Networking & Design Patterns

Advanced Java Programming

Masoud Hamad

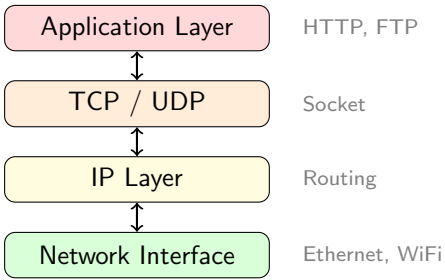
State University of Zanzibar (SUZA)

2025/2026 Academic Year

Outline

Java Networking Overview

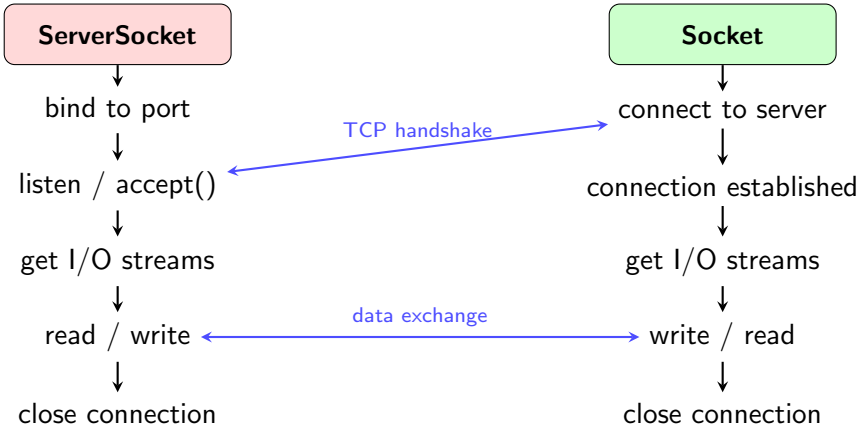
- Java provides comprehensive networking support in `java.net`
- **TCP** (Transmission Control Protocol) — reliable, connection-oriented
- **UDP** (User Datagram Protocol) — fast, connectionless
- Key classes:
  - `URL`, `HttpURLConnection` — HTTP access
  - `Socket`, `ServerSocket` — TCP communication
  - `DatagramSocket` — UDP communication



URL and HttpURLConnection

```
import java.net.*;
import java.io.*;

public class URLDemo {
    public static void main(String[] args) throws Exception {
        URL url = new URL("https://jsonplaceholder.typicode.com/posts/1");

        // URL components
        System.out.println("Protocol: " + url.getProtocol());  // https
        System.out.println("Host: " + url.getHost());           // jsonplaceholder
            ...
        System.out.println("Path: " + url.getPath());           // /posts/1

        // HTTP GET request
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setRequestMethod("GET");
        conn.setRequestProperty("Accept", "application/json");

        int responseCode = conn.getResponseCode();
        System.out.println("Response Code: " + responseCode);

        try (BufferedReader reader = new BufferedReader(
                new InputStreamReader(conn.getInputStream()))) {
```

TCP Client-Server Model



TCP Server — ServerSocket

```
import java.net.*;
import java.io.*;

public class SimpleServer {
    public static void main(String[] args) throws IOException {
        int port = 5000;
        try (ServerSocket serverSocket = new ServerSocket(port)) {
            System.out.println("Server listening on port " + port);

            while (true) {
                Socket clientSocket = serverSocket.accept();  // blocks
                System.out.println("Client connected: "
                    + clientSocket.getInetAddress());

                try (BufferedReader in = new BufferedReader(
                         new InputStreamReader(clientSocket.getInputStream()));
                     PrintWriter out = new PrintWriter(
                         clientSocket.getOutputStream(), true)) {

                    String message = in.readLine();
                    System.out.println("Received: " + message);
                    out.println("Echo: " + message);
                }
```

TCP Client — Socket

```
import java.net.*;
import java.io.*;

public class SimpleClient {
    public static void main(String[] args) throws IOException {
        String host = "localhost";
        int port = 5000;

        try (Socket socket = new Socket(host, port);
             PrintWriter out = new PrintWriter(
                 socket.getOutputStream(), true);
             BufferedReader in = new BufferedReader(
                 new InputStreamReader(socket.getInputStream()));
             BufferedReader console = new BufferedReader(
                 new InputStreamReader(System.in))) {

            System.out.print("Enter message: ");
            String message = console.readLine();
            out.println(message);            // send to server

            String response = in.readLine(); // receive from server
            System.out.println("Server: " + response);
        }
```

