# Объектно-ориентированное программирование

* **Объектно-ориентированное программирование** - методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования

## 1. Инкапсуляция

* **Инкапсуляция** – отделение и защита внутреннего интерфейса от внешнего
* **Внутренний интерфейс** – приватные свойства и методы, доступ к которым может быть осуществлен только из других методов класса
* **Внешний интерфейс** – публичные свойства и методы класса, предоставляющие доступ к внутренней реализации
* Мы обращаемся с объектами как с единой сущностью, а не как с набором отдельных полей и методов, тем самым скрываем и защищаем реализацию класса. Если клиентский код не знает ничего, кроме публичного интерфейса, он не может зависеть от деталей реализации
* Обычно прячут все поля класса и предоставляют набор методов для работы с ними. У разработчиков есть универсальный интерфейс для доступа к свойствам, им не важна какая реализация метода

#### Задача

* Обеспечить согласованность и целостность данны класса
* Защитить данные от некорректного использования или повреждения
* Нет зависимости от деталей внутренней реализации

#### Сокрытие и Инкапсуляция

* Сокрытие - разграничение доступа частей программы к компонентам друг друга
* Инкапсуляция - свойство системы объединять данные и методы работы с ними в класс

#### Геттеры и Сеттеры

* Геттеры и Сеттеры (Аксессоры) - доступ к свойствам класса

#### Модификаторы доступа (Методы классов)

* Модификаторы доступа (access modifiers) - способ ограничения доступа к методам, классам, переменным
* **private** - Видны внутри класса. Запрещает доступ из подкласса. Методы не наследуются
* **protected** - Видны внутри класса, пакета и наследников. Нельзя вызвать вне класса
* **static** - Не имеет доступ к instance класса (this нельзя получить). Н-р, Math статический класс, в котором содержится набор статических методов. Не создаем instance через new, но можем пользоваться методами
* **package** - Видны только из текущего пакета
* **public** - Видны из всего приложения. Можно обратиться к this

## 2. Наследование

* **Наследование** - описание нового класса на основе уже существующего с частично или полностью заимствующейся функциональностью
* Наследование транзитивно: класс может наследоваться от другого класса, который наследуется от третьего, и так далее вплоть до базового класса (Object). Наследник может переопределить какие-то методы и поля чтобы изменить поведение по умолчанию
* *SuperClass: базовый, родительский, предок, надкласс, супер класс*
* *SubClass: наследник, дочерний, потомок, подкласс, суб класс, производный*
* Подкласс должен являться более специализированной версией супер-класса
* Множественное наследование - у одного подкласса существует сразу несколько супер-классов

## 3. Полиморфизм

* **Полиморфизм** ("множество форм") - возможность единообразно обрабатывать объекты с различной реализацией при условии наличия общего интерфейса. «Один интерфейс, множество реализаций»
* Позволяет одно и то же имя (например имя метода) использовать для решения внешне схожих, но технически разных задач
* Изменение поведения одного и того же метода в рамках разных классов
* **Задачи:** позволяет расширить библиотеки не требуя доступа к их исходному коду
* Абстрактный метод - метод без реализации
* Абстрактный класс - если в классе есть хотя бы 1 абстрактный метод, класс является абстрактным, следовательно мы не можем создать экземпляр класса и воспользоваться им, пока не переопределим этот метод, добавляя различную реализацию в подклассах
* Перегрузка - одна из форм полиморфизма (ad hoc / специальный полиморфизм)
* Механизм "Позднего связывания" / "Динамического связывания" - откладывает принятие решения о том, какой из вариантов кода выполнить до момента выполнения программы. В результате вызова метода draw(), будут выполнены именно те инструкции, которые определены в фактическом классе данного объекта. Без механизма позднего связывания полиморфизм не будет работать

## 4. Абстракция

* **Абстракция** способ представления объекта в программе, позволяющий работать с объектами не вдаваясь в особенности их реализации
* Инкапсуляция помогает реализовать абстракцию. Можно абстрагировать внутренне устройство класса
* Абстракция - объекты из окружающего мира которые мы хотим описать. Выделяем только черты, необходимые для решения задачи. Н-р: есть животное, но для конкретной задачи нужны не все характеристики, а только "вес" и "возраст"
* Абстрагировать объект или явление - отбросить все маловажные элементы для того, чтобы увидеть его суть
* Когда моделируем объект, можем отказаться от некоторых его частей неважных в контексте программы

