lichMax

40 уровень

Санкт-Петербург

19.07.2017 9157 11

Уровень 37. Ответы на вопросы к собеседованию по теме уровня

Здравствуйте. Опять-таки, не нашёл ответов и на эти вопросы. Решил выложить ответы, которые я составил для себя.

Вот, собственно вопросы:

Вопросы к собеседованию:

1. Что такое паттерны проектирования?
2. Какие паттерны проектирования вы знаете?
3. Расскажите про паттерн Singleton? Как сделать его потокобезопасным?
4. Расскажите про паттерн Factory?
5. Расскажите про паттерн AbstractFactory
6. Расскажите про паттерн Adaper, его отличия от Wrapper?
7. Расскажите про паттерн Proxy
8. Что такое итератор? Какие интерфейсы, связанные с итератором, вы знаете?
9. Зачем нужен класс Arrays?
10. Зачем нужен класс Collections?

А вот мои ответы:

Мои ответы:

1. Паттерны проектирования — это устоявшиеся удачные решения самых распространённых проблем, возникающих при проектировании и разработке программ или их частей.

2. `Singleton`, `Factory`, `Abstract Factory`, `Template method`, `Strategy`, `Pool`, `Adapter`, `Proxy`, `Bridge`, `MVC`.

3. Когда нужно, чтобы в программе существовал только один экземпляр какого-то класса, то применяют паттерн `Singleton`. Он выглядит так (lazy initialization):

```
clas Singleton {
    private Singleton instance;

    private Singleton() {}

    public static Singletot getInstance() {
        if (instance == null)
            instance = new Singleton();
        return instance;
    }
}
```

Чтобы сделать его потокобезопасным, можно добавить к методу `getInstance()` модификатор `synchronized`. Но это будет не самым лучшим решением (зато самым простым). Гораздно лучшее решение — это написать метод `getInstance` таким образом (double-checked locking):

```
public static synchronized Singleton getInstance() {
    if (instance == null)
        synchronized(Singleton.class) {
            instance = new Singleton();
        }
    return instance;
}
```

4. Паттерн `Factory` — это порождающий паттерн. Он позволяет создавать объекты по требованию (например, при определённых условиях). Выглядит это так:

```
class Factory{
    public static Object1 getObject1() {
        return new Object1();
    }

    public static Object2 getObject2() {
        return new Object2();
    }

    public static Object3 getObject3() {
        return new Object3();
    }
}
```

Также существует разновидность этого паттерна под названием `FactoryMethod`. Согласно этому паттерну, в одном методе создаются разные объекты, в завимости от поступающих входных данных (значений параметров). Все эти объекты должны иметь общего предка (или один общий реализуемый интерфейс). Выглядит он так:

```
        TYPE1,
        TYPE2,
        TYPE3
    }

    public static CommonClass getObject(TypeObject type) {
        switch(type) {
            case TYPE1:
                return new Object1();
            case TYPE2:
                return new Object2();
            case TYPE3:
                return new Object3();
            default:
                return null;
        }
    }
}
```

Классы `Object1`, `Object2` и `Object3` наследуются от класса `CommonClass`.

5. Паттерн `Abstract Factory` — это также порождающий шаблон проектирования. Согласно этому паттерну, создаётся некоторая абстрактная фабрика, служащая шаблоном для нескольких конкретных фабрик. Можно привести такой пример:

```
class Human {}

class Boy extends Human {}
class TeenBoy extends Human {}
class Man extends Human {}
class OldMan extends Human {}

class Girl extends Human {}
class TeenGirl extends Human {}
class Woman extends Human {}
class OldWoman extends Human {}

interface AbstractFactory {
    Human getPerson(int age);
}

class FactoryMale implements AbstractFactory {
    public Human getPerson(int age) {
        if (age < 12)
            return new Boy();
        if (age >= 12 && age <= 20)
            return new TeenBoy();
        if (age > 20 && age < 60)
            return new Man();
        return new OldMan();
    }
}

class FactoryFemale implements AbstractFactory {
    public Human getPerson(int age) {
```

```
        if (age >= 12 && age <= 20)
            return new TeenGirl();
        if (age > 20 && age < 60)
            return new Woman();
        return new OldWoman();
    }
}
```

