Лекция 4
Разработка АС реального времени (часть 2)

ФИО преподавателя: Зорина Наталья Валентиновна

e-mail: zorina_n@mail.ru

Тема лекции:

«Шаблоны проектирования (Design Patterns)»

Кто созданет и использует шаблоны проектирования?

- В основе [...] лежит идея, что люди должны проектировать свои дома, улицы и сообщества. Эта идея [...] исходит из наблюдения, что большинство чудесных мест в мире были созданы не архитекторами, а людьми — Кристофер Александр и др., «Язык шаблонов»

Шаблоны проектирования

“Повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста”

Аналог идиом в естественном языке.

GoF (Gang of Four): «Приёмы объектно-ориентированного проектирования. Паттерны проектирования» (Design Patterns: Elements of Reusable Object-Oriented Software)

Шаблоны проектирования

Порождающие(Creational)	Структурные (Structural)	Поведенческие(Behavioral)
Abstract factory	Adapter	Chain of responsibility
Builder	Bridge	Command
Factory	Composite	Interpreter
Prototype	**Decorator**	**Iterator**
Singleton	Facade	Mediator
	Flyweight	Memento
	Proxy	**Observer**
		State
		Strategy
		Template
		Visitor

Шаблоны проектирования

Паттерн Singleton гарантирует, что в программе присутствует только один объект данного класса:

```
public class SomeClass {
    public static final SomeClass INSTANCE = new SomeClass();
    private SomeClass() { ... }
    public void someMethod() { ... }
}

SomeClass.INSTANCE.someMethod();
```

Шаблоны проектирования

Само по себе изучение паттернов не очень полезно, особенно для начинающих программистов:

- многие паттерны имеют смысл только в сложных программах
- если исходить при проектировании в первую очередь из паттернов, то программа будет переусложненной
- паттерны более важны не для проектирования, а для обсуждения в команде
- знание паттернов не дает конкретных рекомендаций по проектированию программ
- паттерны были модны в начале 00-х, сейчас уже нет

Шаблоны проектирования

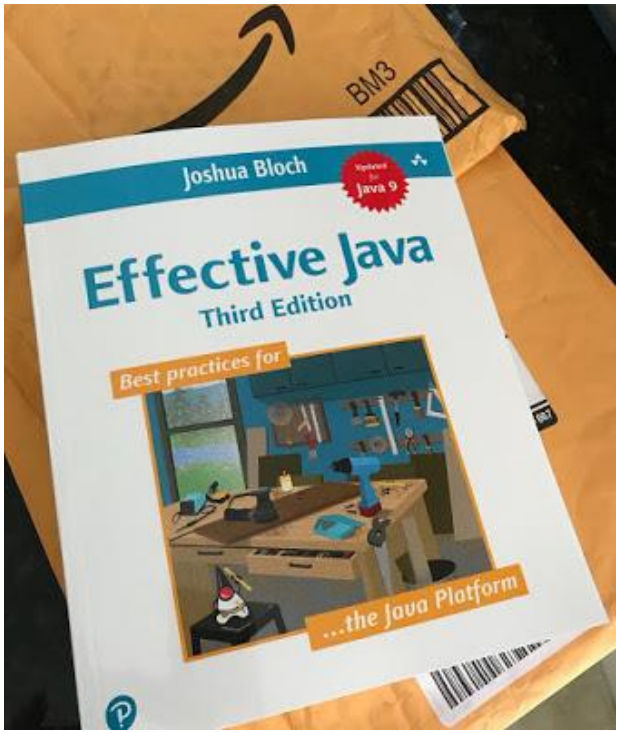
Простые примеры паттернов:

https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries/2707195#2707195

Шаблоны проектирования

Джошуа Блох. Java: эффективное программирование, 3-е издание

Joshua Bloch. Effective Java, 3rd edition



Шаблоны проектирования

Item 1: Consider static factory methods instead of constructors (Джошуа Блох)

```
Boolean trueBoolean = Boolean.valueOf(true);
String number = String.valueOf(12);
List<Integer> list = Arrays.asList(1, 2, 4);
```