# Функциональное программирование

* **Функциональное программирование** - последовательный вызов набора функций
* Методы для функционального программирования в JavaScript: arr.map, arr.forEach, arr.filter, arr.sort, arr.concat, arr.join, arr.reduce, arr.reduceRight, Object.assign, Object.keys()

**Что не используется**

1. Хранение состояния (нет переменных и операторов присваивания)

**Концепции**

1. **Чистые функции** (pure functions) - функция, которая при одинаковых аргументах всегда возвращает одни и те же значения и не имеет видимых побочных эффектов (side effects) ввода-вывода и памяти. Зависят только от своих параметров и возвращают только свой результат. Не влияют на состояние программы

**Условия**

* + Детерменированная - для одного и того же набора входных значений возвращает одинаковый результат
  + Без побочных эффектов В императивных языках некоторые функции в процессе выполнения своих вычислений могут модифицировать значения глобальных переменных, осуществлять операции ввода-вывода, реагировать на исключительные ситуации, вызывая их обработчики. Они называются функциями с побочными эффектами
  + Для замены циклов в JS можно использовать map и reduce

1. **Функции высшего порядка** - функция принимающая в качестве аргументов другие функции или возвращающая другую функцию в качестве результата
2. **Функции первого класса** - функция трактуются как объект, то есть может быть передана другим функциям и ее можно вернуть из функций. Так же ее можно присваивать переменным let func = function(){}
3. **Каррирование функций** - все функции могут быть от одного аргумента. Преобразование функции от пары аргументов в функцию, берущую свои аргументы по одному
4. **Рекурсия** В функциональных языках цикл обычно реализуется в виде рекурсии. Рекурсивные функции вызывают сами себя, позволяя операции выполняться снова и снова
5. **Лямбда-исчисление** - механизм «анонимных функций» - callback-функций, которые можно определить прямо в том месте, где они используются, и которые имеют доступ к локальным переменным текущей функции
6. **Иммутабельность** - свойство объектов не изменяться (н-р строка. строку нельзя поменять на месте, для изменения нужно создать новую строку). Принцип иммутабельности в ФП должен применяется для другими страктурами данных (списки, словари). Иммутабельный список - список в который нельзя добавить / удалить элемент, можно только итерироваться по ним. Можно создать новый список который содежит все элементы +1, при этом старый остается валадным
7. **Замыкания**

# Композиция

* **Композиция и Агрегация** - расширение функционала класса за счет "внедрения" других классов
* Об агрегировании также часто говорят как об «отношении принадлежности» по принципу «у машины есть корпус, колёса и двигатель»
* **Агрегация** (агрегирование по ссылке) - отношение «часть-целое» между двумя равноправными объектами, когда один объект (контейнер) имеет ссылку на другой объект. Оба объекта могут существовать независимо: если контейнер будет уничтожен, то его содержимое - нет
* **Композиция** (агрегирование по значению) - более строгий вариант агрегирования, когда включаемый объект может существовать только как часть класса. Если класс будет уничтожен, то и включённый объект тоже будет уничтожен. Объект не может выйти за рамки своего класса. Объект будет создаваться внутри класса при создании класса и будет уничтожаться перед уничтожением родительского класса
* **Функциональная композиция** - передача результа вызова одной функции в качестве аргумента другой функции



* Агрегация: профессора - факультеты, профессора остаются жить после разрушения факультета
* Композиция: университет - факультеты, факультеты без университета уничтожаются

