



## Управляемые коллекции

Я всегда  
тщательно  
инкапсулирую свои  
коллекции!



**Существует много способов создания коллекций.** Объекты можно разместить в контейнере Array, Stack, List, HashMap — выбирайте сами. Каждый способ обладает своими достоинствами и недостатками. Но в какой-то момент клиенту потребуется перебрать все эти объекты, и когда это произойдет, собираетесь ли вы раскрывать реализацию коллекции? Надеемся, нет! Это было бы крайне непрофессионально. В этой главе вы узнаете, как предоставить клиенту механизм перебора объектов без раскрытия информации о способе их хранения. Также в ней будут описаны способы создания суперколлекций. А если этого недостаточно, вы узнаете кое-что новое относительно обязанностей объектов.

## Сенсация в Объектвиле: бистро объединяется с блинной!

Отличные новости! Теперь мы можем заказать аппетитный завтрак с блинчиками и обед в одном месте! Но похоже, у поваров возникла небольшая проблема...

Они хотят использовать меню моей блинной для завтраков, а меню бистро — для обедов. Мы согласовали реализацию элементов меню...

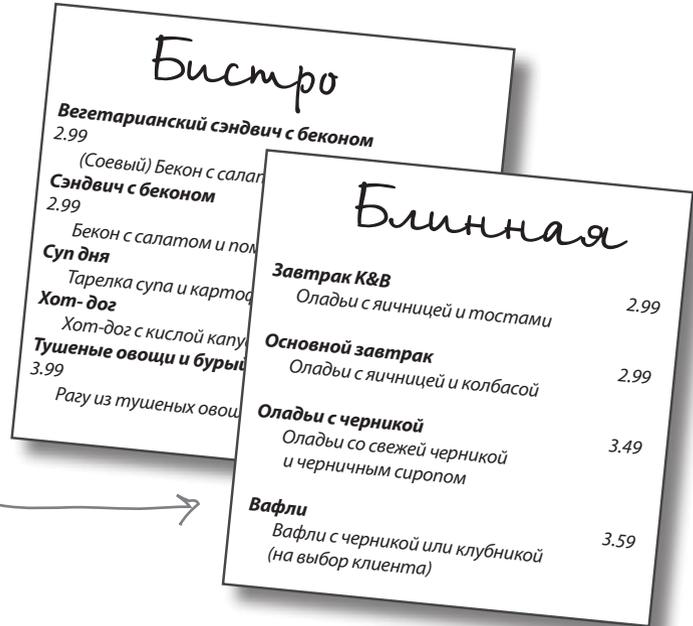
...но не можем согласовать реализацию самих меню. Мой коллега хранит элементы в контейнере ArrayList, а я использовал Array. Ни один из нас не желает изменять свою реализацию... Для нее написано слишком много кода, который от нее зависит.



## Проверяем элементы меню

По крайней мере, Лу и Мэл согласовали реализацию класса MenuItem. Рассмотрим содержимое обоих меню, а заодно познакомимся с реализацией.

*В меню бистро много обеденных блюд, а меню блинной состоит из завтраков. С каждым элементом меню связывается название, описание и цена.*



```
public class MenuItem {
    String name;
    String description;
    boolean vegetarian;
    double price;

    public MenuItem(String name,
                    String description,
                    boolean vegetarian,
                    double price)
    {
        this.name = name;
        this.description = description;
        this.vegetarian = vegetarian;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public double getPrice() {
        return price;
    }

    public boolean isVegetarian() {
        return vegetarian;
    }
}
```

*Объект MenuItem содержит несколько полей: имя, описание, флаг вегетарианского блюда и цена. Все эти значения передаются конструктору для инициализации объекта MenuItem.*

*Методы для чтения/записи полей элемента меню.*

## Две реализации меню

Давайте разберемся, о чем спорят Лу и Мэл. Оба повара потратили немало времени и сил на написание кода хранения элементов меню и другого кода, который от него зависит.