2.1. Рассмотрите применение статических фабричных методов вместо конструкторов

```
public class Point {
    public Point(double x, double y) { ... }
    public static Point fromXY(double x, double y) { ... }
}
Point p1 = new Point(1, 2);
Point p2 = Point.fromXY(1, 2);
```

Шаблоны проектирования

Преимущества статических фабричных методов (factory methods):

- говорящее имя, подсказывающее порядок параметров
- может кэшировать объекты (пример: Integer.valueOf)
- метод может вернуть подкласс (пример: Collections.emptyList())

Недостатки:

- если создание объектов доступно только через factory method, создание классов-наследников невозможно (но это может быть даже полезно, см. Item 18)

Шаблоны проектирования

Item 2: Consider a builder when faced with many constructor parameters

2.2. При большом количестве параметров конструктора подумайте о проектном шаблоне Строитель

Шаблоны проектирования

```
public class NutritionFacts {

   private final int servingSize; // (mL) required
   private final int servings; // (per container) required
   private final int calories; // (per serving) optional
   private final int fat; // (g/serving) optional
   private final int sodium; // (mg/serving) optional
   private final int carbohydrate; // (g/serving) optional

   public NutritionFacts(int servingSize, int servings, int calories, int fat, int sodium, int carbohydrate) {
      this.servingSize = servingSize;
      this.servings = servings;
      this.calories = calories;
      this.fat = fat;
      this.sodium = sodium;
      this.carbohydrate = carbohydrate;
   }
}
```

Шаблоны проектирования

```
public class FoodFacts {

   private final int servingSize;
   private final int netWeight;
   private final String packType;
   private int calories;
   private int sodium;
   private int carbohydrate;

   public FoodFacts(int servingSize, int netWeight, String packType) {
      this(servingSize, netWeight, packType,0);
   }
   public FoodFacts(int servingSize, int netWeight, String packType, int calories) {
      this(servingSize, netWeight, packType, calories, 0);
   }
   public FoodFacts(int servingSize, int netWeight, String packType, int calories, int sodium) {
      this(servingSize, netWeight, packType, calories, sodium, 0);

   public FoodFacts(int servingSize, int netWeight, String packType, int calories, int sodium, int carbohydrate) {
      this.servingSize = servingSize;
      this.netWeight = netWeight;
      this.packType = packType;
      this.calories = calories;
      this.sodium = sodium;
      this.carbohydrate = carbohydrate;
   }
}
```

Шаблоны проектирования

```
public class FoodFactBuilder {


    private final int servingSize;
    private final int netWeight;
    private final String packType;
    private final int calories;
    private final int sodium;
    private final int carbohydrate;

private FoodFactBuilder(Builder builder)
{
    servingSize = builder.servingSize;
    netWeight = builder.netWeight;
    packType = builder.packType;
    calories = builder.calories;
    sodium = builder.sodium;
    carbohydrate = builder.carbohydrate;
}

.....
```

Inner class

```
public static class Builder {
    private final int servingSize;
    private final int netWeight;
    private final String packType;
    private int calories = 0;
    private int sodium = 0;
    private int carbohydrate = 0;
    public Builder(int servingSize, int netWeight, String
packType) {
        this.servingSize = servingSize;
        this.netWeight = netWeight;
        this.packType = packType;
    }
    public Builder calories(int val) {
        calories = val;
        return this;
    }
……
public FoodFactBuilder build() {
    return new FoodFactBuilder(this);
}
}
```

Шаблоны проектирования

```
public class Main {
    public static void main(String[] args) {
        FoodFactBuilder pineappleJam = new FoodFactBuilder
                .Builder( servingSize: 240, netWeight: 8,  packType: "Bottle")
                .calories(8)
                .sodium(7)
                .build();
        System.out.println(pineappleJam);
    }
```

def food_fact_builder(servingSize, netWeight, packType, calories=0, sodium=0, carbohydrate=0): pineapple_jam = food_fact(servingSize, netWeight, packType, carbohydrate, calories, sodium) return pineapple_jam