# Каррирование

* **Каррирование** / Карринг (Currying) - в функциональном программирование - это преобразование функции с множеством аргументов в набор вложенных функций с одним аргументом. При вызове каррированной функции с передачей ей одного аргумента, она возвращает новую функцию, которая ожидает поступления следующего аргумента. Новые функции, ожидающие следующего аргумента, возвращаются при каждом вызове каррированной функции — до тех пор, пока функция не получит все необходимые ей аргументы. Ранее полученные аргументы, благодаря механизму замыканий, ждут того момента, когда функция получит всё, что ей нужно для выполнения вычислений. После получения последнего аргумента функция выполняет вычисления и возвращает результат.
* При каррировании число вложенных функций равно числу аргументов исходной функции. Каждая из этих функций ожидает собственный аргумент. Если функция аргументов не принимает, или принимает лишь один аргумент, то каррировать её нельзя
* **Частичное применение функций** (Partial Function Application) - функцию преобразуют в другую функцию, обладающую меньшим числом аргументов. Некоторые аргументы такой функции оказываются зафиксированными (для них задаются значения по умолчанию)
* Каррирование и преобразования функций реализуются в JavaScript благодаря замыканиям и тому, что функции в JS являются объектами первого класса (их можно передавать в качестве аргументов другим функциям, возвращать из них, присваивать переменным)

# Наследование

* Класс - функция-конструктор вместе с её prototype

|  |  |  |
| --- | --- | --- |
| Прототипный стиль | Функциональный стиль | Классы |
|  |  |  |

# Object.create и Object.assign

|  |  |
| --- | --- |
| **Object.create** - добавление прототипа при создании объекта (создаёт пустой объект с \_\_proto\_\_, равным первому аргументу, второй содержит дескрипторы свойств) | **Object.assign** - используется для копирования значений всех собственных перечисляемых свойств (при обходе в цикле for..in его можно будет получить) из одного или более исходных объектов в целевой объект. После копирования он возвращает целевой объект |
|  |  |

# Конструктор new

1. Создаётся новый пустой объект через Object.create
2. Устанавливает \_\_proto\_\_ этому объекту в которое записывается ссылка из prototype функции-конструктора *(устанавливается прототип)*
3. this получает ссылку на этот объект. Модифицируется this (т.е. этот новый объект) и инициализируются методы и свойства.
4. Неявно возвращается this без вызова return.   
   *Возвращает экземпляр функции-класса, но если FnClass нам вернул обьект, тогда его*

# \_\_proto\_\_ и prototype

**\_\_proto\_\_** - свойство объекта Object.prototype, хранящее ссылку на свойство prototype конструктора

* При обращении к любому свойству объекта, оно в первую очередь ищется в самом объекте. Но если его там нет, поиск происходит в свойстве \_\_proto\_\_. Если его нет и там, оно ищется дальше по цепочке (prototype chain)
* \_\_proto\_\_ - наследование на уровне объектов

**prototype** - свойство функции. По умолчанию является объектом с единственным свойством constructor, которое ссылается на саму функцию

* **Используется:** при создании новых объектов оператором new
* **Цель:** указывать \_\_proto\_\_ для новых объектов. При создании объекта через new, в его прототип \_\_proto\_\_ записывается ссылка из prototypeфункции-конструктора