6. Паттерн `Adapter` — это структурный паттерн. Его реализация позволяет использовать объект одного типа там, где требуется объект другого типа (обычно это абстрактные типы). Пример реализации этого паттерна:

```
interface TotalTime {
    int getTotalSeconds();
}
interface Time {
    int getHours();
    int getMinutes();
    int getSeconds();
}

class TimeAdapter extends TotalTime {
    private Time time;
    public TimeAdapter(Time time) {
        this.time = time;
    }
    public int getTotalTime() {
        return time.getSeconds + time.getMinutes * 60 + time.getHours * 60 * 60;
    }
}

class TotalTimeAdapter extends Time {
    private TotalTime totalTime;
    public TotalTimeAdapter(TotalTime totalTime) {
        this.totalTime = totalTime;
    }

    public int getSeconds() {
        return totalTime % 60;
    }

    public int getMinutes() {
        return (totalTime / 60) % 60;
    }

    public int getHours() {
        return totaltime/ (60 * 60) ;
    }
}

class Main {
    public static void main(String[] args) {
        Time time = new Time() {
            public int getSeconds() {
```

```
45
46              public int getMinutes() {
47                  return LocalTime.now().getMinute();
48              }
49
50              public int getHours() {
51                  return LocalTime.now().getHour() ;
52              }
53          };
54
55          TotalTime totalTime = new TimeAdapter(time);
56          System.out.println(totalTime.getTotalSeconds());
57
58          TotalTime totalTime2 = new TotalTime() {
59              public int getTotalSeconds() {
60                  LocalTime currTime = LocalTime.now();
61                  return currTime.getSecond() + currTime.getMinute * 60 + currTime.getHour * 60 * 6
62              }
63          };
64
65          Time time2 = new TotalTimeAdapter(totalTime2);
66          System.out.println(time2.getHours + ":" + time2.getMinutes() + ":" + time2.getSeconds());
67      }
68  }
```

При реализации паттерна `Wrapper` создаётся класс, который оборачивает исходный класс и реализует тот же интерфейс, который реализует исходный класс. Таким образом, это позволяет расширить функциональность исходного класса и использовать новый класс там, где ожидается использование исходного класса. Это отличается от реализации паттерна `Adapter` тем, что в данном случае используется один интерфейс (тот же, что есть у исходного класса). В паттерне `Adapter` же используется два интерфейса, и класс, который оборачивает экземпяр исходного класса, реализует совсем другой инферфейс, не интерфейс исходного класса.

7. Паттерн `Proxy` — это структурный паттерн проектирования. Он нужен для того, чтобы контролировать доступ к какому-то объекту. Для этого пишется класс по типу "обёртка", то есть внутрь класса передаётся исходный объект, реализующий некий интерфейс, сам класс тоже реализует этот интерфейс, и в каждом методе этого класса вызывается похожий метод у исходного объекта. Реализация того же интерфейса, что и у исходного объекта, позволяет подменить исходный объект прокси-объектом. Также это позволяет, не меняя исходного объекта, "навешивать" на его методы какую-то специальную дополнительную функциональность (например, логирование, проверка прав доступа, кэширование и т.д.). Пример:

```
1   interface Bank {
2       void setUserMoney(User user, double money);
3       double getUserMoney(User user);
4   }
5
6   class CitiBank implements Bank { //оригинальный класс
7       public void setUserMoney(User user, double money) {
8           UserDAO.update(user,money);
9       }
10
11      public double getUserMoney(User user) {
12          UserDAO.getUserMoney(user);
13      }
14  }
```

```
17      private Bank bank;
18
19      public SecurityProxyBank(Bank bank) {
20          this.bank = bank;
21      }
22
23      public void setUserMoney(User user, double money) {
24          if (!SecurityManager.authorize(user,BankAccounts.Manager)
25              throw new SecurityException("User can't change money value");
26
27          UserDAO.update(user,money);
28      }
29
30      public double getUserMoney(User user) {
31          if (!SecurityManager.authorize(user,BankAccounts.Manager)
32              throw new SecurityException("User can't get money value");
33
34          UserDAO.getUserMoney(user);
35      }
```

8. Итератор — это специальный внутренний объект коллекции, который позволяет последовательно перебирать элементы этой коллекций. Этот объект должен реализовывать интерфейс `Iterator<E>`, либо `ListIterator<E>` (для списков). Также, для того, чтобы перебирать элементы коллекции, коллекция должна поддерживать интерфейс `Iterable<E>`. Интерфейс `Iterable<E>` содержит всего один метод — `iterator()`, который позволяет извне получить доступ к итератору коллекции.