Multi-Client Server with Threads

```
public class MultiClientServer {
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = new ServerSocket(5000);
        System.out.println("Server started. Waiting for clients...");
        ExecutorService pool = Executors.newFixedThreadPool(10);

        while (true) {
            Socket client = serverSocket.accept();
            pool.submit(new ClientHandler(client));
        }
    }
}

class ClientHandler implements Runnable {
    private Socket socket;
    public ClientHandler(Socket socket) { this.socket = socket; }

    @Override
    public void run() {
        try (BufferedReader in = new BufferedReader(
                new InputStreamReader(socket.getInputStream()));
             PrintWriter out = new PrintWriter(
                socket.getOutputStream(), true)) {
```

What are Design Patterns?

- **Reusable solutions** to commonly occurring problems in software design
- Not finished code, but *templates* or *blueprints*
- Documented by the "Gang of Four" (GoF) in 1994
- Provide a shared vocabulary for developers
- Improve code maintainability and flexibility

Creational

Singleton, Factory, Builder

Structural

Adapter, Decorator, Facade

Behavioral

Observer, Strategy, Command

Three Categories

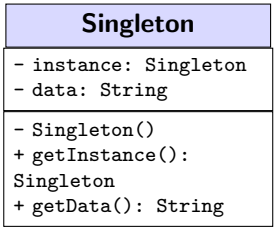
Creational: object creation mechanisms. **Structural**: object composition and relationships. **Behavioral**: object interaction and responsibility.

Singleton Pattern — Intent

Intent: Ensure a class has only **one instance** and provide a global point of access to it.

Use Cases:

- Database connection pool
- Configuration manager
- Logger
- Thread pool
- Cache



Singleton — Eager and Lazy Initialization

```java
// Eager Initialization - instance created at class loading
public class EagerSingleton {
    private static final EagerSingleton INSTANCE = new EagerSingleton();

    private EagerSingleton() {}  // private constructor

    public static EagerSingleton getInstance() {
        return INSTANCE;
    }
}
```

```java
// Lazy Initialization - instance created on first use
public class LazySingleton {
    private static LazySingleton instance;

    private LazySingleton() {}

    public static LazySingleton getInstance() {
        if (instance == null) {          // NOT thread-safe!
            instance = new LazySingleton();
        }
        return instance;
    }
```

Singleton — Thread-Safe Implementations

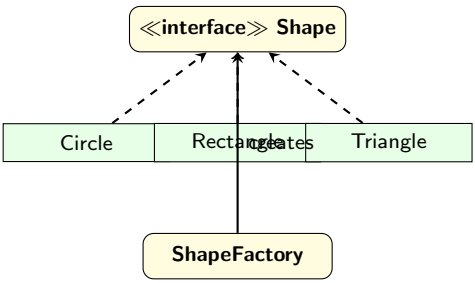
```
// Thread-safe with synchronized (simple but slower)
public class SyncSingleton {
    private static SyncSingleton instance;
    private SyncSingleton() {}

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

// Double-Checked Locking (efficient and thread-safe)
public class DCLSingleton {
    private static volatile DCLSingleton instance;
    private DCLSingleton() {}

    public static DCLSingleton getInstance() {
        if (instance == null) {                    // first check (no lock)
            synchronized (DCLSingleton.class) {
                if (instance == null) {            // second check (with lock)
                    instance = new DCLSingleton();
```

Factory Method Pattern — Intent

Intent: Define an interface for creating objects, but let subclasses decide which class to instantiate.