# Контекст: this, call, apply, bind

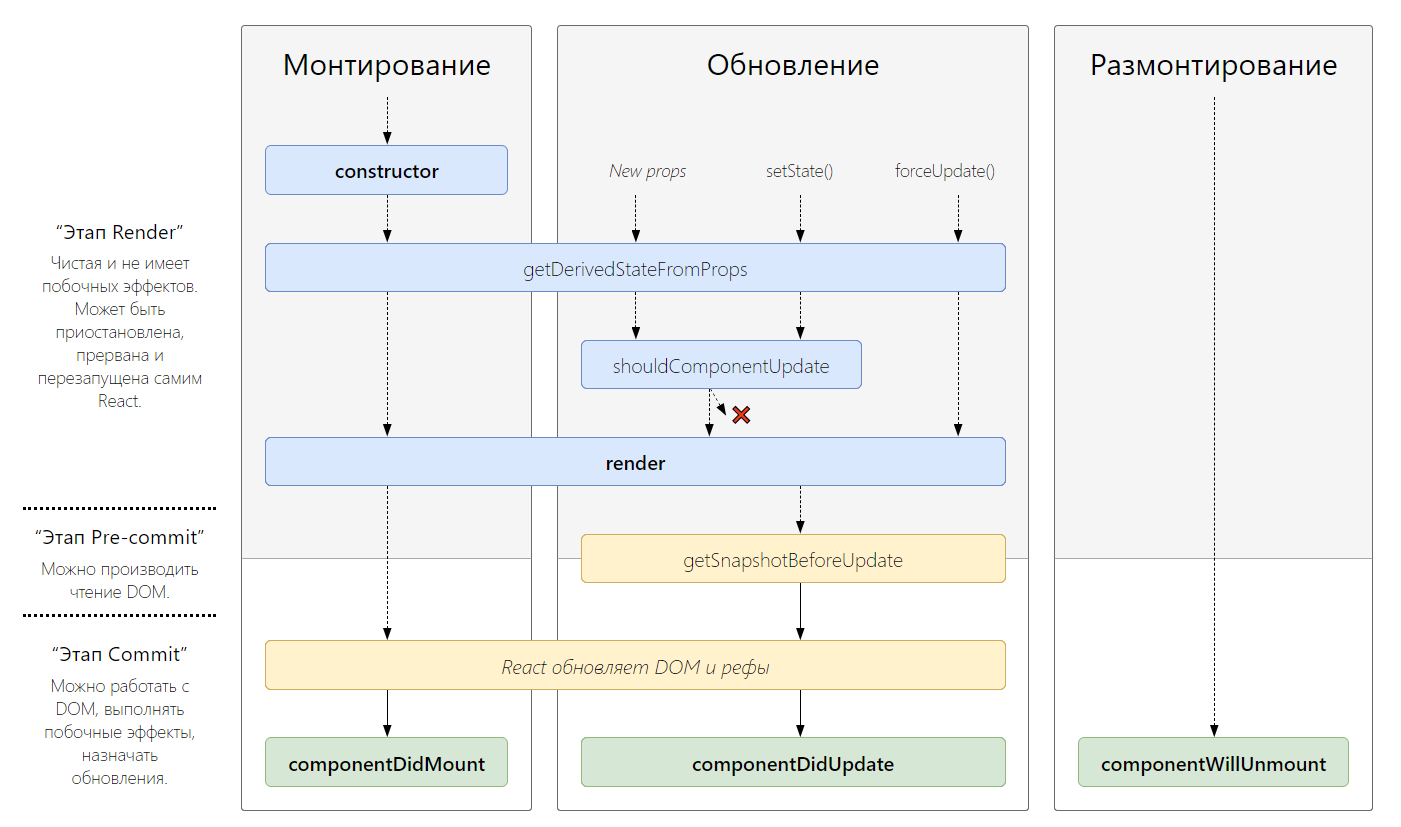
* **Контекст выполнения** (Execution Context) - служебная информация, которая соответствует текущему запуску функции. Она включает в себя локальные переменные функции и конкретное место в коде, на котором находится интерпретатор. У каждого вызова функции свой «контекст выполнения» и он всегда разный в зависимости от того как функция была вызвана
* Контекст бывает либо глобальным, с которого начинается исполнения скрипта, либо контекстом выполнения вызова, который начинается с момента входа в тело функции
* В каждый момент времени активен только один контекст выполнения. Именно поэтому Javascript называют “однопоточным”, имея ввиду, что только одна инструкция исполняется в один момент времени. Типичный браузер отслеживает контексты выполнения с помощью стека
* **Контекст вызова функции** - тот объект который ее вызывает
* **this** - ссылка на текущий контекст вызова (на текущий объект с которым произошло событие). Определяется в момент вызова функции

*Методы*

* **call**() явное указание контекст вызова, вызывает функцию с заданным контекстом и аргументами
* **apply**() аналогичен call(), только аргументы передаются массивом вторым аргументом
* **bind**() не вызывает функцию, а связывает с объектом (возвращает новую): возвращает «обёртку», которая передаст вызов в исходную функцию, с привязанным контекстом



# Жизненный цикл React-компонентов

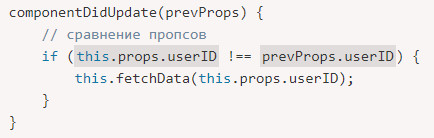


#### **1. componentDidMount()**

* МОНТИРОВАНИЕ (mounting) вызывается сразу после монтирования (то есть, вставки компонента в DOM). В этом методе должны происходить действия, которые требуют наличия DOM-узлов. Это хорошее место для создания сетевых запросов
* Вы можете сразу вызвать setState()