   Интерфейс `Iterator<E>` содержит следующие методы:

   - `boolean hasNext()` — проверяет, есть ли в коллекции ещё какой-то элемент
   - `E next()` — позволяет получить очередной элемент коллекции (после получения элемента, внутренний курсор итератора передвигается на следующий элемент коллекции)
   - `void remove()` — удаляет текущий элемент из коллекции

   Интерфейс же `ListIterator<E>` содержит такие методы:

   - `boolean hasNext()` — проверяет, существуют ли ещё один элемент в коллекции (следующий за текущим)
   - `E next()` — возвращает очередной элемент коллекции (и передвигает внутренний курсок итератора на следующий элемент)
   - `int nextIndex()` — возвращает индекс следующего элемента
   - `void set(E e)` — устанавливает значение текущего элемента `void add(E e)`. Добавляет элемент в конец списка.
   - `boolean hasPrevious()` — проверяет, существует ли какой-то элемент в коллекции перед данным элементом
   - `E previous()` — возвращает текущий элемент коллекции и переводит курсор на предыдущий элемент коллекции
   - `int previousIndex` — возвращает индекс предыдущего элемента коллекции
   - `void remove()` — удаляет текущий элемент коллекции
   - `void add(E e)` — добавляет элемент e после текущего элемента коллекции

9. Класс `Arrays` — это утилитарный класс, предназначенный для разнообразных манипуляций с массивами. В этом классе есть методы превращения массива в список, поиска по массиву, копирования массива, сравнения массивов, получения хешкода массива, представление массива в виде строки и др.

10. Класс `Collections` — это утилитарный класс для работы с коллекциями. В этом классе есть методы добавления

модификаций коллекций известных типов (например, можно получить потокобезопасную коллекции или неизменяемую коллекцию с одним элементом).

+33

Комментарии (11) популярные новые старые

JavaCoder

Введите текст комментария

Игорь Full Stack Developer в **IgorApplications** 10 августа 2021, 21:36

Фабричный метод это не разновидность Фабрики, это просто abstract метод который может иметь разную реализацию в подклассах, которая будет вызываться у супер класса.

Ответить 0

Urcont Уровень 35, Russian Federation 16 августа, 22:46

Это точно не про Template Method написано?

Ответить 0

Марат Уровень 41, Москва, Россия 26 мая 2021, 17:22

3. Singleton

```
1  public class Singleton {
2
3      private Singleton() {
4      }
5
6      private static class SingletonHolder {
7          public static final Singleton HOLDER_INSTANCE = new Singleton();
8      }
9
10     public static Singleton getInstance() {
11         return SingletonHolder.HOLDER_INSTANCE;
12     }
13 }
```

Ответить +1

Даниил Salesforce Developer в **Viseven** MASTER 4 октября 2019, 14:25

Вопрос 4. Я ещё сам толком не разобрался, но вроде бы у вас описано не правильно. Тут описана разница между разными понятиями Фабрики. На самом деле самый сложный для понимания паттерн как по мне...

Ответить 0

Денис Дворецкий Уровень 38, Минск, Беларусь 10 февраля 2019, 13:40

Измените пример 3 с Singleton.
Переменная private Singleton instance должна быть еще и static
У вас в примере с doubleCheck не убрана синхронизация метода:
public static synchronized Singleton getInstance()
Таким образом теряется всякий смысл внутренней двойной проверки.
И при наличии любых полей АНТИпаттерн с Double-Checked Locking в таком виде работать не будет.
надо так :
private static volatile Singleton instance;

Ответить +6

Vitaly Khan Java Developer в **Onollo** MASTER 3 января 2020, 09:37

именно. в 3 вопросе сразу две ошибки. до сих пор не исправлены.

Ответить +1

Сергей Уровень 38 20 января 2021, 23:18

Ответить

0

MrArtemAA Уровень 40, Москва, Россия

14 декабря 2017, 15:00

Спасибо за материал.

В качестве дополнения:

Вопрос 4. Как такового Factory нет (см Design Patterns, GoF) — это либо AbstractFactory, либо Factory Method (которым, как отмечено ниже, обычно порождает объекты AbstractFactory).

Вопрос 6. По сути все верно. Несколько смущает, что Wrapper иногда называют синонимом Adapter, а иногда Decorator. Согласно Design Patterns (GoF) Wrapper — это Decorator.

Ответить

+3

Павел Х. Team Lead в **Иннотех** EXPERT

19 декабря 2018, 18:47

В лекциях как раз говорили, что Adapter и Wrapper - одно и то же. Думаю, что GoF - более авторитетный источник. Но из того, что я вижу в различных обсуждениях этого вопроса - паттерна Wrapper не существует, это обобщённое название Адаптера и Декоратора.

Ответить

0

NodeOne Уровень 41 EXPERT

24 февраля 2019, 14:24

Даже задачи у Adapter и Wrapper совершенно разные. Это совершенно не одно и тоже. Wrapper — это Decorator.

Ответить

0

Vitaly Khan Java Developer в **Onollo** MASTER

3 января 2020, 09:51

Wrapper - это скорее обобщающий термин, под него подходят и Адаптер, и Декоратор. на сайте refactoring.guru это тоже видно: здесь и здесь (смотрите рядом с заголовком: Also known as Wrapper в обоих случаях)

Ответить

+1