React.js Course

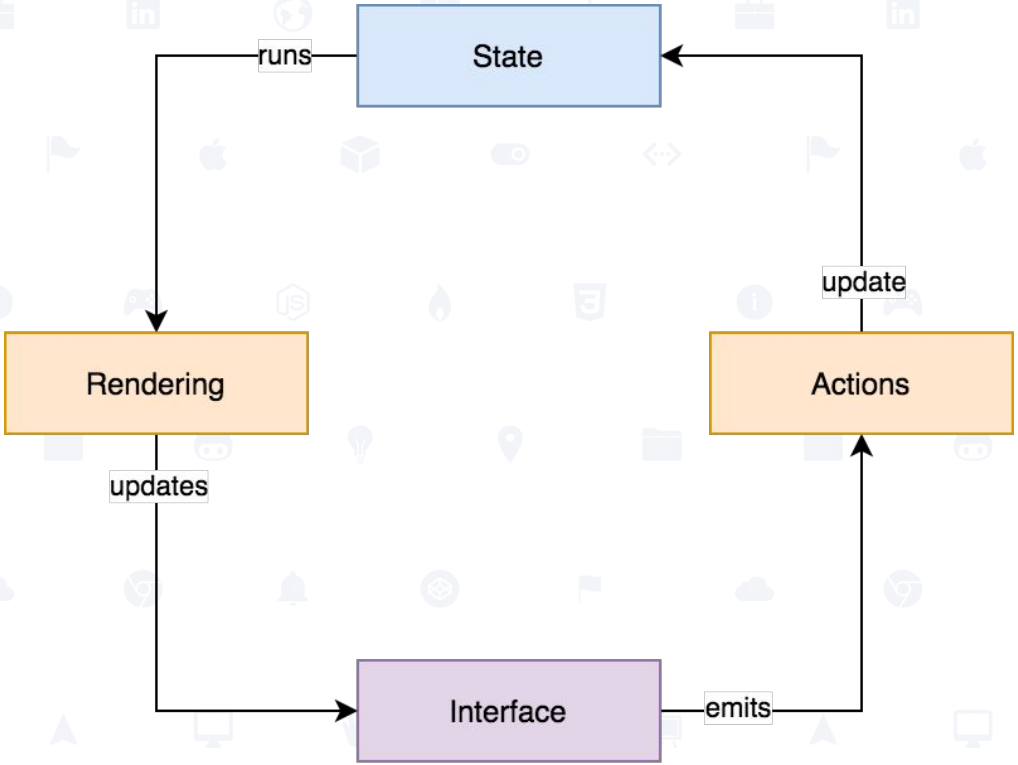
Занятие 2. Компоненты и состояние

Автор программы и преподаватель - Помазкин Илья

Структура занятия

1. Классовые компоненты
2. Состояние
3. Обработка событий
4. Работа с формами

Напоминание - концепция интерактивности



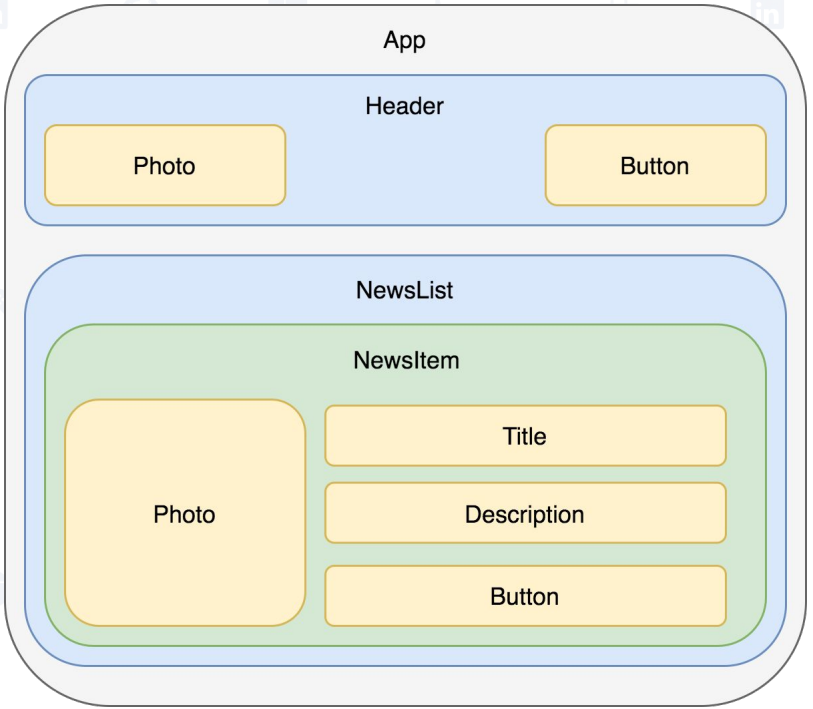
1. Пользователь видит **интерфейс** и делает в нем какие-то действия
2. **Действия** обновляют состояние
3. Изменение **состояния** запускает рендер
4. **Рендер** обновляет интерфейс

