for_each to capture Node if name
matches.

        final NodeContainer first_match = new NodeContainer();
        ring.for_each(m -> {
            if (first_match.node == null && m.name.equals(start_name)) {
                // ??? We want the Node, but for_each only gives data object.
                // So capture the data, then find the node with exact object.
                first_match.captured_data = m;
            }
        });
    }
}

```

```

        if (first_match.captured_data != null) {
            start_node = ring.find(first_match.captured_data);
        } else {
            System.out.println("Named materia not found. Using first
element instead.");
        }
    }

    // If still no start_node, pick the first element captured by
for_each
    if (start_node == null) {
        final NodeContainer any = new NodeContainer();
        ring.for_each(m -> {
            if (any.captured_data == null) {
                any.captured_data = m;
            }
        });
        start_node = ring.find(any.captured_data);
    }

    if (start_node == null) {
        System.out.println("Could not obtain starting node
(unexpected).");
        return;
    }

    int steps = prompt_int("How many steps to simulate? ", 1, 1000);
    System.out.println("Rotating starting at -> " +
start_node.get_data());
    Node<FF7_Types.Materia> cur = start_node;
    int step = 0;
    while (step < steps && cur != null) {
        System.out.println(" Step " + step + " -> " + cur.get_data());
        cur = cur.get_next();
        step++;
        // Safety: break if looped more than size*2 (prevent infinite)
        if (step > size * 2) {
            System.out.println("Safety break triggered!");
            break;
        }
    }
}

```



```

    }

    /**
     * Helper: prompt integer with min/max validation.
     */
    private static int prompt_int(String prompt, int min, int max) {
        while (true) {
            System.out.print(prompt);
            String line = input.nextLine().trim();
            try {
                int v = Integer.parseInt(line);
                if (v < min || v > max) {
                    System.out.println("Value must be between " + min + "
and " + max + ".");
                    continue;
                }
                return v;
            } catch (NumberFormatException ex) {
                System.out.println("Please enter a valid number.");
            }
        }
    }

    /**
     * Print quick sizes and short lists.
     */
    private static void show_summary(Doubly_Linked_List<FF7_Types.Character>
party, Singly_Linked_List<String> inventory,
Circular_Linked_List<FF7_Types.Materia> ring) {
        System.out.println("\n=== Summary ===");
        System.out.println(" Party size: " + party.get_size());
        System.out.println(" Inventory size: " + inventory.get_size());
        System.out.println(" Materia size: " + ring.get_size());
        System.out.println("\nParty:");
        party.for_each(ch -> System.out.println(" - " + ch));
        System.out.println("\nInventory:");
        inventory.for_each(it -> System.out.println(" * " + it));
        System.out.println("\nMateria:");
        ring.for_each(m -> System.out.println(" ~ " + m));
    }

```

```

private static void clear_screen(){
    try{
        if(System.getProperty("os.name").contains("Windows")){
            new ProcessBuilder("cmd", "/c",
"cls").inheritIO().start().waitFor();
        }else{
            System.out.print("\033[H\033[2J");
            System.out.flush();
        }
    }catch(Exception e){
        System.out.println("Could not clear screen.");
    }
}

private static void wait_for_enter() {
    System.out.println("\nPress ENTER to continue...");
    input.nextLine();
}