Benefits:

- Decouples object creation from usage
- Easy to add new types
- Follows Open/Closed Principle



Factory Method — Implementation

```
// Product interface
interface Shape {
    void draw();
    double area();
}

class Circle implements Shape {
    private double radius;
    public Circle(double radius) { this.radius = radius; }
    public void draw() { System.out.println("Drawing Circle"); }
    public double area() { return Math.PI * radius * radius; }
}

class Rectangle implements Shape {
    private double w, h;
    public Rectangle(double w, double h) { this.w = w; this.h = h; }
    public void draw() { System.out.println("Drawing Rectangle"); }
    public double area() { return w * h; }
}

class Triangle implements Shape {
    private double base, height;
    public Triangle(double b, double h) { this.base = b; this.height = h; }
```

Factory Method — Factory Class and Usage

```
public class ShapeFactory {
    public static Shape createShape(String type, double... params) {
        return switch (type.toLowerCase()) {
            case "circle"    -> new Circle(params[0]);
            case "rectangle" -> new Rectangle(params[0], params[1]);
            case "triangle"  -> new Triangle(params[0], params[1]);
            default -> throw new IllegalArgumentException(
                "Unknown shape: " + type);
        };
    }
}
```

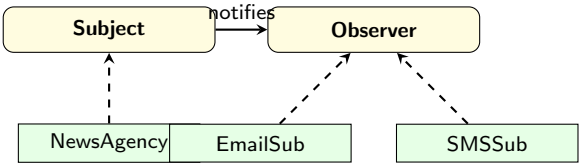
```
public class FactoryDemo {
    public static void main(String[] args) {
        Shape s1 = ShapeFactory.createShape("circle", 5.0);
        Shape s2 = ShapeFactory.createShape("rectangle", 4.0, 6.0);
        Shape s3 = ShapeFactory.createShape("triangle", 3.0, 8.0);

        for (Shape s : List.of(s1, s2, s3)) {
            s.draw();
            System.out.printf("  Area: %.2f%n", s.area());
        }
    }
```

Observer Pattern — Intent

Intent: Define a one-to-many dependency so that when one object (Subject) changes state, all its dependents (Observers) are notified automatically.

Use Cases:

- Event handling (GUI)
- Pub/Sub messaging
- Model-View notification
- Stock price alerts



Observer — Implementation

```java
// Observer interface
interface Observer {
    void update(String news);
}

// Subject interface
interface Subject {
    void subscribe(Observer o);
    void unsubscribe(Observer o);
    void notifyObservers();
}

// Concrete Subject
class NewsAgency implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private String latestNews;

    public void subscribe(Observer o)   { observers.add(o); }
    public void unsubscribe(Observer o) { observers.remove(o); }

    public void notifyObservers() {
        for (Observer o : observers) {
            o.update(latestNews);
```

Observer — Concrete Observers and Usage

```
class EmailSubscriber implements Observer {
    private String name;
    public EmailSubscriber(String name) { this.name = name; }

    @Override
    public void update(String news) {
        System.out.println("[Email to " + name + "]: " + news);
    }
}

class SMSSubscriber implements Observer {
    private String phone;
    public SMSSubscriber(String phone) { this.phone = phone; }

    @Override
    public void update(String news) {
        System.out.println("[SMS to " + phone + "]: " + news);
    }
}
```

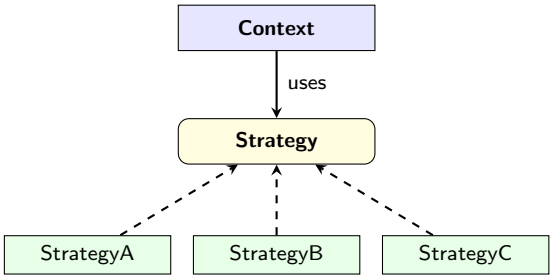
```
NewsAgency agency = new NewsAgency();
Observer email = new EmailSubscriber("Ali");
Observer sms   = new SMSSubscriber("+255-123-456");
```

Strategy Pattern — Intent

Intent: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients.