Шаблоны проектирования

Item 5: Prefer dependency injection to hardwiring resources

2.5. Предпочитайте внедрение зависимостей жестко прошитым ресурсам

```
public class SpellChecker {
    private final Lexicon dictionary = new RussianDictionary();
    public boolean isValid(String word) { ... }
    public List<String> suggestions(String typo) { ... }
}
```

Шаблоны проектирования

```
class Cake {
    private CakeBatter cakeBatter;
    private Icing icing;
    private Topping topping;
    //Constructor Injection of cakeBatter in the Cake Object
    Cake(CakeBatter cakeBatter) {
        this.cakeBatter = cakeBatter;
    }

//Setter Injection of the Icing in the Cake Object
void setIcing(Icing icing) {
    this.icing = icing;
}
```

Шаблоны проектирования

```
interface  Topping {
    void addTopping();
}
class FruitTopping implements   Topping {
    @Override
    public void addTopping() {
        System.out.println("Adding fruit toppings.");
    }
}
class ChocolateTopping implements  Topping {
    @Override
    public void addTopping() {
        System.out.println("Adding choco chips topping");
    }
}
```

Шаблоны проектирования

```
class Cake {

   private  CakeBatter cakeBatter;
   private  Icing icing;
   private  Topping topping;

   //Constructor Injection of cakeBatter in the Cake Object
   Cake(CakeBatter cakeBatter) {
      this.cakeBatter = cakeBatter;
   }
   //Setter Injection of the Icing in the Cake Object
   void setIcing(Icing icing) {
      this.icing = icing;
   }
   //Method Injection of the Topping in the Cake Object
   void addToppingToCake(Topping topping) {
      this.topping = topping;
      topping.addTopping();
   }
}
```

Шаблоны проектирования

```
//Adding fruit topping in delegated class
void addFruitTopping(Cake cake) {
    Topping topping = new FruitTopping();
    cake.addToppingToCake(topping);
}

//Adding chocolate topping in delegated class
void addChocolateTopping(Cake cake) {
    Topping topping = new ChocolateTopping();
    cake.addToppingToCake(topping);
}
```

https://engineering.monstar-lab.com/en/post/2019/10/02/simplifying-dependency-injection-and-ioc-concepts-with-typescript/
https://betterprogramming.pub/the-3-types-of-dependency-injection-141b40d2cebc

Шаблоны проектирования

Простое решение – Dependency Injection (DI):

```
public class SpellChecker {
    private final Lexicon dictionary;
    public SpellChecker(Lexicon dictionary) {
        this.dictionary = dictionary;
    }
    public boolean isValid(String word) { ... }
    public List<String> suggestions(String typo) { ... }
}
```

Шаблоны проектирования

Минусы этого кода:

- нельзя поменять на другой словарь
  - в частности, для тестирования

Другой возможный подход – использование Singleton – имеет те же проблемы:

Dictionary dictionary = Dictionary.getDefaultInstance();

так как может быть использован только один словарь на всю программу.

Шаблоны проектирования

При развитии программ очень часто оказывается, что некоторая сущность, которая ранее присутствовала в 1 экземпляре, требуется в нескольких экземплярах.

Поэтому использованию Singleton предпочитайте dependency injection.

Шаблоны проектирования

Item 6: Avoid creating unnecessary objects

2.6. Избегайте создания излишних объектов

- По возможности используйте примитивные типы (int, long, double) вместо классов-оберток (Integer, Long, Double). Например, тип int занимает 4 байта, объект Integer занимает 16 байт в куче плюс 4 байта для указателя на объект, т.е. в 5 раз больше.
- Если нужно создать класс-обертку, используйте метод valueOf вместо вызова конструктора:

```
Boolean b = new Boolean(true);
Boolean b = Boolean.valueOf(true);
Boolean b = true; // вызывает valueOf автоматически
```

Шаблоны проектирования

Метод Boolean.valueOf не создает новый объект, а возвращает либо Boolean.TRUE, либо Boolean.FALSE.