/**
 * Small mutable container so lambda can write a Node-like result.
 * Only store the captured data object then call find() to get Node.
 */
private static class NodeContainer {
    public FF7_Types.Materia captured_data = null;
    public Node<FF7_Types.Materia> node = null;
}
}

```

Table 10. Main Class Implementation (FF7_Main.java)

This **FF7_Main** class works as an interactive console driver that demonstrates how the three different linked list types behave in practical scenarios. It organizes functionality into three subsystems: party, inventory, and materia ring, each mapped to the list structure that best fits their expected operations.

The party uses a doubly linked list of **FF7_Types.Character**. This choice allows traversal in both directions, which makes sense for something like a roster that a player might scroll through. CRUD operations are implemented: characters can be added, removed, searched, or edited. Editing is handled in-place on the node's stored object, showing how lists can hold complex, mutable data.

The inventory uses a singly linked list of **String**. This keeps it lightweight since only one-way traversal is needed, and items don't require reverse navigation or deep editing. The

menu offers simple add/remove/find operations, which mirror basic inventory management in an RPG.

The materia ring is based on a circular linked list of `FF7_Types.Materia`. This structure fits the conceptual design of a ring, where traversal naturally loops. The rotate simulation emphasizes how a circular list differs from linear ones; the head can keep shifting without losing continuity. The implementation cleverly uses `for_each` with a captured reference and then `find` to recover the node, maintaining encapsulation of internal pointers.

Utility methods like `prompt_int`, `clear_screen`, and `wait_for_enter` keep the menu code clean and readable. The `clear_screen` approach also ensures a reasonable cross-platform behavior, falling back to ANSI escape codes if needed.