Реализация меню блинной.

```
public class PancakeHouseMenu {
    List<MenuItem> menuItems;

    public PancakeHouseMenu() {
        menuItems = new ArrayList<MenuItem>();

        addItem("K&B's Pancake Breakfast",
            "Pancakes with scrambled eggs, and toast",
            true,
            2.99);

        addItem("Regular Pancake Breakfast",
            "Pancakes with fried eggs, sausage",
            false,
            2.99);

        addItem("Blueberry Pancakes",
            "Pancakes made with fresh blueberries",
            true,
            3.49);

        addItem("Waffles",
            "Waffles, with your choice of blueberries or strawberries",
            true,
            3.59);
    }

    public void addItem(String name, String description,
        boolean vegetarian, double price)
    {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menuItems.add(menuItem);
    }

    public ArrayList<MenuItem> getMenuItems() {
        return menuItems;
    }

    // другие методы
}
```

Лу хранит элементы меню в ArrayList.

Каждый элемент меню включается в ArrayList в конструкторе. Для каждого объекта MenuItem задается имя, описание, признак вегетарианского блюда и цена.

Чтобы добавить новый элемент меню, Лу создает новый объект MenuItem, задает все необходимые аргументы и включает созданный объект в ArrayList.

Метод getMenuItems() возвращает список элементов меню.

Лу написал большой объем кода, зависящего от реализации ArrayList. И он не хочет переписывать весь код заново!



Какой еще ArrayList...  
Я выбрал НОРМАЛЬНЫЙ  
массив, чтобы ограничить  
максимальный размер  
меню.

А вот реализация Мэла.

```
public class DinerMenu {
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    MenuItem[] menuItems;
```

Мэл выбрал другой подход: он использует массив Array, чтобы ограничить максимальный размер меню.

```
public DinerMenu() {
    menuItems = new MenuItem[MAX_ITEMS];

    addItem("Vegetarian BLT",
        "(Fakin') Bacon with lettuce & tomato on whole wheat", true, 2.99);
    addItem("BLT",
        "Bacon with lettuce & tomato on whole wheat", false, 2.99);
    addItem("Soup of the day",
        "Soup of the day, with a side of potato salad", false, 3.29);
    addItem("Hotdog",
        "A hot dog, with saurkraut, relish, onions, topped with cheese",
        false, 3.05);
    // a couple of other Diner Menu items added here
}
```

Мэл тоже создает элементы меню в конструкторе при помощи вспомогательного метода addItem().

```
public void addItem(String name, String description,
    boolean vegetarian, double price)
{
    MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
    if (numberOfItems >= MAX_ITEMS) {
        System.err.println("Sorry, menu is full! Can't add item to menu");
    } else {
        menuItems[numberOfItems] = menuItem;
        numberOfItems = numberOfItems + 1;
    }
}
```

Метод addItem() получает все параметры, необходимые для создания MenuItem, и создает объект. Он также проверяет, не нарушает ли новый объект максимальный размер массива.

Мэл ограничивает размер меню, чтобы не запоминать слишком много рецептов.

```
public MenuItem[] getMenuItems() {
    return menuItems;
}
```

Метод getMenuItems() возвращает массив элементов меню.

```
// ...Другие методы меню...
}
```

Мэл ТОЖЕ написал большой объем кода, зависящего от выбранной им реализации меню. И он слишком занят, чтобы переписывать этот код заново.

## Какие проблемы создает наличие двух разных реализаций меню?

Чтобы понять, какие сложности возникают с двумя разными представлениями меню, мы попробуем реализовать клиент, использующий оба меню. Допустим, новая компания, возникшая в результате слияния, наняла вас для создания официантки с поддержкой Java (не забывайте, что вы находитесь в Объективиле!). Спецификация требует, чтобы официантка могла при необходимости напечатать сокращенное меню и даже определить, является ли блюдо вегетарианским, не обращаясь к повару!

Сначала посмотрим спецификацию, а затем шаг за шагом разберемся, что потребуется для ее реализации...