Аналогично Integer.valueOf(x) для x в диапазоне [-128..127] не создает новый объект Integer, а возвращает закэшированный экземпляр (то, что называется Flyweight pattern).

Для собственных классов лучше не использовать кэширование объектов, только если это реально не улучшает производительность. Современные сборщики мусора лучше справляются с б**о**льшим количеством короткоживущих объектов, чем с меньшим количеством долгоживущих объектов.

Шаблоны проектирования

Некоторые методы класса String, работающие с регулярными выражениями, создают экземпляры Pattern. Более эффективно создавать Pattern один раз; вместо

```
static boolean isHexNumeral(String s) {
    return s.matches("[A-Fa-f0-9]+");
}
```

лучше писать:

```
private static final Pattern HEX = Pattern.compile("[A-Fa-f0-9]+");
static boolean isHexNumeral(String s) {
    return HEX.matcher(s).matches();
}
```

Шаблоны проектирования

Item 9: Prefer try-with-resources to try-finally

2.9. Предпочитайте try-с-ресурсами использованию try-finally

```
try (BufferedReader rdr = Files.newBufferedReader(...)) {
   String line = rdr.readLine();
   ...
}
```

Из-за удобства механизма try-with-resources можно создавать свои реализации интерфейса AutoCloseable, если нужно гарантировать возвращение ресурса в некотором блоке кода.

Шаблоны проектирования

```
public class Locked implements AutoCloseable {
    private final Lock lock;
    private Locked(Lock lock) { this.lock = lock; }
    public static Locked lock(Lock lock) {
        lock.lock();
        return new Locked(lock);
    }
    @Override
    public void close() {
        lock.unlock();
    }
}
try (Locked locked = Locked.lock(lock)) {
    ...
}
```

Шаблоны проектирования

Item 10: Obey the general contract when overriding equals

3.1. Перекрывая equals, соблюдайте общий контракт

Контракт метода equals:

1. x.equals(x) всегда true
2. x.equals(y) == y.equals(x)
3. x.equals(y) && y.equals(z) => x.equals(z) (транзитивность)
4. x.equals(y) должно возвращать одно и то же значение для тех же объектов x и y
5. x.equals(null) всегда false

Шаблоны проектирования

Свойства 2 и 3 трудно удовлетворить, если вы используете наследование с переопределением метода equals базового класса (если базовый класс – не Object). Это еще одна причина не использовать наследование (Item 18).

Шаблоны проектирования

Шаблон метода equals:

```
public final class PhoneNumber {
   private final String prefix; private final short areaCode; private final long number;

   @Override
   public boolean equals(Object o) {
      if (o == this)
         return true;
      if (!(o instanceof PhoneNumber))
         return false;
      PhoneNumber that = (PhoneNumber) o;
      return Objects.equals(this.prefix, that.prefix) &&
               this.areaCode == that.areaCode &&
               this.number == that.number;
   }
}
```

Необязательная оптимизация

Шаблоны проектирования

Item 11: Always override hashCode when you override equals

3.2. Всегда при перекрытии equals перекрывайте hashCode

Контракт метода hashCode:

1. x.hashCode() должен возвращать всегда одно и то же значение
2. x.equals(y) => x.hashCode() == y.hashCode()

Так как в хэш-таблицах методы equals и hashCode работают в паре, в классах они тоже должны объявляться в паре.

Шаблоны проектирования

Шаблон метода hashCode:

```
public final class PhoneNumber {
    private final String prefix;
    private final short areaCode;
    private final long number;
    @Override
    public int hashCode() {
        int result = Objects.hashCode(prefix);
        result = result * 31 + Short.hashCode(areaCode);
        result = result * 31 + Long.hashCode(number);
        return result;
    }
}
```

Шаблоны проектирования

Item 15: Minimize the accessibility of classes and members

4.1. Минимизируйте доступность классов и членов

Иерархия модификаторов доступа:

public > protected > package > private

Классам и их членам нужно давать наименее возможный уровень доступа. Для классов – не public, для членов – private. Увеличивать доступность нужно только при необходимости.