Use Cases:

- Sorting algorithms
- Payment methods
- Compression algorithms
- Validation strategies



Strategy — Implementation

```
// Strategy interface
interface PaymentStrategy {
    void pay(double amount);
}

// Concrete strategies
class CreditCardPayment implements PaymentStrategy {
    private String cardNumber;
    public CreditCardPayment(String cardNumber) { this.cardNumber = cardNumber; }
    public void pay(double amount) {
        System.out.printf("Paid $%.2f with Credit Card ending %s%n",
            amount, cardNumber.substring(cardNumber.length() - 4));
    }
}

class MobilePayment implements PaymentStrategy {
    private String phoneNumber;
    public MobilePayment(String phone) { this.phoneNumber = phone; }
    public void pay(double amount) {
        System.out.printf("Paid $%.2f via Mobile Money (%s)%n",
            amount, phoneNumber);
    }
}
```

Strategy — Context and Usage

```
// Context class
class ShoppingCart {
    private List<String> items = new ArrayList<>();
    private double total = 0;

    public void addItem(String item, double price) {
        items.add(item);
        total += price;
    }

    public void checkout(PaymentStrategy strategy) {
        System.out.println("Items: " + items);
        strategy.pay(total);  // delegate payment to strategy
    }
}
```

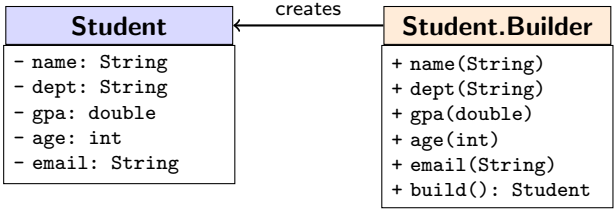
```
ShoppingCart cart = new ShoppingCart();
cart.addItem("Java Book", 45.00);
cart.addItem("USB Cable", 12.50);

// Pay with credit card
cart.checkout(new CreditCardPayment("4532-1234-5678-9012"));
```

Builder Pattern — Intent

Intent: Separate the construction of a complex object from its representation, allowing the same construction process to create different representations.

Use Cases:

- Objects with many optional parameters
- Immutable objects with complex construction
- Fluent API design
- Avoids "telescoping constructor" problem



Builder — Implementation

```
public class Student {
    private final String name;        // required
    private final String department; // required
    private final double gpa;         // optional
    private final int age;            // optional
    private final String email;       // optional

    private Student(Builder builder) {
        this.name = builder.name;
        this.department = builder.department;
        this.gpa = builder.gpa;
        this.age = builder.age;
        this.email = builder.email;
    }

    public static class Builder {
        private final String name;        // required
        private final String department;  // required
        private double gpa = 0.0;
        private int age = 0;
        private String email = "";

        public Builder(String name, String department) {
```

Builder — Fluent Usage

```
public class BuilderDemo {
    public static void main(String[] args) {
        // Full builder with all fields
        Student s1 = new Student.Builder("Ali", "CS")
            .gpa(3.8)
            .age(22)
            .email("ali@suza.ac.tz")
            .build();

        // Minimal builder with only required fields
        Student s2 = new Student.Builder("Fatma", "IT")
            .build();

        // Selective optional fields
        Student s3 = new Student.Builder("Said", "CS")
            .gpa(3.5)
            .email("said@suza.ac.tz")
            .build();

        System.out.println(s1);  // Ali (CS) GPA: 3.8, Age: 22
        System.out.println(s2);  // Fatma (IT) GPA: 0.0, Age: 0
        System.out.println(s3);  // Said (CS) GPA: 3.5, Age: 0
    }
```

Design Patterns Comparison

Pattern	Category	When to Use
Singleton	Creational	Need exactly one instance (config, logger, pool)
Factory	Creational	Object creation logic is complex or type-dependent
Builder	Creational	Complex objects with many optional parameters
Observer	Behavioral	Objects need to react to state changes in another object
Strategy	Behavioral	Multiple algorithms for the same task; chosen at runtime

Important

Design patterns are *guidelines*, not rules. Do not force a pattern where it adds unnecessary complexity. Apply patterns when they solve a real problem in your design.

Summary

Networking

- URL and HttpURLConnection for HTTP
- Socket/ServerSocket for TCP
- Client-Server architecture
- Multi-client servers with thread pools
- Always close connections properly

Design Patterns

- **Singleton**: one instance, global access
- **Factory**: decouple object creation
- **Observer**: event-driven notification
- **Strategy**: interchangeable algorithms
- **Builder**: readable complex construction

Next Steps

Implement a complete client-server chat application combining networking with design patterns (Observer for message broadcast, Strategy for message encryption, Singleton for server instance).