1. **单例模式**

例子：public class Users {  
    // 1. 静态私有实例变量（延迟初始化）  
    private static Users instance = null;  
  
    // 2. 私有构造方法（禁止外部创建实例）  
    private Users() {  
        // 这里可以初始化用户数据  
        System.out.println("Users单例实例被创建");  
    }  
  
    // 3. 公共静态访问方法（全局访问点）  
    public static Users getInstance() {  
        if(instance == null) {  
            instance = new Users();  
        }  
        return instance;  
    }  
  
    // 4. 示例成员方法（实际业务逻辑）  
    public void addUser(String username) {  
        System.out.println("添加用户: " + username);  
    }  
      
    public void printUserCount() {  
        System.out.println("当前用户数量: 0（示例数据）");  
    }  
}

特点：

1、单例设计模式保证一个类只有一个实例;

1. 要提供一个访问该类对象实例的全局访问点。
2. **建造模式**

例子：创建复杂用户对象

public class User {  
    private final String username; // 必选  
    private final String email;    // 必选  
    private final int age;        // 可选  
    private final String phone;    // 可选  
    private final String address; // 可选  
  
    private User(Builder builder) {  
        this.username = builder.username;  
        this.email = builder.email;  
        this.age = builder.age;  
        this.phone = builder.phone;  
        this.address = builder.address;  
    }  
  
    // 建造者类  
    public static class Builder {  
        private final String username; // 必选参数  
        private final String email;    // 必选参数  
        private int age = 0;          // 可选参数（默认值）  
        private String phone = "";     // 可选参数  
        private String address = "";   // 可选参数  
  
        public Builder(String username, String email) {  
            this.username = username;  
            this.email = email;  
        }  
  
        public Builder age(int age) {  
            this.age = age;  
            return this;  
        }  
  
        public Builder phone(String phone) {  
            this.phone = phone;  
            return this;  
        }  
  
        public Builder address(String address) {  
            this.address = address;  
            return this;  
        }  
  
        public User build() {  
            return new User(this);  
        }  
    }  
}

特点：

1、分离构造与表示：将复杂对象的构建过程与其表示分离  
2、分步构建：通过链式调用逐步构造对象  
3、灵活配置：可自由组合可选参数，避免构造器参数爆炸  
4、不可变对象：通常创建不可变对象，线程安全  
5、构建控制：可在build()方法中添加参数校验逻辑

1. **观察者模式**

例子： 用户状态变更通知系统

import java.util.ArrayList;  
import java.util.List;  
  
// 主题接口  
interface UserSubject {  
    void registerObserver(UserObserver o);  
    void removeObserver(UserObserver o);  
    void notifyObservers();  
}  
  
// 具体主题（被观察者）  
class UserProfile implements UserSubject {  
    private String username;  
    private int loginCount;  
    private List<UserObserver> observers = new ArrayList<>();  
  
    public void updateProfile(String username, int loginCount) {  
        this.username = username;  
        this.loginCount = loginCount;  
        notifyObservers(); // 状态变更时通知观察者  
    }  
  
    @Override  
    public void registerObserver(UserObserver o) {  
        observers.add(o);  
    }  
  
    @Override  
    public void removeObserver(UserObserver o) {  
        observers.remove(o);  
    }  
  
    @Override  
    public void notifyObservers() {  
        for (UserObserver observer : observers) {  
            observer.update(username, loginCount);  
        }  
    }  
}  
  
// 观察者接口  
interface UserObserver {  
    void update(String username, int loginCount);  
}  
  
// 具体观察者：日志记录器  
class LoginLogger implements UserObserver {  
    @Override  
    public void update(String username, int loginCount) {  
        System.out.println("[日志] 用户更新: " + username + " 登录次数: " + loginCount);  
    }  
}  
  
// 具体观察者：数据分析模块  
class AnalyticsModule implements UserObserver {  
    @Override  
    public void update(String username, int loginCount) {  
        System.out.println("[分析] 检测到用户行为变化: " + username);  
    }  
}

特点：

1. 松耦合设计：主题和观察者之间抽象耦合  
   2、一对多依赖：一个主题状态变化可通知多个观察者  
   3、事件驱动：基于状态变化的推送通知机制  
   4、开放封闭：易于新增观察者而不修改主题  
   5、广播通信：支持无差别广播通知
2. **中介者模式**

例子：用户聊天室系统

import java.util.ArrayList;  
import java.util.List;  
  
// 中介者接口  
interface ChatMediator {  
    void sendMessage(String msg, User user);  
    void addUser(User user);  
}  
  
// 具体中介者  
class ChatRoom implements ChatMediator {  
    private List<User> users = new ArrayList<>();  
      
    @Override  
    public void sendMessage(String msg, User sender) {  
        for (User user : users) {  
            // 不发送给自己  
            if (user != sender) {  
                user.receive(msg);  
            }  
        }  
    }  
      
    @Override  
    public void addUser(User user) {  
        users.add(user);  
    }  
}  
  
// 抽象同事类  
abstract class User {  
    protected ChatMediator mediator;  
    protected String name;  
      
    public User(ChatMediator med, String name) {  
        this.mediator = med;  
        this.name = name;  
    }  
      
    public abstract void send(String msg);  
    public abstract void receive(String msg);  
}  
  
// 具体同事类  
class ChatUser extends User {  
    public ChatUser(ChatMediator med, String name) {  
        super(med, name);  
    }  
      
    @Override  
    public void send(String msg) {  
        System.out.println(name + " 发送: " + msg);  
        mediator.sendMessage(msg, this);  
    }  
      
    @Override  
    public void receive(String msg) {  
        System.out.println(name + " 收到: " + msg);  
    }  
}

特点：

1、集中控制：通过中介者集中管理对象间交互  
2、解耦同事：对象间不直接通信，降低耦合度  
3、简化协议：将多对多交互转换为一对多  
4、职责分离：交互逻辑与业务逻辑分离  
5、简化维护：交互关系变化只需修改中介者