Шаблоны проектирования

Минимизация доступа классов и членов упрощает развитие программы. Например, если метод является public, то его могут вызывать из любого другого класса. Поэтому при необходимости изменения этого метода (например, добавления параметра) изменения также могут захватывать всю остальную программу. Если же метод является package, то изменения могут захватывать только классы, находящиеся в том же пакете.

Шаблоны проектирования

Это правило более важно для приложений, чем для библиотек. Для библиотек иногда требуется модифицировать поведение классов библиотеки не предусмотренным создателем библиотеки способом, поэтому лучше оставить эту возможность, делая методы protected вместо package private.

Шаблоны проектирования

Item 17: Minimize mutability

4.3. Минимизируйте изменяемость

По возможности делайте объекты неизменяемыми:

- все поля final
- от класса невозможно унаследоваться (final class)

Шаблоны проектирования

Даже при невозможности сделать класс полностью неизменяемым, изменяемую часть нужно делать как можно меньше. Чем больше полей с модификатором final, тем проще понять поведение класса:

- это уменьшает количество “движущихся частей”
- это гарантирует корректную инициализацию полей

Шаблоны проектирования

Item 18: Favor composition over inheritance

4.4. Предпочитайте композицию наследованию

```
class A {
    void f() { ... }
}
```

Наследование	Композиция

`class B extends A {`
`  void g() { ... }`
`}`
	`class B {`
	`  private A a;`
	`  void f() { a.f(); }`
	`  void g() { ... }`
	`}`

Шаблоны проектирования

Речь идет о наследовании реализации (т.е. конкретных методов); реализация абстрактных методов и наследование интерфейсов как правило не создает проблем.

Проблемы наследования:

- возможно extend только один класс
- слишком сильная связь с конкретным классом; в случае композиции мы можем использовать любой класс-наследник A
- слишком глубокие иерархии наследования трудно понимать
- затруднено переиспользование кода из глубины иерархии
- наследование может нарушать инкапсуляцию

Шаблоны проектирования

Затруднено переиспользование кода из глубины иерархии:

Иерархия 1	Иерархия 2
class A1 { ... } class B1 extends A1 { ... } class C1 extends B1 { ... } class D1 extends C1 { ... }	class A2 { ... } class B2 extends A2 { ... } class C2 extends B2 { ... }

Предположим, нам понадобилась функциональность класса D1 в иерархии 2. Мы не можем просто добавить его в эту иерархию, так как он уже наследуется от других классов.

При использовании композиции и замены конкретных классов на интерфейсы такие проблемы решаются достаточно легко.

Шаблоны проектирования

Наследование может нарушать инкапсуляцию:

```
public class InstrumentedHashSet<E> extends HashSet<E> {
   // The number of attempted element insertions
   private int addCount = 0;
   @Override public boolean add(E e) {
      addCount++;
      return super.add(e);
   }
   @Override public boolean addAll(Collection<? extends E> c) {
      addCount += c.size();
      return super.addAll(c);
   }
   public int getAddCount() {
      return addCount;
   }
}
```

Шаблоны проектирования

```
InstrumentedHashSet<String> s = new InstrumentedHashSet<>();
s.addAll(List.of("Snap", "Crackle", "Pop"));
```

Если реализация метода HashSet.addAll вызывает add для каждого элемента, то мы посчитаем каждый элемент 2 раза: один раз при вызове addAll, и второй раз при вызове add.

Но метод HashSet.addAll может быть реализован и по-другому, это деталь реализации.

Получается, что унаследованный класс работает по-разному в зависимости от деталей реализации метода addAll. Если бы метод addAll был задокументирован, как вызывающий add для каждого элемента, можно было переопределить только add. Но такая реализация addAll неэффективна для ArrayList.