Компоненты

Компоненты - это кирпичики, из которых строится приложение.
- упрощают логику
- легко переиспользовать
- легко компоновать

Примеры простых компонентов:
- фото
- кнопка
- заголовок
- список тегов

Примеры более сложных составных компонентов:
- карточка товара, новости, поста
- хедер, футер
- модальное окно



Компоненты - изолированы

Каждый компонент ничего не знает ни о приложении, ни одругих компонентах - он изолирован. Единственное о чем знает компонент - это входные данные, **пропсы**.

В них мы можем передать:
- дочерний контент
- коллбеки
- любые входные данные
- спец. пропсы: key, ref, …

Компоненты похожи на функции: в них можно передать набор параметров, пропсов, а в ответ получить React-элемент.

Классовые компоненты

React-компоненты есть двух типов: классовые и функциональные. На этом уроке рассмотрим классовые.

Классовый компонент:

```
import React, { Component } from "react";

export class Button extends Component {
   render() {
       const { children, ...rest } = this.props;

       return (
           <button {...rest}>{children}</button>
       );
   }
}
```

Так же у класса есть дополнительные методы - мы разберем их чуть позже.

Класс должен наследовать от React.Component.
За рендер отвечает метод render - он должен вернуть React-элемент.

Пропсы записываются в свойство объекта “props”.
К ним можно обратиться через this.props в методе render.

Дочерние элементы доступны в props.children.

Пропсы

Пропсы очень похожи на аргументы функции:
- каждый пропс - как отдельный именованый аргумент
- можно присваивать значения по умолчанию
- можно указывать тип аргумента, почти как в TypeScript

Пропсы условно можно разделить на 3 вида:
- React-пропсы: *key, ref, dangerouslySetInnerHTML*
- пользовательские пропсы: *что угодно*
- дочерний контент: *children*

Валидация пропсов

Поскольку JS - это язык с динамической типизацией, мы не можем указать, пропсы какого типа ожидаем получить. А очень часто валидацию делать нужно, например при разработке UI-библиотеки.

Для этого в React-стеке есть пакет prop-types.

Для того, что бы указать типы пропсов нужно классу или функции компонента присвоить статическое свойство **propTypes**.

В него нужно присвоить объект, где ключ - это название пропса, а значение - это функция, которая будет вызвана для валидации.

```
import React, { Component } from 'react';
import PropTypes from 'prop-types';

export class Button extends Component {
 render() {
   return 'some content...';
 }
}

Button.propTypes = {
 as: PropTypes.oneOf([
   "link", "button", "div"
 ]).isRequired,
 href: PropTypes.string,
 onClick: PropTypes.func,
 uid: PropTypes.number,
 data: PropTypes.shape({
   id: PropTypes.number.isRequired,
   title: PropTypes.node,
   description: PropTypes.string,
 }),
};
```

Значение пропсов по умолчанию

Для того, что бы указать значение пропсов по умолчанию нужно классу или функции компонента присвоить статическое свойство **defaultProps**.

В него нужно присвоить объект, где ключ - это название пропса, а значение - это собственно дефолтное значение пропса.

Если в компонент не будет передан какой-то пропс, React попробует установить значение пропса по умолчанию (если оно задано). После этого будет запущена проверка типов пропсов.

Дефолтные значения тоже будут валидироваться.

```
import React, { Component } from 'react';
import PropTypes from 'prop-types';

export class Button extends Component {
 render() {
   return 'some content...';
 }
}

Button.propTypes = {
 as: PropTypes.oneOf([
   "link", "button", "div"
 ]).isRequired,
 href: PropTypes.string,
};

Button.defaultProps = {
 as: "button",
};
```

Состояние

Для работы с сотоянием нужно определиться, за чем мы будем следить и какие есть состояния.