#### **2. componentDidUpdate**(prevProps, prevState, snapshot)

* ОБНОВЛЕНИЕ вызывается сразу после обновления (изменение пропсов или состояния). Не вызывается при первом рендере
* Метод позволяет работать с DOM при обновлении компонента
* Метод подходит для выполнения таких сетевых запросов, которые выполняются на основании результата сравнения текущих пропсов с предыдущими. Если пропсы не изменились, новый запрос может и не требоваться



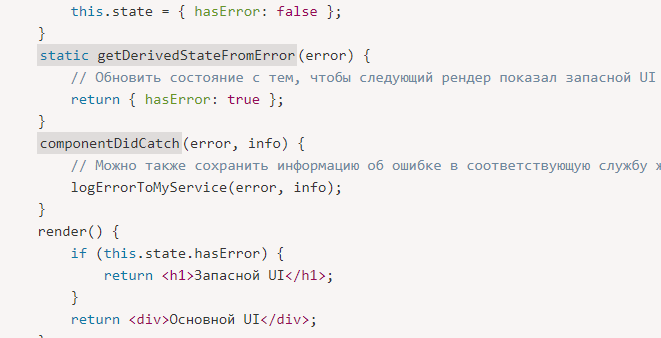
#### **3. componentWillUnmount()**

* РАЗМОНТИРОВАНИЕ (unmounting) вызывается перед размонтированием и удалением компонента (после удаления DOM-узла, созданного компонентом)
* В этом методе выполняется необходимый сброс: отмена таймеров, сетевых запросов и подписок, созданных в componentDidMount()
* Не используйте setState() в методе, так как компонент никогда не рендерится повторно

#### **componentDidCatch**(error, info - информация о компоненте, в котором произошла ошибка)

* ОБРАБОТКА ОШИБОК если произошла ошибка в процессе рендеринга, методе жизненного цикла или конструкторе любого дочернего компонента
* В случае ошибки вы можете рендерить запасной интерфейс с помощью getDerivedStateFromError()

***static getDerivedStateFromError(error)*** - вызывается после возникновения ошибки у компонента-потомка. Он получает ошибку в качестве параметра и возвращает значение для обновления состояния



***static getDerivedStateFromProps(props, state)***

Он должен вернуть объект для обновления состояния или null, чтобы ничего не обновлять. Существует для редких случаев, когда состояние зависит от изменений в пропсах. Например, это подойдёт для реализации компонента <Transition>, который сравнивает свои предыдущие и следующие дочерние компоненты, чтобы решить, какой из них нужно анимировать. Метод запускается при каждом рендере, независимо от причины

***shouldComponentUpdate(nextProps, nextState)***

для указания необходимости следующего рендера на основе изменений состояния и пропсов. По умолчанию происходит повторный рендер при любом изменении состояния. Вызывается перед рендером, когда получает новые пропсы или состояние. Необходим только для повышения производительности

***getSnapshotBeforeUpdate(prevProps, prevState)*** вызывается прямо перед этапом «фиксирования» (например, перед добавлением в DOM). Он позволяет компоненту брать некоторую информацию из DOM (например, положение прокрутки) перед её возможным изменением. Любое значение, возвращаемое этим методом жизненного цикла, будет передано как параметр componentDidUpdate(). Это применяется редко, но может быть полезно в таких интерфейсах, как цепочка сообщений в чатах, в которых позиция прокрутки обрабатывается особым образом

***render()*** - при вызове он проверяет this.props и this.state

* Функция render() должна быть **чистой**. Это означает, что она не изменяет состояние компонента, всегда возвращает один и тот же результат, не взаимодействует напрямую с браузером

Возвращает один из следующих вариантов

1. Элемент React. Обычно создаётся с помощью JSX. Указывает React, что рендерить: DOM-узел или пользовательский компонент. Например, <div /> или <MyComponent />
2. Массивы и фрагменты. Возвращает несколько элементов из render()
3. Порталы. Рендерит несколько дочерних элементов в другое поддерево DOM
4. Строки и числа. Рендерит текстовые DOM-узлы
5. Booleans или null. Ничего не рендерит. Обычно необходим для поддержки паттерна return test && <Child />, где test - логическое значение