Шаблоны проектирования

Решение – использовать композицию (паттерн Decorator):

```
public class InstrumentedSet<E> implements Set<E> {
    private final Set<E> s;
    private int addCount = 0;
    public InstrumentedSet(Set<E> s) { this.s = s; }
    public boolean add(E e) {
        addCount++;
        return s.add(e);
    }
    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return s.addAll(c);
    }
    // Все методы Set, включая equals, hashCode и toString, делегируются s
}
```

Шаблоны проектирования

Item 24: Favor static member classes over nonstatic

4.10. Предпочитайте статические классы-члены нестатическим

```
public class Outer {
    public static class Inner {
    }
}
```

Это уменьшает потребление памяти и устраняет возможные утечки памяти за счет отсутствия ссылки Inner на объект Outer.

Шаблоны проектирования

Item 25: Limit source files to a single top-level class

4.11. Ограничивайтесь одним классом верхнего уровня на исходный файл

Java запрещает несколько public классов в одном файле, но разрешает иметь дополнительные не-public классы в файле. Тем не менее так делать не следует. Компилятор ищет исходные файлы по имени класса, и если вы не придерживаетесь однозначного соответствия класс↔файл, то компилятор может найти не тот класс, который вы ожидаете.

Шаблоны проектирования

Item 26: Don’t use raw types

5.1. Не используйте несформированные типы

Item 27: Eliminate unchecked warnings

5.2. Устраняйте предупреждения о непроверяемом коде

Raw type: если у класса есть т**и**повый параметр, то использование этого класса без указания параметра – это raw type.

```
List list = new ArrayList();
List<String> list = new ArrayList();
List<String> list = new ArrayList<>();
```

Шаблоны проектирования

Generics предназначены для предотвращения ClassCastException; если вы используете raw types, вы теряете эту возможность.

Наличие unchecked warning говорит о том, что в программе потенциально могут возникать ClassCastException.

Шаблоны проектирования

Item 28: Prefer lists to arrays

5.3. Предпочитайте списки массивам

```
public User[] getUsers();
public List<User> getUsers();
```

Шаблоны проектирования

Списки более универсальны, чем массивы:

- в списки можно добавлять/удалять значения
- массив можно преобразовать в список через Arrays.asList без копирования данных; при вызове list.toArray данные массива копируются
- пустой список Collections.emptyList() не выделяет память, в отличие от “new Type[0]”
- если нужно запретить изменение списка, можно использовать Collections.unmodifiableList; массив всегда изменяемый
- можно создать “new ArrayList<T>()”, но нельзя “new T[n]”, где T – generic тип

Шаблоны проектирования

Item 49: Check parameters for validity

8.1. Проверяйте корректность параметров

```
public class User {
    private final String name;
    public User(String name) {
        this.name = Objects.requireNonNull(name, “name is null”);
    }
}
```

Шаблоны проектирования

Для проверки индексов есть методы (Java 9+):

int Objects.checkFromIndexSize(fromIndex, size, length):

проверка [fromIndex, fromIndex + size) ⊆ [0, length)

int Objects.checkFromToIndex(fromIndex, toIndex, length):

проверка [fromIndex, toIndex) ⊆ [0, length)

int Objects.checkIndex(index, length):

проверка index ∈ [0, length)

Шаблоны проектирования

Для произвольных проверок при нарушении условия можно выбрасывать IllegalArgumentException:

```
public Map<String, String> mapOf(String... values) {
    if (values.length % 2 != 0)
        throw new IllegalArgumentException("Должно быть четное число аргументов");
    Map<String, String> result = new HashMap<>(values.length / 2);
    for (int i = 0; i < values.length; i += 2) {
        result.put(values[i], values[i + 1]);
    }
    return result;
}
```

Шаблоны проектирования

Item 50: Make defensive copies when needed

8.2. При необходимости создавайте защитные копии

Попытка создать неизменяемый класс, описывающий логин и пароль пользователя:

```
public class LoginInfo {
    private final String login;
    private final byte[] passwordHash;
    public LoginInfo(String login, byte[] passwordHash) {
        this.login = login;
        this.passwordHash = passwordHash;
    }
}
```

Шаблоны проектирования

```
byte[] hash1 = {1, 2, 3};
LoginInfo info = new LoginInfo(“test_login”, hash1);
hash1[0] = 10; // мы изменили состояние объекта info!
```

Меры защиты:

```
public LoginInfo(String login, byte[] passwordHash) {
    this.login = login;
    this.passwordHash = Arrays.copyOf(passwordHash);
}
```

Создание defensive copy предотвращает неожиданное изменение данных внешним кодом.