Например:
- открыто / закрыто - для меню, выпадающих списков
- выбранная опция - для списка опций
- выбранные категории - для списка категорий
- текстовое значение - для текстовых полей
- и т.д.

После этого состояние можно описать как объект с определенными свойствами.

```
// Пример объекта состояния для секции с фильтрами в виде чекбоксов
const state = {
 isOpen: false, // секция может быть открыта или закрыта, поэтому просто булево значение
 selected: [],  // а вот выбранных фильтров может быть несколько - поэтому храним их как массив
};
```

Состояние компонента

Точно так же и в компоненте - нужно определить и присвоить начальное состояние.

Для этого объекту, который создается классом нужно задать свойство **state**. Сделать это можно в конструкторе или напрямую указав поле класса.

```
export class Filters extends Component {
 constructor(props) {  // в конструктор первым аргументом попадают пропсы
   super(props);       // их нужно передать в родительский конструктор
   this.state = {      // присваиваем инстансу класса объект состояния
     isOpen: false,
     selected: [],
   };
 }
}
```

```
export class Filters extends Component {
 state = {             // присваиваем инстансу класса объект состояния
   isOpen: false,
   selected: [],
 };
}
```

Чтение состояния и рендеринг

Состояние будет храниться ровно там, где мы его определили - в поле инстанса класса (поле объекта, который был создан этим классом). Получить к нему доступ можно через **this.state**.

Таким образом мы можем использовать его при рендеринге:

```
export class Filters extends Component {
 state = {              // присваиваем инстансу класса объект состояния
   isOpen: false,
   selected: [],
 };

 render() {
   return (
     <div className="filters">
       <button className="filters__toggler">
         {this.state.isOpen ? 'Close' : 'Open'}
       </button>
       <div className="filters__cont">
         ...
       </div>
     </div>
   );
 }
}
```

Обновление состояния

Для обновления состояния в классе React.Component есть метод **setState**. Поскольку класс нашего компонента наследует от React.Component - у нашего компонента тоже будет доступен этот метод.

Для ***запроса на обновление состояния*** вызовите **this.setState**. Описание метода:

```
setState(updater, [callback])
```

Первый аргумент - это updater. Это либо объект с набором полей состояния, которые нужно обновить, либо функция, которая возвращает такой объект.

Второй аргемент, необязательный - это функция-коллбек. Она будет вызвана после того, как состояние будет обновлено.

```
// передача объекта с полями для обновления
this.setState({
  isOpen: !this.state.isOpen,
});
```

```
// передача функции для обновления
// функция получит 2 агрумента - ссылки на состояние и пропсы
this.setState((state, props) => {
 // state и props изменять нельзя
 // функция должна вернуть объект с полями для обновления
 return {
   isOpen: !state.isOpen,
 };
});
```

setState - это запрос на обновление

Вызов функции setState **не вызовет моментальное обновление** состояния. Вместо этого мы даем знать React, что хотим обновить состояние. А React сам принимает решение, когда это сделать.

Это сделано для, того что бы React мог делать оптимизации: собирать несколько вызовов setState в один или откладывать обновление.

Если в момент вызова setState вы строите новое состояние исходя из текущего состояния, то обращение к this.state может вернуть неактуальное, старое состояние.

Если нужно иметь доступ к актуальному состоянию и пропсам - используйте setState с функцией в качестве 1 аргумента.

```
// не факт, что будет актуальное состояние
this.setState({
  isOpen: !this.state.isOpen,
});
```

```
// тут всегда будет актуальное состояние
this.setState((state, props) => {
 return {
   isOpen: !state.isOpen,
 };
});
```

Обработка событий

Для привязки DOM-события в React-элемент нужно передать проп, имя которого строится как:

on + EventName

onClick

onKeyPress

onFocus

В сам проп нужно передать функцию-обработчик события. При ее вызове первым агрументом будет объект синтетического события.

```
<button
  onClick={(e) => console.log(e)} // добавляем обработчик события click
  className="filters__toggler">
  ...
</button>
```

