#9 Модельный показ

В этом задании мы начнем практиковаться в подходе MV* и перепишем взаимодействие модулей так, чтобы их роли были разделены. У нас получатся отдельные модули, которые будут работать с данными и отдельные модули, которые будут управлять отображением этих данных и переключением экранов. Модули, отвечающие за отображение, будут описаны в виде классов, остальные модули будут функциональными.

Перепишите код вашего проекта следующим образом:

1. Создайте базовый класс для представлений: `AbstractView`. Этот класс должен иметь следующие методы:
    - Геттер `template`, который возвращает строку, содержащую разметку. Метод должен быть абстрактным, то есть этот метод должен быть обязательно переопределен в объектах-наследниках
    - `render` — метод, который будет создавать DOM-элемент на основе шаблона, который возвращается геттером `template`
    - `bind` — метод, который будет добавлять обработчики событий. Метод должен быть абстрактным и переопределяться для каждого из наследников. Обратите внимание, что этот метод не должен вызывать ошибку, потому что его необязательно реализовывать в классах-наследниках, то есть может выполняться метод `bind` класса `AbstractView`
    - Геттер `element`, который возвращает DOM-элемент, соответствующий представлению. Метод должен создавать DOM-элемент с помощью метода `render`, добавлять ему обработчики, с помощью метода `bind` и возвращать созданный элемент. *Метод должен использовать **ленивые вычисления** — элемент должен создаваться при первом обращении к методу с помощью метода `getMarkup`, должны добавляться обработчики (если возможно). При последующих обращениях должен использоваться элемент, созданный при первом вызове метода*
2. Для каждого из экранов добавьте модуль, который будет отвечать за отображение соответствующего экрана. Этот модуль должен быть наследником модуля `AbstractView` и расширять его методы.

   Каждый из модулей отображения (view) должен быть пассивным, то есть в самом модуле не должно быть заложено никакой логики по переключению экранов, каждое отображение умеет только отрисовывать переданные данные. Все изменения передаются внешнему слушателю через обработчик событий или коллбэк и вся работа с данными должна вестись снаружи

3. Модуль отображения экрана должен реализовывать паттерн «Слушатель»: модуль должен содержать метод, который будет переопределяться снаружи
4. Для каждого из экранов добавьте модуль, который будет управлять отображением этого экрана. Этот модуль должен быть подписан на изменения в модуле отображения (view) и при изменениях в нём должен вызывать соответствующие действия: изменять состояние *(например, уменьшать количество жизней)*, переключать экраны и так далее.

   Пример

У вас есть модуль отрисовки экрана `buttonView`, который принимает на вход объект с данными этого экрана. Этот модуль отрисовывает DOM-элемент экрана, содержащий кнопку и добавляет на него обработчик, который показывает диалоговое окно.

Вам нужно разделить логику этого модуля на две части: часть отвечающую за отрисовку и работу с DOM и часть, которая управляет переключением экранов. В итоге у вас должно получиться два модуля: `ButtonView` и `button`.

- **Модуль `ButtonView`** будет экспортировать класс, который наследуется от `AbstractView`. Этот модуль будет создавать DOM-элемент экрана и добавлять на него обработчики DOM-событий. Кроме этого никакой логики в этом модуле заложено не будет

  Класс `ButtonView` должен содержать некий метод `onClick`, который будет переопределяться снаружи для того, чтобы мы могли описать изменения которые должны произойти при нажатии на кнопку

- **Модуль `button`**, который будет экспортировать функцию. Эта функция будет вызывать отрисовку класса `ButtonView` и переопределять метод `onClick` так, чтобы при каждом вызове `onClick` показывалось диалоговое окно

```
// button-view.js
/**
 * Модуль ButtonView занимается только отображением
 * кнопки. Поскольку модуль знает как именно отрисовывается
 * кнопка, он добавляет на нее нужные обработчики, но никакой
 * специальной логики в этих обработчиках не содержится —
 * вся логика будет описана снаружи, потому что ButtonView
 * должен быть «пассивным» модулем, который ничего не знает
 * про поведение кнопки. В этом заключается особенность
 * подхода MV*: каждый модуль выполняет свою роль и модуль
 * отображения описывает только логику отображения
 */
export default class ButtonView extends AbstractView {
  /** Геттер template создает разметку экрана */
  get template() {
    return `<button>Нажми меня</button>`
  }

  /**
   * Метод bind описывает поведение кнопки при нажатии на нее.
   * Обратите внимание, что метод не вызывает напрямую действия
   * которые должны произойти по нажатию на кнопку, а вместо
   * этого вызывает коллбэк onClick, который будет переопределяться
   * снаружи (паттерн «Слушатель»)
   */
  bind() {
    this.element.onclick = (evt) => {
      evt.preventDefault();
      this.onClick();
    }
  }

  onClick() {

  }
}
```

```
// button.js
/**
 * Файл button.js управляет тем, как ведет себя ButtonView.
 * Для того, чтобы отрисовать экран, он создает новый
 * объект ButtonView и переопределяет коллбэк onClick.
 * Таким образом ButtonView просто предоставляет инструменты,
 * которые помогают описать любое поведение, а метод button,
 * пользуясь этими инструментами, описывает бизнес-логику
 * нажатия на кнопку
 */
import ButtonView from `./button-view`;

export default (data) => {
  const myButton = new ButtonView();

  /**
   * Показ диалогового окна описывается снаружи модуля
   * ButtonView
   */
  myButton.onClick = () => {
    alert(`Диалоговое окно`);
  };
}
```

Названия методов (например `onClick`) условны и даны только для примера. В вашем случае методы должны называться соответственно логике вашего приложения.