Шаблоны проектирования

```
public class LoginInfo {
    private final String login;
    private final byte[] passwordHash;
    public byte[] getPasswordHash() {
        return passwordHash;
    }
}
LoginInfo info = new LoginInfo("test_login", new byte[] {1, 2, 3});
byte[] hash1 = info.getPasswordHash();
hash1[0] = 10; // мы изменили состояние объекта info!
```

Шаблоны проектирования

Меры защиты:

```
public byte[] getPasswordHash() {
    return Arrays.copyOf(passwordHash);
}
```

T.e. создание defensive copies должно быть как на входе (в конструкторе/setter'е), так и на выходе (getter'е), если мы используем поле изменяемого класса, на неизменяемость которого мы рассчитываем.

Для коллекций в getter'ах вместо полного копирования можно использовать методы Collections.unmodifiableXXX, которые гарантируют то, что при вызовах методов коллекций она не будет изменена. При этом в конструкторе полная копия все равно требуется.

Шаблоны проектирования

Item 52: Use overloading judiciously

8.4. Перегружайте методы разумно

Если у вас есть несколько методов с одинаковым именем, можно случайно вызвать не тот метод:

```
void write(int i);
void write(byte b);

short s = 0;
write(s); // какой метод вызывается?
```

Шаблоны проектирования

Иногда, если методы имеют одно название, их трудно вызывать:

```
void execute(Function<String, String> f);

void execute(Consumer<String> c);

// ошибка компиляции, непонятно, какой из методов вызывать:

execute(str -> str.toLowerCase());
```

Если перегружаемые методы имеют разное количество параметров, то таких проблем не возникает. Но если методов с одним именем много, можно случайно вызвать не тот.

Шаблоны проектирования

Item 54: Return empty collections or arrays, not nulls

8.6. Возвращайте пустые массивы и коллекции, а не null

```
public List<User> findUsers(String firstName);
```

Если пользователи не найдены, возвращайте пустой список, а не null. Это избавит от необходимости постоянной проверки на null:

```
List<User> found = findUsers(...);
if (found != null && !found.isEmpty()) {
    ...
```

Шаблоны проектирования

Item 57: Minimize the scope of local variables

9.1. Минимизируйте область видимости локальных переменных

Вместо

```
int i;
...
i = 10;
```

лучше использовать

```
int i = 10;
```

Шаблоны проектирования

Объявлять переменную желательно непосредственно перед ее использованием.

Если переменная используется только внутри блока, она должна быть объявлена внутри блока:

```
int x;
if (...) {
    x = 10;
}
if (...) {
    int x = 10;
}
```

Шаблоны проектирования

Следует предпочитать цикл for циклу while:

```
Iterator<Element> i = c.iterator();
while (i.hasNext()) {
    doSomething(i.next());
}
...
Iterator<Element> i2 = c2.iterator();
while (i.hasNext()) { // BUG!
    doSomethingElse(i2.next());
}
```

Шаблоны проектирования

Следует предпочитать цикл for циклу while:

```
for (Iterator<Element> i = c.iterator(); i.hasNext(); ) {
    doSomething(i.next());
}
...
for (Iterator<Element> i2 = c2.iterator(); i.hasNext(); ) { // ошибка компиляции
    doSomethingElse(i2.next());
}
```

Шаблоны проектирования

Item 58: Prefer for-each loops to traditional for loops

9.2. Предпочитайте циклы for для коллекции традиционным циклам for

```
List<User> users = ...;
for (int i = 0; i < users.size(); i++) {
    User user = users.get(i);
    ...
}
for (User user : users) {
    ...
}
```