Обработка событий - обновление состояния

Внутри обработчиков событий мы можем делать запрос на обновление состояния, тем самым замыкая цикл интерактивности.

```
export class Filters extends Component {
 state = { // присваиваем инстансу класса объект состояния
   isOpen: false,
   selected: [],
 };

 handleToggle() {
   // делаем запрос на обновление состояния
   this.setState((state) => ({ isOpen: !state.isOpen }));
 }

 render() {
   return (
     <div className="filters">
       <button
         onClick={this.handleToggle.bind(this)} // добавляем обработчик события click
         className="filters__toggler">
         {this.state.isOpen ? 'Close' : 'Open'}
       </button>
       <div className="filters__cont">...</div>
     </div>
   );
 }
}
```

Обработчики событий - проблема контекста this

В классовых компонентах обработчики событий обычно выносят в методы класса. Это удобно и оптимально для производительности. Но есть одно проблема - по умолчанию методы класса не привязаны к контексту.

То есть: если прямо в JSX передать обработчик как *this.handlerName,* тогда в момент исполнения JS *this* внутри функции обработчика будет равен undefined.

```
class Checkbox extends Component {
 state = { checked: false };

 handleToggle() {
   // this будет равен undefined
   this.setState((state) => ({ checked: !state.checked }));
 }

 render() {
   return (
     <button
       onClick={this.handleToggle}> {/* Передаем ссылку на
метод без привязки к this */}
       {this.state.checked ? "+" : "-"}
     </button>
   );
 }
}
```

Решение проблемы контекста - Function.bind

```
class Checkbox extends Component {
 constructor(props) {
   super(props);
   this.state = { checked: false };
   this.handleToggle = this.handleToggle.bind(this); // привязка метода в конструкторе
 }

 handleToggle() {
   // this будет равен нашему компоненту
   this.setState((state) => ({ checked: !state.checked }));
 }

 render() {
   return (
     <button
       onClick={this.handleToggle}>
       {this.state.checked ? "+" : "-"}
     </button>
   );
 }
}
```

Решение проблемы контекста - свойства класса

```
class Checkbox extends Component {
 state = { checked: false };

 // присваиваем свойству объекта стрелочную функцию
 handleToggle = () => {
   // this будет равен нашему компоненту
   this.setState((state) => ({ checked: !state.checked }));
 };

 render() {
   return (
     <button
       onClick={this.handleToggle}>
       {this.state.checked ? "+" : "-"}
     </button>
   );
 }
}
```

* Это [экспериментальный синтаксис](), но по факту его уже все давно юзают.

Решение проблемы контекста - стрелочные функции в коллбеке

```
class Checkbox extends Component {
 state = { checked: false };

 handleToggle() {
   // this будет равен нашему компоненту
   this.setState((state) => ({ checked: !state.checked }));
 }

 render() {
   return (
     <button
       {/* Передаем стрелочную функцию, внутри которой обращаемся к нужному методу (замыкание) */}
       onClick={(e) => this.handleToggle(e)}>
       {this.state.checked ? "+" : "-"}
     </button>
   );
 }
}
```

* Может сказаться на производительности, потому что каждый рендер будет создаваться новая функция-обработчик

Решение проблемы контекста - ::this.handler

```
class Checkbox extends Component {
 state = { checked: false };

 handleToggle() {
   // this будет равен нашему компоненту
   this.setState((state) => ({ checked: !state.checked }));
 }

 render() {
   return (
     <button
       {/* Это аналог this.handleToggle.bind(this) */}
       onClick={::this.handleToggle}>
       {this.state.checked ? "+" : "-"}
     </button>
   );
 }
}
```

* Это [экспериментальный синтаксис](), который может часто встречаться **в старом коде**.

Синтетические события

Для упрощения и стандартизации React использует синтетические события. Это - обрертка над нативным браузерным событием.

- такие события работают во всех брузерах одинаково
- упрощается и стандартизируется доступ к доп. данным события: координатам курсора, код нажатой клавиши и т.д.
- тип синт. события и нативного события могут не совпадать: например синт. событие *onMouseLeave* будет указывать на нативное событие *mouseout*

Если нужен доступ к нативному событию браузера, нужно обратиться к свойству *event.nativeEvent*

Детальную документацию можно почитать вот тут.

К чему в действительности привязываются события

Очень полезно вспомнить, как в DOM работает *высплытие* событий.
А так же узнать про прием *делегирования* событий.

