Подсказки к заданию MiniBank

Подсказки смотрите только после того, как сами **уже решили задачу или попробовали, но не получилось**.

Как работать с application.properties

Описание: Необходимо корректно извлечь значения из `application.properties` для использования в приложении.

Решение: Убедитесь, что в вашем проекте настроен класс с аннотацией `@Configuration` и `@PropertySource`, указывающий на ваш файл свойств. Затем используйте аннотацию `@Value` для внедрения этих значений в поля вашего компонента.

Пример `application.properties`

```
account.default-amount=500
account.transfer-commission=0.02
```

Класс `AccountProperties` для извлечения настроек:

```
@Component
public class AccountProperties {
    private final int defaultAmount;
    private final double transferCommission;

    public AccountProperties(
            @Value("${account.default-amount}") int defaultAmount,
            @Value("${account.transfer-commission}") double
transferCommission
    ) {
        this.defaultAmount = defaultAmount;
        this.transferCommission = transferCommission;
    }

    // геттеры
}
```

В `AppConfig` укажите путь к файлу настроек:

```
@Configuration
@ComponentScan("com.example")
@PropertySource("classpath:application.properties")
public class AppConfig {
}
```

Как сделать обработку всех типов операций не в одном классе

Описание: Для улучшения читаемости и поддержки кода каждый тип операции можно обрабатывать отдельным классом.

Решение: Определите интерфейс `OperationCommand` с методами `execute` и `getOperationType`, который будут реализовывать все обработчики. Используйте `enum` `ConsoleOperationType` для определения типов операций. В классе `ConsoleListener` создайте карту, связывающую каждый тип операции с соответствующим обработчиком.

Пример `enum ConsoleOperationType`:

```
public enum ConsoleOperationType {
    USER_CREATE,
    SHOW_ALL_USERS,
    ACCOUNT_CREATE,
    ACCOUNT_DEPOSIT,
    ACCOUNT_WITHDRAW,
    ACCOUNT_TRANSFER,
    ACCOUNT_CLOSE,
    EXIT
}
```

Пример интерфейса `OperationCommand` и его реализации:

```
public interface OperationCommand {
    void execute();
    ConsoleOperationType getOperationType();
}
```

```
@Component
public class CreateUserCommand implements OperationCommand {

    @Override
    public void execute() {
        // Логика выполнения команды
    }

    @Override
    public ConsoleOperationType getOperationType() {
        return ConsoleOperationType.USER_CREATE;
    }
}
```

В `ConsoleListener` соберите все команды и свяжите их с типами операций:

```
@Component
public class ConsoleListener {
    private final Map<ConsoleOperationType, OperationCommand> commandMap;

    public ConsoleListener(List<OperationCommand> commands) {
        this.commandMap = new HashMap<>();
        commands.forEach(command ->
            commandMap.put(command.getOperationType(), command)
        );
    }

    // Обработка команд
}
```

Это необязательная оптимизация — можно реализовать и через обычный switch. Но паттерн Command сделает код чище, если команд много.

Как сделать перевод с одного счёта на другой

Описание: Необходимо реализовать функцию перевода денег между счетами, учитывая комиссию.

Решение: Метод `transfer` в `AccountService` должен сначала проверить достаточность средств на счёте отправителя, вычесть сумму перевода и

зачислить средства на счёт получателя (с учётом комиссии, если счета принадлежат разным пользователям).

Пример метода `transfer` в `AccountService`:

```
public void transfer(int fromAccountId, int toAccountId, int amount) {
    Account accountFrom = findAccountById(fromAccountId)
            .orElseThrow(() -> new IllegalArgumentException(
                "No such account: id=%s".formatted(fromAccountId)));
    Account accountTo = findAccountById(toAccountId)
            .orElseThrow(() -> new IllegalArgumentException(
                "No such account: id=%s".formatted(toAccountId)));

    if (amount > accountFrom.getMoneyAmount()) {
        throw new IllegalArgumentException(
                "Not enough money to transfer from account: id=%s,
moneyAmount=%s, attemptedTransfer=%s"
                        .formatted(accountFrom.getId(),
accountFrom.getMoneyAmount(), amount)
        );
    }

    // Отправитель теряет полную сумму
    accountFrom.setMoneyAmount(accountFrom.getMoneyAmount() - amount);

    // Если счета принадлежат разным пользователям — применяем комиссию
    int amountToReceive = (accountFrom.getUserId() !=
accountTo.getUserId())
            ? (int) Math.round(amount * (1 -
accountProperties.getTransferCommission()))
            : amount;

    accountTo.setMoneyAmount(accountTo.getMoneyAmount() +
amountToReceive);
}
```

Как реализовать выдачу ID новым пользователям и аккаунтам

Описание: При создании новых пользователей и аккаунтов необходимо автоматически генерировать уникальные ID.

Решение: Используйте счётчик внутри сервисов `UserService` и `AccountService` для генерации ID. Увеличивайте значение счётчика при создании каждого нового

пользователя или аккаунта.

Пример в `UserService`:

```
private int idCounter = 0;

public User createUser(String login) {
    idCounter++;
    User user = new User(idCounter, login, new ArrayList<>());
    // дальнейшая логика
}
```

Аналогично для `AccountService`. Счётчики у `UserService` и `AccountService` независимые — это гарантирует, что каждый новый пользователь и аккаунт получат уникальный идентификатор.