Finally, the summary option ties everything together by showing the current state of all three lists at once. This provides immediate feedback and validates that the underlying data structures behave as expected after multiple operations.

C. Diagrams and Relations

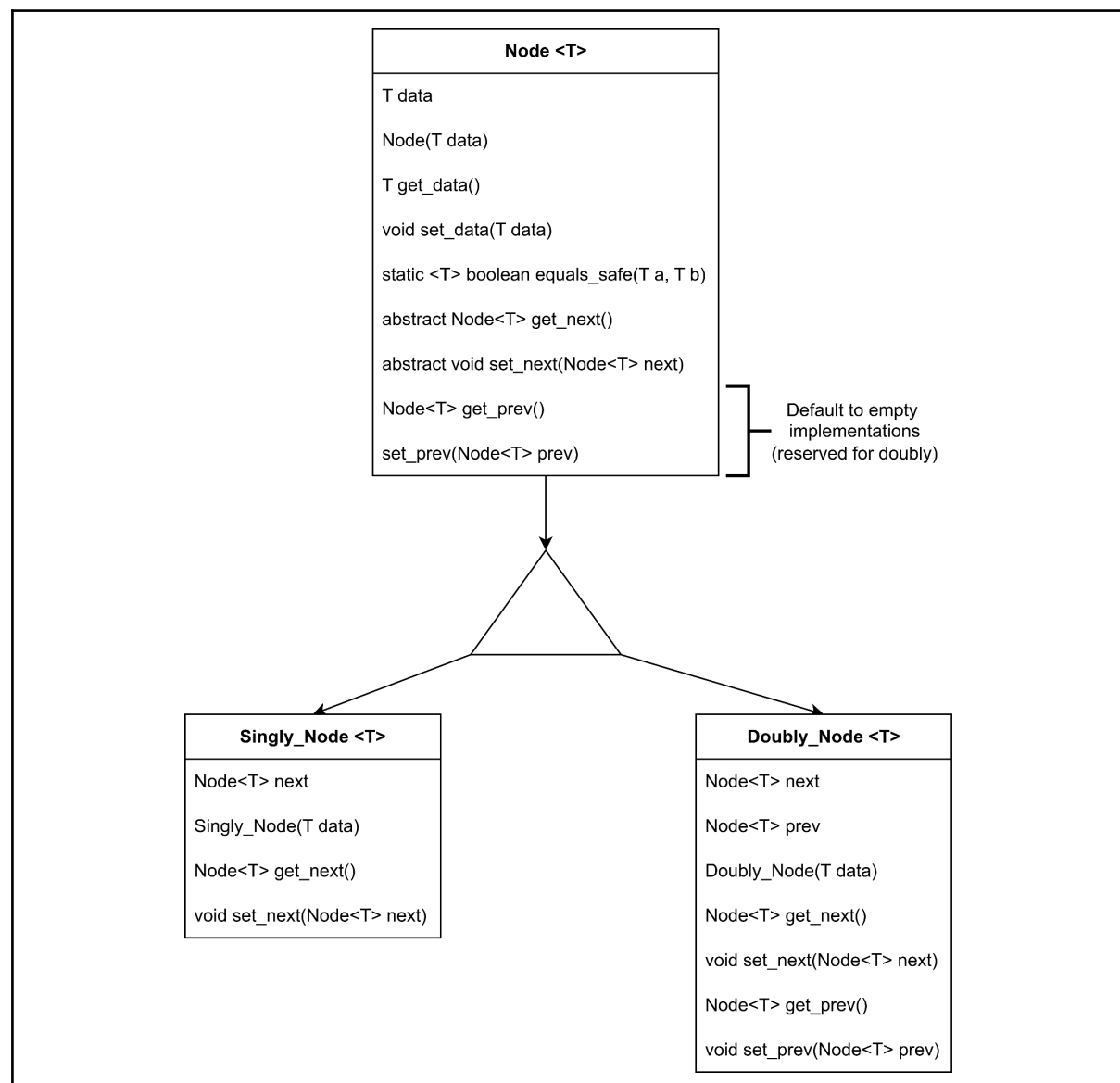


Figure 2. Diagram of Nodes

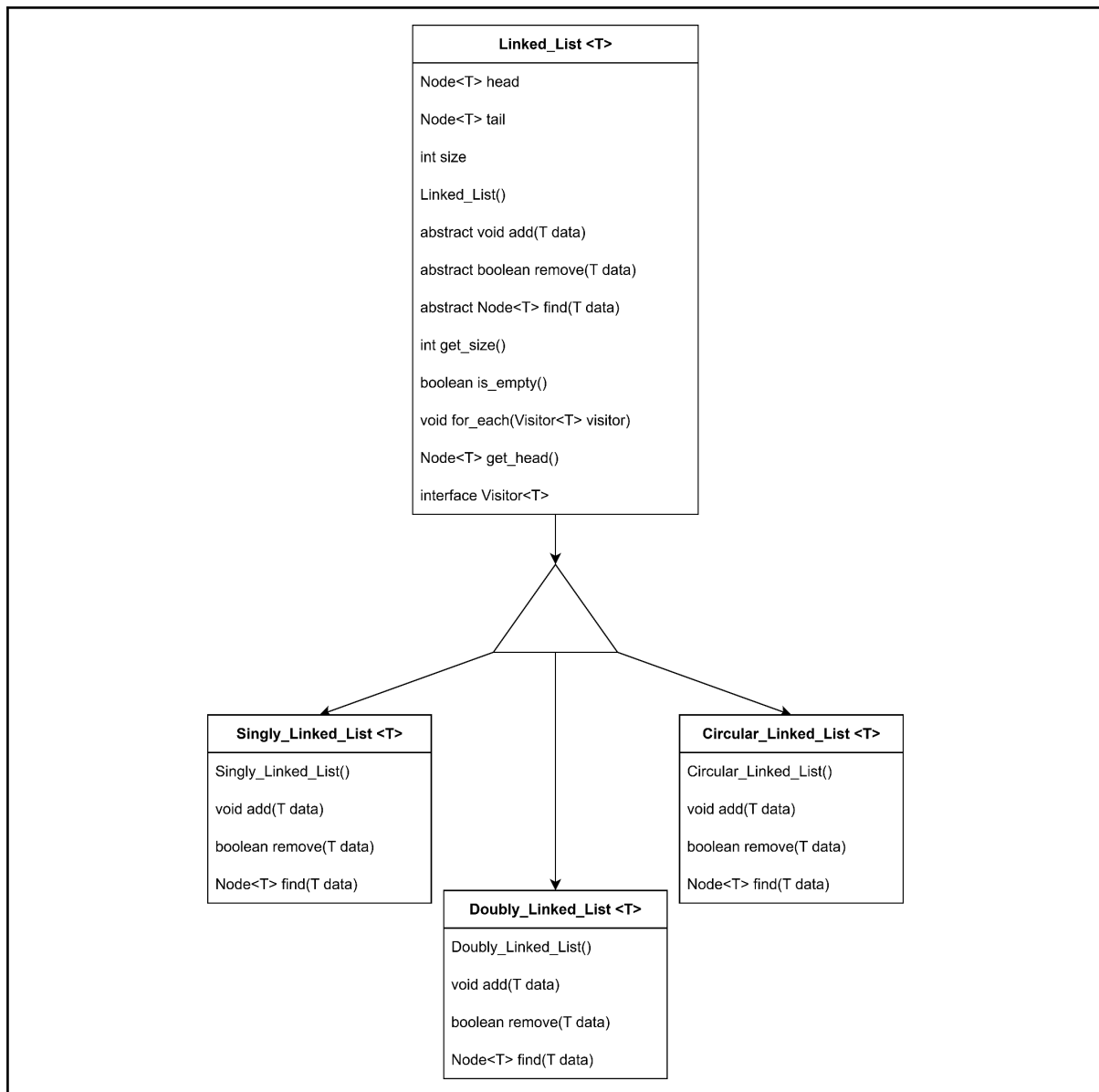


Figure 3. Diagram of Lists

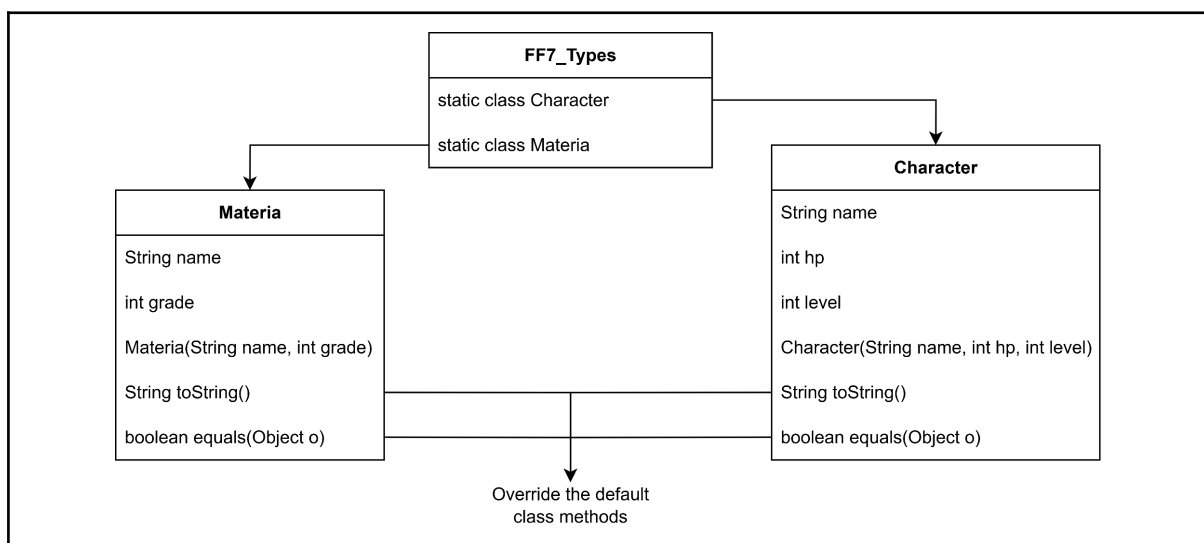
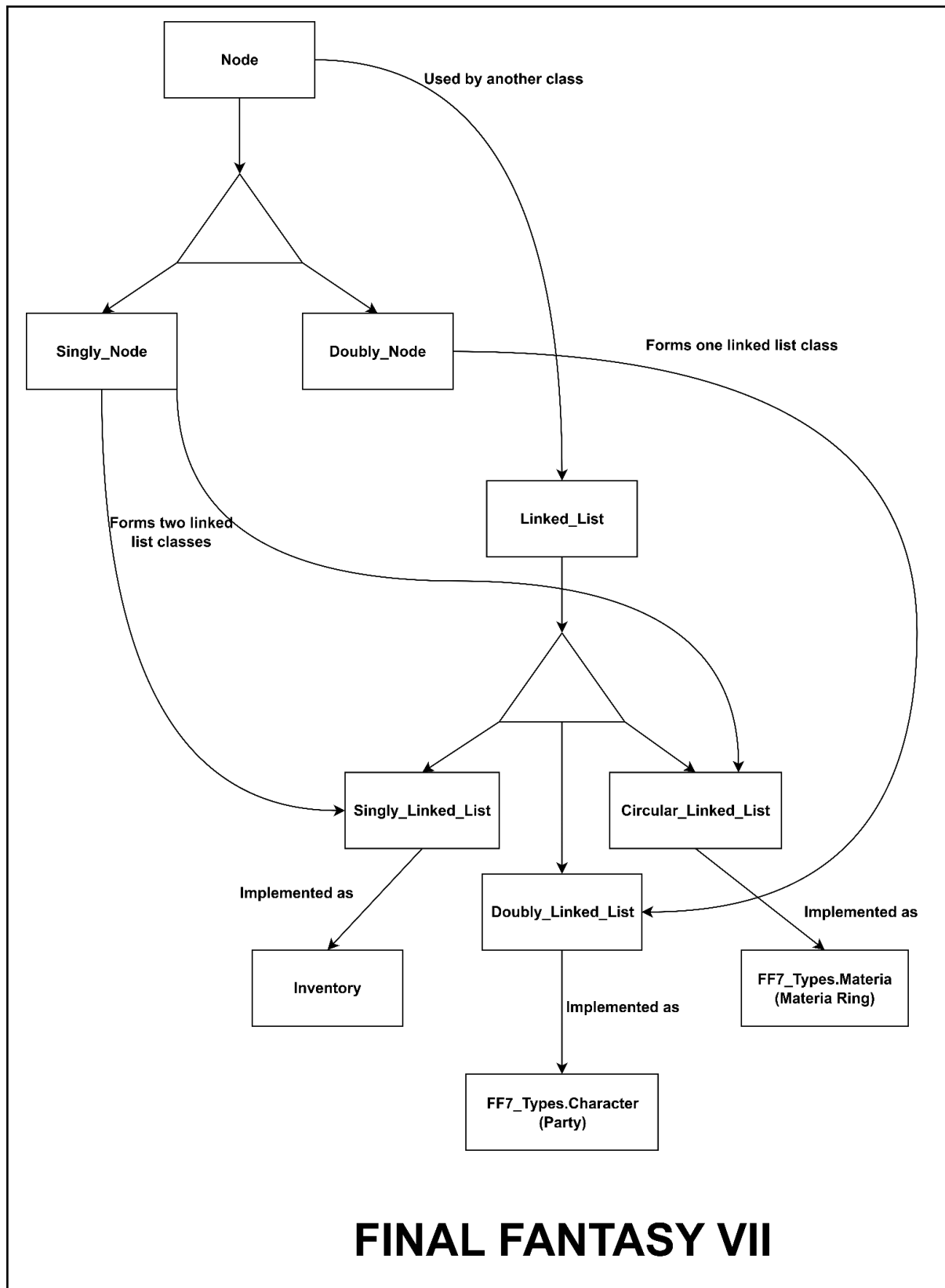
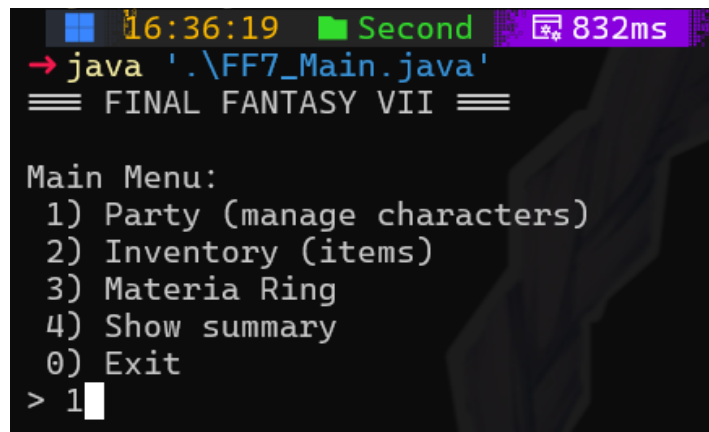


Figure 4. Diagram of the FF7_Types Nested Classes



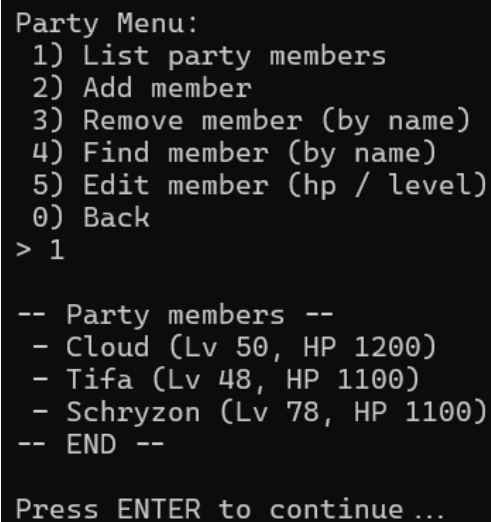
D. Program Result