В действительности React привязывает события к корневому элементу приложения (начиная с версии 17, до нее события привязывались к document).
В момент срабатывания нативного события React делает следующее:

- Через event.target узнает, на каком DOM-элементе сработало нативное событие
- Создает экземпляр SynteticEvent, копирует в него данные из нативного события, стандартизирует их
- У SynteticEvent присваевает свойству target значение event.target
- Зная структуру DOM и дерева компонентов, React находит DOM-элемент, к которму мы привязывали событие в JSX и этот DOM-элемент присваиват в SynteticEvent.currentTarget
- Вызывает функцию-обработчик события, которую мы добавили в JSX и передает в нее SynteticEvent

Работа с формами

Формы - это прежде всего поля (input, поле ввода). В React с полями можно работать в двух режимах:

- управляемое поле
- неуправляемое поле

Поля в HTML уже сами по себе интерактивны - браузер берет на себя реализацию интерактивности. React в этот цикл интерактивности никак не вмешивается - все делает сам браузер. Такие поля называются **неуправляемыми**.

Если нам нужно контролировать цикл интерактивности, то его реализацию нам нужно делать самим - с помощью обработки событий, установки состояния и присваивания значения полю. Такие поля называются **управляемыми**.

Управляемые компоненты

Ниже - типичный пример управляемого поля:

```
export class SearchForm extends React.Component {
 state = { query: '' }; // будем хранить текущее значение поля в состоянии

 handleChange = (e) => {
   // реакция на изменение поля - обновляем состояние новым значением поля
   let val = e.target.value;
   this.setState({ query: val });
 };

 render() {
   return (
     <div className="search">
       <input
         value={this.state.query}      // устанавливаем атрибут value
         onChange={this.handleChange} // добавляем обработчик события на изменение поля
         type="text"
         className="search__input"/>
     </div>
   );
 }
}
```

Управляемые компоненты - text, textarea

textarea в React ведет себя так же как и текстовое поле - установка значение происходит через установку пропа value.

```
render() {
 return (
   <div className="form">
     <input type="text" value={this.state.firstName} onChange={this.handleChangeFirstName} />
     <textarea value={this.state.message} onChange={this.handleChangeMessage} />
   </div>
 );
}
```

Управляемые компоненты - radio, checkbox

С чекбоксами и радиокнопками нужно ориентироваться на атрибут checked. Именно его устанавливаем как значение поля.

Чекбоксы:

```
handleChangeEnabled = (e) => {
  let enabled = e.target.checked; // проверка атрибута checked
  this.setState({ enabled });
};

render() {
 return (
   <div className="form">
     <input
       type="checkbox"
       name="enabled"
       checked={this.state.enabled} // установка значения
       onChange={this.handleChangeEnabled} />
   </div>
 );
}
```

Радиокнопки:

```
handleChangeColor = (e) => {
  let color = e.target.value; // берем value выбранного radio
  this.setState({ color });
};

render() {
 return (
   <div className="form">
     <input type="radio" name="color"
       value="red" // у каждого radio есть свое значение
       checked={this.state.color === "red"} // установка checked в true, если текущий цвет равен значению radio
       onChange={this.handleChangeColor} />
   </div>
 );
}
```

Управляемые компоненты - select

Селекты рендерятся как в HTML - тег select, и внутри него набор option.

Пропсы onChange и value устанавливаются у элемента select.

Если в селекте включен мультивыбор - в значении поля будет массив выбранных опций

Обычный селект:

```
handleChangeLanguage = (e) => {
 let lang = e.target.value;
 this.setState({ lang });
};

render() {
 return (
   <div className="form">
     <select
       onChange={this.handleChangeLanguage}
       value={this.state.lang}
       name="language">
       <option value="uk">UK</option>
       <option value="ru">RU</option>
     </select>
   </div>
 );
}
```

С мультивыбором:

```
handleChangeColors = (e) => {
 let colors = e.target.value; // тут будет масссив
выбранных опций
 this.setState({ colors });
};

render() {
 return (
   <div className="form">
     <select
       multiple // включаем мультивыбор
       onChange={this.handleChangeColors}
       value={this.state.colors}
       name="colors">
       <option value="red">Red</option>
       <option value="green">Green</option>
       <option value="blue">Blue</option>
     </select>
   </div>
 );
}
```