***constructor(props)***

* Вы можете не использовать конструктор в React-компоненте, если вы не определяете состояние или не привязываете методы
* Конструктор компонента React вызывается до того, как компонент будет примонтирован. В начале конструктора необходимо вызывать super(props). Если это не сделать, this.props не будет определён. Это может привести к багам
* Не использовать setState() в constructor(). Начальное состояние сразу присваивается в конструкторе. Конструктор - единственное место, где можно напрямую изменять this.state. В остальных методах необходимо использовать this.setState()
* Не использовать побочные эффекты или подписки в конструкторе. Вместо этого используйте componentDidMount()

Обычно используют для двух целей:

* Инициализация внутреннего состояния через присвоение объекта this.state
* Привязка обработчиков событий к экземпляру

# React.js

***Higher-Order Component -* Компонент высшего порядка** функция, расширяющая функционал компонента без изменение его исходного кода. HOC не наследует поведение оборачиваемого компонента. HOC является чистой функцией без побочных эффектов

* connect в Redux

|  |
| --- |
| const EnhancedComponent = higherOrderComponent(WrappedComponent, SelectData);  // WrappedComponent - оборачиваемый компонент  // SelectData - функция, которая извлекает нужные данные |

**Действия**

* 1. Принимает оборачиваемый компонент (оборачивает оригинальный компонент в контейнер посредством композиции)
  2. Через пропсы передает ему новые данные
  3. Возвращает новый компонент с расширенной логикой
* **Для чего:** Есть 2 или несколько компонентов, логика которых совпадает. Чтобы не дублировать логику, можно использовать HOC

***React.PureComponent***

* React.PureComponent похож на React.Component. Отличие заключается в том, что React.Component не реализует shouldComponentUpdate(), а React.PureComponent реализует его поверхностным сравнением пропсов и состояния
* Если метод render() React-компонента всегда рендерит одинаковый результат при одних и тех же пропсах и состояниях, для повышения производительности в некоторых случаях вы можете использовать React.PureComponent
* Метод shouldComponentUpdate() базового класса React.PureComponent делает только поверхностное сравнение объектов. Если они содержат сложные структуры данных, это может привести к неправильной работе для более глубоких различий (то есть, различий, не выраженных на поверхности структуры). Наследуйте класс PureComponent только тогда, когда вы ожидаете использовать простые пропсы и состояние, или используйте forceUpdate(), когда знаете, что вложенные структуры данных изменились. Также подумайте об использовании иммутабельных объектов, чтобы упростить процесс сравнения вложенных данных
* Метод shouldComponentUpdate() пропускает обновление пропсов для всего поддерева компонентов. Убедитесь, что все дочерние компоненты также являются «чистыми»

#### Как сравнивает элементы PureComponent

* Простые типы данных - по значению
* Объекты - по ссылкам. Если ссылка на объект поменялась – перерендер

***Фрагменты*** - позволяют формировать список дочерних элементов, не создавая лишних узлов в DOM

* <React.Fragment>

***Порталы*** - позволяют рендерить дочерние элементы в DOM-узел, который находится вне DOM-иерархии родительского компонента

* Типовой случай применения порталов - когда в родительском компоненте заданы стили overflow: hidden или z-index, но вам нужно чтобы дочерний элемент визуально выходил за рамки своего контейнера. Например, диалоги, всплывающие карточки и всплывающие подсказки
* Событие, сгенерированное изнутри портала, будет распространяться к родителям в содержащем React-дереве, даже если эти элементы не являются родительскими в DOM-дереве

***Предохранители*** - компоненты React, которые отлавливают ошибки JavaScript в любом месте деревьев их дочерних компонентов, сохраняют их в журнале ошибок и выводят запасной UI вместо рухнувшего дерева компонентов. Предохранители отлавливают ошибки при рендеринге, в методах жизненного цикла и конструкторах деревьев компонентов, расположенных под ними

* getDerivedStateFromError, componentDidCatch

***Рефы*** дают возможность получить доступ к DOM-узлам или React-элементам, созданным в рендер-методе. Используется когда требуется императивно изменить дочерний элемент, обойдя обычный поток данных

* Рефы создаются с помощью React.createRef() и прикрепляются к React-элементам через ref атрибут