```
16:36:19 Second 832ms
→ java \\.\\FF7_Main.java'
===== FINAL FANTASY VII =====

Main Menu:
 1) Party (manage characters)
 2) Inventory (items)
 3) Materia Ring
 4) Show summary
 0) Exit
> 1
```

Figure 6. Main Menu

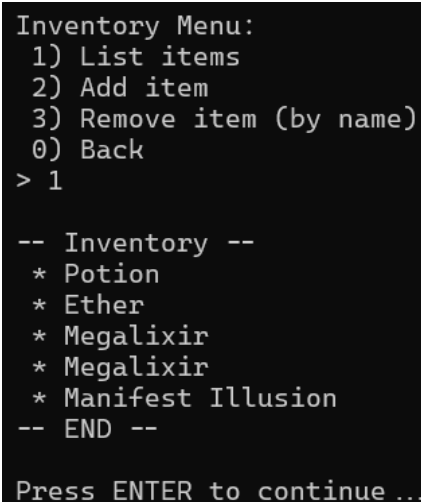


```
Party Menu:
 1) List party members
 2) Add member
 3) Remove member (by name)
 4) Find member (by name)
 5) Edit member (hp / level)
 0) Back
> 1

-- Party members --
- Cloud (Lv 50, HP 1200)
- Tifa (Lv 48, HP 1100)
- Schryzon (Lv 78, HP 1100)
-- END --

Press ENTER to continue...
```

Figure 3. Party Menu and Members



```
Inventory Menu:
 1) List items
 2) Add item
 3) Remove item (by name)
 0) Back
> 1

-- Inventory --
* Potion
* Ether
* Megalixir
* Megalixir
* Manifest Illusion
-- END --

Press ENTER to continue...
```

Figure 7. Inventory Menu and List of Items

```
Materia Menu:
 1) List materia
 2) Add materia
 3) Remove materia (by name+grade)
 4) Rotate (simulate advancing slots)
 0) Back
> 4
Start from materia name (leave empty to use first): Stopza
How many steps to simulate? 6
Rotating starting at → Stopza (G9)
Step 0 → Stopza (G9)
Step 1 → Zettaflare (G9)
Step 2 → Fire (G3)
Step 3 → Cure (G2)
Step 4 → Blizzaga (G8)
Step 5 → Stopza (G9)

Press ENTER to continue...
```

Figure 8. Materia Menu and Materia Rotation

```
≡≡ Summary ≡≡
Party size: 3
Inventory size: 5
Materia size: 5

Party:
- Cloud (Lv 50, HP 1200)
- Tifa (Lv 48, HP 1100)
- Schryzon (Lv 78, HP 1100)

Inventory:
* Potion
* Ether
* Megalixir
* Megalixir
* Manifest Illusion

Materia:
~ Fire (G3)
~ Cure (G2)
~ Blizzaga (G8)
~ Stopza (G9)
~ Zettaflare (G9)

Press ENTER to continue...
```

Figure 9. Summary of Every Category