Шаблоны проектирования

Item 61: Prefer primitive types to boxed primitives

9.5. Предпочитайте примитивные типы упакованным примитивным типам

```
Integer sum = 0;
for (int elem : array) {
    sum += elem;
}
```

Лучше использовать “int sum = 0;”

Шаблоны проектирования

Item 62: Avoid strings where other types are more appropriate

9.6. Избегайте применения строк там, где уместнее другой тип

Если ключом какого-либо объекта в хэш-таблице является пара целых чисел, то может возникнуть соблазн писать так:

```
int i = 3;
int j = 4;
String key = i + “/” + j;
Map<String, MyObject> map = new HashMap<>();
map.put(key, myObject);
```

Шаблоны проектирования

Это приводит к stringly-typed programming. В таком Map могут быть не только пары чисел, но и произвольные строки.

Новый класс создать никогда не вредно, особенно в Java 16 с records:

```
public record IntPair(int i, int j);

Map<IntPair, MyObject> map = new HashMap<>();

map.put(new IntPair(i, j), myObject);
```

Не бойтесь создавать новые классы при малейшей в них потребности. Переиспользование готовых классов (типа String) создаст проблемы при дальнейшем развитии программы.

Шаблоны проектирования

Item 63: Beware the performance of string concatenation

9.7. Помните о проблемах производительности при конкатенации строк

```
String str = “”;
for (int i = 0; i < 1_000_000; i++) {
    str += “X”;
}
StringBuilder buf = new StringBuilder();
for (int i = 0; i < 1_000_000; i++) {
    buf.append(“X”);
}
String str = buf.toString();
```

Шаблоны проектирования

str += “X”;

эквивалентно

str = str + “X”;

На i-ой итерации длина str – i, операция str + “X” создает **новую** строку длиной i+1, копируя в нее i символов из исходной строки плюс 1 символ “X”, т.е. на i-ом шаге мы совершаем i+1 операций. Итого за N итераций получаем для i=[0..1_000_000):

$1+2+...+N = N*(N+1)/2 = O(N^2)$

Для $N=10^6$ это будет 10^{12} – очень много.

В отличие от этого StringBuilder добавляет “X” к уже существующему объекту, копируя только 1 символ “X”, всего O(N).

Шаблоны проектирования

Item 69: Use exceptions only for exceptional conditions

10.1. Используйте исключения только в исключительных ситуациях

```
// Horrible abuse of exceptions. Don't ever do this!
try {
    int i = 0;
    while (true)
        range[i++].climb();
} catch (ArrayIndexOutOfBoundsException e) {
}
```

Шаблоны проектирования

```
for (Moutain m : range) {
    m.climb();
}
```

Исключения не следует использовать для управления последовательностью выполнения программы. Исключения – только для исключительных ситуаций.

- исключения работают медленнее, чем if/while/for
- можно поймать не то исключение, которое мы ждем
- код с if/while/for является идиоматичным и легче читается

Шаблоны проектирования

Item 70: Use checked exceptions for recoverable conditions and runtime exceptions for programming errors

10.2. Используйте для восстановления проверяемые исключения, а для программных ошибок — исключения времени выполнения

В правильно написанной программе не должно возникать исключений типа NullPointerException, IndexOutOfBoundsException, ClassCastException, NoSuchElementException. Все эти исключения – результат некорректного кода. Если они возникают, то код программы нужно переписать, ловить их никогда не нужно.

Шаблоны проектирования

В отличие от RuntimeException, проверяемые исключения (checked exceptions) сигнализируют не об ошибке в коде, а о наступлении проблемы, которую можно исправить.

Например, IOException может сигнализировать о том, что указано неверное имя файла, и нужно выбрать корректный файл.

Шаблоны проектирования

Item 77: Don’t ignore exceptions

10.9. Не игнорируйте исключения

```
try {
    ...
} catch (Exception ex) {
    // Как минимум, исключение нужно записать в протокол
}
```

Шаблоны проектирования

Item 82: Document thread safety

11.5. Документируйте безопасность с точки зрения потоков