**Значение рефа отличается в зависимости от типа узла:**

* + HTML-элемент - свойство current созданного рефа в конструкторе с помощью React.createRef() получает соответствующий DOM-элемент. React присвоит DOM-элемент свойству current при монтировании компонента и присвоит обратно значение null при размонтировании. Обновление свойства ref происходит перед вызовом методов componentDidMount и componentDidUpdate
  + Классовый компонент - свойство current объекта-рефа получает экземпляр смонтированного компонента
  + Функциональный компонент - нельзя использовать. Для них не создаётся экземпляров. Можно использовать атрибут ref внутри функционального компонента при условии, что он ссылается на DOM-элемент или классовый компонент

***Подъём состояния*** - совместное использование состояния, которое достигается перемещением его до ближайшего предка компонентов, которым оно требуется

***Контекст*** - позволяет передавать данные через дерево компонентов без необходимости передавать пропсы на промежуточных уровнях

* Контекст разработан для передачи данных, которые можно назвать «глобальными» для всего дерева React-компонентов
* В типичном React-приложении данные передаются сверху вниз (от родителя к дочернему компоненту) с помощью пропсов. Однако, этот способ может быть чересчур громоздким для некоторых типов пропсов (например, выбранный язык, UI-тема), которые необходимо передавать во многие компоненты в приложении. Контекст предоставляет способ делиться такими данными между компонентами без необходимости явно передавать пропсы через каждый уровень дерева
* Усложняет переиспользование компонентов

# State

* Состояние (state) - содержит данные, специфичные для этого компонента. Они могут измениться со временем. Состояние определяется пользователем и должно быть простым объектом JavaScript
* Контролируется и доступно только конкретному компоненту. Этот процесс называется «нисходящим» («top-down») или «однонаправленным» («unidirectional») потоком данных. Состояние всегда принадлежит определённому компоненту, а любые производные этого состояния могут влиять только на компоненты, находящиеся «ниже» в дереве компонентов
* Нельзя менять состояние напрямую, иначе не происходит повторного рендер (так как более поздний вызов setState() может перезаписать эту мутацию). Конструктор - это единственное место, где можно присвоить значение this.state напрямую
* Не нужно вставлять в состояние значение, если оно не используется для рендера или потока данных (например, идентификатор таймера). Такие значения можно определить как поля экземпляра компонента
* **setState** - добавляет в очередь изменения в состоянии компонента. Также он указывает React, что компонент и его дочерние элементы должны быть повторно отрендерены с обновлённым состоянием. Этот метод используется для обновления интерфейса в ответ на обработчики событий и ответы сервера
  + ☰ updater - значения state для обновления
  + callback - гарантированно вызывается после того как было применено обновление

|  |  |
| --- | --- |
|  |  |

* Второй вариант вызова setState(), принимает функцию, а не объект. Эта функция получит предыдущее состояние в качестве первого аргумента и значения пропсов непосредственно во время обновления в качестве второго аргумента

# Virtual DOM

* Virtual DOM - техника и набор библиотек / алгоритмов, которые позволяют нам улучшить производительность на клиентской стороне, избегая прямой работы с DOM путем работы с легким JavaScript-объектом, имитирующем DOM-дерево
* Вместо того, чтобы взаимодействовать с DOM напрямую, мы работаем с его легковесной копией. Мы можем вносить изменения в копию, исходя из наших потребностей, а после этого применять изменения к реальному DOM. При этом происходит сравнение DOM-дерева с его виртуальной копией, определяется разница и запускается перерисовка того, что было изменено
* React создает легковесное дерево из JavaScript-объектов для имитации DOM-дерева. Затем он создает из них HTML, который вставляется или добавляется к нужному DOM-элементу, что вызывает перерисовку страницы в браузере
* JS объкты просто строить и сравнивать

#### 1. Первоначальный Render DOM

1. Вернули JSX (render return)
2. Транспиляция в React.createElement (Virtual DOM)
3. Рендер в браузер DOM (react-dom) ReactDOM.render

#### 2. Изменение setState()

* При нажатии setState() каждый раз асинхронно (React 16) перестраивается Virtual DOM для компонента и всех его потомков. Но в DOM идут только изменения

1. Произошел setState() в компонене App. Помечаем компонент как "грязный"
2. Перестроение Virtual DOM
3. Сравнение измененого Virtual DOM и старого Virtual DOM. При нахождении изменений, меняется компонент в реальном DOM