### Спецификация официантки с поддержкой Java

Официантка с поддержкой Java: проект «Элис»

```
printMenu()
  - выводит каждый элемент меню

printBreakfastMenu()
  - выводит только блюда завтраков

printLunchMenu()
  - выводит только обеденные блюда

printVegetarianMenu()
  - выводит все вегетарианские блюда

isItemVegetarian(name)
  - по названию блюда возвращает true, если
  оно является вегетарианским, или false в про-
  тивном случае
```

Официантка  
с поддержкой  
Java.



↑  
← Спецификация  
официантки

## Реализация спецификации: первая попытка

Начнем с реализации метода printMenu():

- 1 Чтобы вывести полное меню, необходимо вызвать метод getMenuItems() для всех элементов обеих реализаций. Обратите внимание: методы возвращают разные типы:

```
PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
ArrayList<MenuItem> breakfastItems = pancakeHouseMenu.getMenuItems();
```

```
DinerMenu dinerMenu = new DinerMenu();
MenuItem[] lunchItems = dinerMenu.getMenuItems();
```

Методы внешне похожи, но вызовы возвращают разные типы.

Здесь проявляются различия реализации: блюда для завтрака хранятся в ArrayList, а обеденные блюда — в Array.

- 2 Чтобы вывести меню блинной, мы перебираем элементы контейнера ArrayList, а для вывода меню быстро перебираются элементы Array.

```
for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = breakfastItems.get(i);
    System.out.print(menuItem.getName() + " ");
    System.out.println(menuItem.getPrice() + " ");
    System.out.println(menuItem.getDescription());
}

for (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
    System.out.print(menuItem.getName() + " ");
    System.out.println(menuItem.getPrice() + " ");
    System.out.println(menuItem.getDescription());
}
```

Теперь нам придется написать два разных цикла для перебора двух реализаций меню...

...один цикл для ArrayList...

другой — для Array.

- 3 Реализация каждого метода будет представлять собой вариацию на эту тему — содержимое двух меню будет перебираться в двух разных циклах. А если вдруг добавится новый ресторан со своей реализацией, то в программе будут использоваться *три* разных цикла.

## Возьми в руку карандаш



Какие из следующих утверждений относятся к нашей реализации `printMenu()`?

- A. Мы программируем для конкретных реализаций `PancakeHouseMenu` и `DinerMenu`, а не для интерфейсов.
- B. Официантка не реализует `Java Waitress API`, а следовательно, не соответствует стандарту.
- C. Если мы решим перейти с `DinerMenu` на другое меню с реализацией в виде хеш-таблицы, нам придется изменять большой объем кода.
- D. Официантка должна знать, как в каждом объекте меню организована внутренняя коллекция элементов, а это нарушает инкапсуляцию.
- E. В реализации присутствует дублирование кода: метод `printMenu()` содержит два разных цикла для перебора двух разновидностей меню. А при появлении третьего меню понадобится еще один цикл.
- F. Реализация не использует язык `MXML (Menu XML)`, что снижает ее универсальность.

## Что дальше?

Мы оказались в затруднительном положении. Мэл и Лу не желают изменять свои реализации, потому что им придется переписать большой объем кода в соответствующих классах меню. Но если ни один из них не уступит, наша реализация официантки окажется сложной в сопровождении и расширении.

Хорошо бы найти механизм, позволяющий им реализовать единый интерфейс для своих меню (они и так достаточно близки, если не считать возвращаемого типа метода `getMenuItems()`). Это позволит нам свести к минимуму конкретные ссылки, а также избавиться от повторения циклов при переборе элементов меню.