Управляемые компоненты - форматирование значения

Можно форматировать значение поля прямо в обработчике события. Например, если нужно запретить вводить в текстовое поле что угодно кроме цифр:

```
handleChangeNumber = (e) => {
 let val = e.target.value;

 // удаляем из строки все символы, которые не являются цифрами
 val = val.replace(/\D/gmi, '');

 this.setState({
   number: val,
 });
};

render() {
 return (
   <div className="form">
     <input type="text" value={this.state.number} onChange={this.handleChangeNumber}/>
   </div>
 );
}
```

Управляемые компоненты - пример валидации

В момент изменения значения можно делать валидацию и выводить соответственные сообщения.

```
handleChangeNumber = (e) => {
 let val = e.target.value,
   errorMessage = null; // по умолчанию ошибки нет

 // если значение не пустое - делаем валидацию
 if (val.length && !/^\d+$/gmi.test(val)) {
   errorMessage = "Значение поля должно содержать только цифры!";
 }

 this.setState({
   number: val,
   errorMessage, // устанавливаем сообщение об ошибке
 });
};

render() {
 return (
   <div className="form">
     <input type="text" value={this.state.number} onChange={this.handleChangeNumber}/>
     {this.state.errorMessage && (
       <span className="error">{this.state.errorMessage}</span>
     )}
   </div>
 );
}
```

Неуправляемые компоненты

Для работы с неуправляемыми компонентами есть два варианта:

- добавляем обработчик события onChange и при измененении значения копируем его в state
- в момент отправки формы обращаемся к DOM-элементу каждого поля и берем из него значение поля. Для этого нужно использовать *рефы (ref)* - их мы разберем на следующих занятиях.

Неуправляемые компоненты - text input, значение по умолчанию

Если мы укажем полю значение value, но не добавим обработку изменений - пользователь не сможет ничего вводить в поле. React раз за разом будет устанавливать значение, переданное в проп value.

```
render() {
 // юзер не может ничего ввести в поле
 return <input type="text" value="123" /> ;
}
```

Но иногда нам нужно установить какое-то начальное значение поля и после - разрешить юзеру его редактировать. Для этого нужно передать проп defaultValue.

```
render() {
 // теперь у умолчанию у поля будет значение "123"
 // но юзер уже сможет вводить что-то в поле
 return <input type="text" defaultValue="123" /> ;
}
```

Неуправляемые компоненты - file input

Значение файлового поля нельзя устанавливать из JS, иначе это было бы небезопасно. Поэтому с мы можем только отслеживать изменение файлового поля и как-то на это реагировать. (File API в помощь :))

```
handleChangeFile = (e) => {
 let fileSize = null,
   files = e.target.files; // получаем доступ к прикрепленным файлам

 // если минимум 1 файл есть - читаем его размер
 if (files.length) fileSize = files[0].size;

 // устанавливаем размер файла
 this.setState({ fileSize });
};

render() {
 return (
   <div>
     <input
       type="file"
       // можем только следить за изменениями
       onChange={this.handleChangeFile} />
     {this.state.fileSize !== null && ( // если размер файла известен - выводим его
       <span>File Size: {this.state.fileSize} bytes</span>
     )}
   </div>
 );
}
```

Обработка отправки формы

Для перехвата отправки формы есть событие onSubmit. В обработчике события мы отменяем отправку формы браузером и делаем что-то.

Если все поля управляемые - то значение полей берем из state.

Если поля не управляемые - но значение полей берем напрямую из DOM.

```
handleSubmit = (e) => {
 e.preventDefault(); // отменяем отправку формы браузером
 // делаем что-то
};

render() {
 return (
   <form onSubmit={this.handleSubmit}>
     <input type="text" name="firstName" value={this.state.firstName} onChange={this.handleChange} />
     <input type="text" name="lastName" value={this.state.lastName} onChange={this.handleChange} />
     <input type="submit" />
   </form>
 );
}
```

Библиотеки для работы с формами - введение

Библиотеки для работы с формами деляться на 2 типа:

- с управляемыми полями (Formik)
- и не управляемыми полями (react-hook-form, react-final-form)

Они сильно упрощают жизнь и позволяют делать:

- валидацию, показ ошибок
- форматирование значений
- доступ к значениям полей для неконтролируемых полей
- и т.д.

Мы разберем библиотеки работы с формами на одном из следующих уроков.