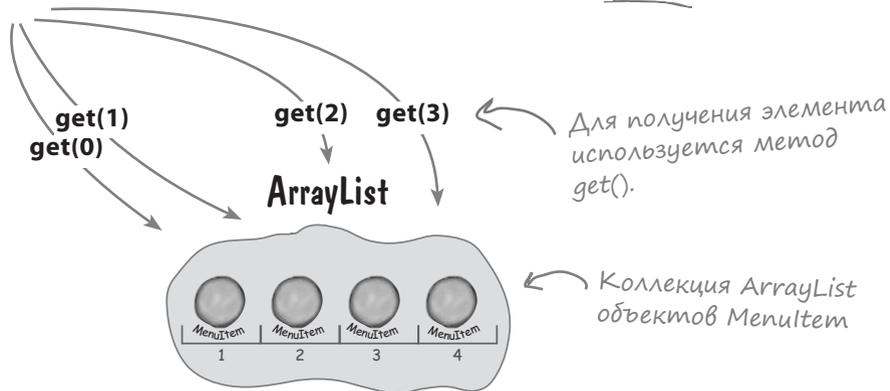
Заманчиво, верно? Но как это сделать?

## Как инкапсулировать перебор элементов?

Инкапсулируйте то, что изменяется, — это едва ли не самое важное из всего, о чем говорится в книге. Понятно, что изменяется в данном случае: механизм перебора для разных коллекций объектов (элементов меню). Но как его инкапсулировать? Давайте подробно разберем эту идею...

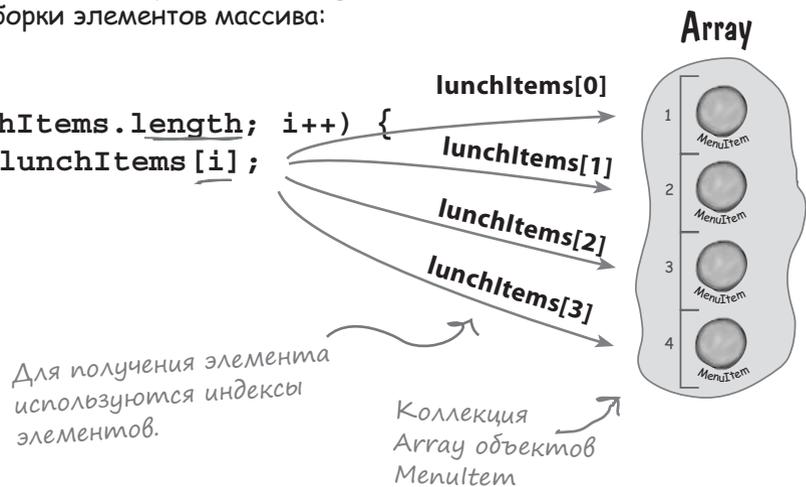
- 1 Для перебора элементов `ArrayList` используются методы `size()` и `get()`:

```
for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = breakfastItems.get(i);
}
```



- 2 Для перебора элементов массива используется поле `length` объекта `Array` и синтаксис выборки элементов массива:

```
for (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
}
```



## инкапсуляция итераций

- 3 Почему бы не создать объект (назовем его **итератором**), инкапсулирующий механизм перебора объектов в коллекции? Попробуем сделать это для `ArrayList`:

Запрашиваем у `breakfastMenu` итератор для перебора объектов `MenuItem`.

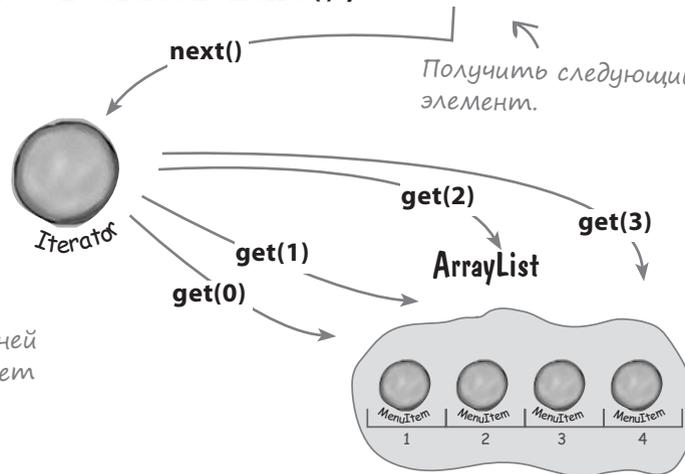
```
Iterator iterator = breakfastMenu.createIterator();
```

```
while (iterator.hasNext()) {  
    MenuItem menuItem = iterator.next();  
}
```

Пока остаются элементы...

Получить следующий элемент.

Клиент просто вызывает `hasNext()` и `next()`; во внутренней реализации итератор вызывает `get()` для `ArrayList`.



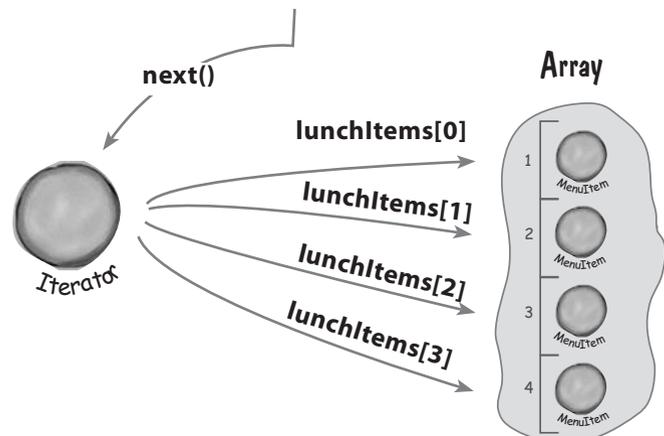
- 4 Теперь сделаем то же для `Array`:

```
Iterator iterator = lunchMenu.createIterator();
```

```
while (iterator.hasNext()) {  
    MenuItem menuItem = iterator.next();  
}
```

Код точно такой же, как для `ArrayList`.

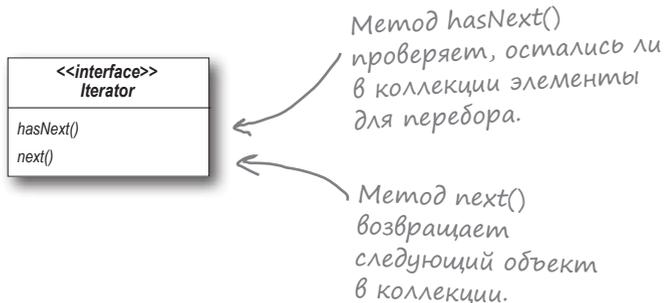
Аналогичная ситуация: клиент вызывает `hasNext()` и `next()`, а итератор во внутренней реализации индексирует `Array`.



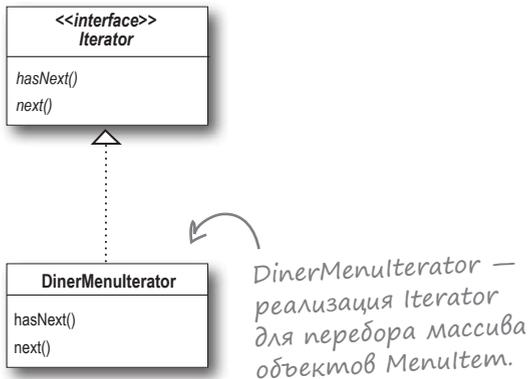
## Паттерн Итератор

Похоже, наш план инкапсуляции перебора элементов вполне реален. И как вы, вероятно, уже догадались, для решения этой задачи существует паттерн проектирования, который называется Итератор.

Первое, что необходимо знать о паттерне Итератор, — то, что он зависит от специального интерфейса (допустим, `Iterator`). Одна из возможных форм интерфейса `Iterator`:



При наличии такого интерфейса мы можем реализовать итераторы для любых видов коллекций объектов: массивов, списков, хеш-карт... Допустим, мы хотим реализовать итератор для коллекции `Array` из нашего примера. Реализация будет выглядеть так:



Давайте реализуем этот итератор и свяжем его с коллекцией `DinerMenu`, чтобы вы лучше поняли, как работает механизм перебора...

Под термином КОЛЛЕКЦИЯ мы подразумеваем группу объектов. Такие объекты могут храниться в разных структурах данных: списках, массивах, хеш-картах... но при этом все равно остаются коллекциями.

