ИММУТАБЕЛЬНОСТЬ. ЧИСТЫЕ ФУНКЦИИ. МОДЕЛЬ MVC

JavaScript

Модуль 4. Урок 2.

LUXOFT

LXFT LISTED NYSE

ИММУТАБЕЛЬНОСТЬ

Самым простым и быстрым способом проверить, изменился ли объект, оказывается проверка ссылки на объект — изменилась она или нет.

В этом случае можно не делать детальное сравнение свойств, например такое:

```
_.isEqual(object1, object2)
```

Намного быстрее и проще в случае изменения объекта заменять его, а не редактировать. Тогда проверку можно максимально упростить:

```
object1 === object2
```

В этом и заключается основная идея «неизменяемости». Дело не в том, что невозможно изменить объект.

Можно просто следовать правилу неизменяемости: «Если нужно изменить какой-то объект, замените его».

ИММУТАБЕЛЬНОСТЬ МАССИВОВ

- Неизменяемость массивов:

  […array], concat, slice, splice, filter, map

```
let arr = [1,2,3];
let arr2 = arr; // таким образом мы создаем копию ссылки, и при внесении изменений
// будет меняться исходный массив, например:
arr2.push(10); // в результате изменился и arr: arr==[1,2,3,10]

let arr3 = […arr]; // создание копии массива
arr3.push(10); // теперь мы меняем только копию
console.log(arr); // [1,2,3]
console.log(arr3); // [1,2,3,10]
```

Для иммутабельного добавления элемента в массив мы можем просто использовать спред-оператор:

```
let arr4 = […arr, 10]; // иммутабельно добавляем в конец массива элемент 10: arr не меняется
let arr5 = [10, …arr]; // иммутабельно добавляем в начало массива элемент 10
```

ИММУТАБЕЛЬНОСТЬ ОБЪЕКТОВ

Обычные операции над объектами – например, присвоение значения свойству, меняют объект.
Для иммутабельного изменения мы можем использовать функцию Object.assign.

Object.assign(target, src1, src2...) – копирует все свойства из src1, src2 и т.д. в объект target.

```
let user = { name: "Вася" };
let visitor = { isAdmin: false, visits: true };
let admin = { isAdmin: true };

Object.assign(user, visitor, admin);

// user <- visitor <- admin
alert( JSON.stringify(user) ); // name: Вася, visits: true, isAdmin: true
```

ИММУТАБЕЛЬНОСТЬ ОБЪЕКТОВ

Но Object.assign() можно использовать для 1-уровневого (не глубокого!) клонирования объектов:

```
let user = { name: "Вася", isAdmin: false };

// clone = пустой объект + все свойства user
let clone = Object.assign({}, user);
```

Таким образом, мы получаем клон объекта user.

Мы можем использовать это для получения объекта, в котором изменено значение 1 или нескольких свойств. Например, нам нужно изменить значение name:

```
let user2 = Object.assign({}, user, {name: "Дима"});
```

Теперь user2 содержит значение {name: "Дима”, isAdmin: false }

При этом объект user остался неизменным.

Также мы можем изменить значения сразу 2 полей, например:

```
let user3 = Object.assign({}, user, {name: "Иван", isAdmin: true});
```

Теперь у нас получился новый объект user3, при этом user остался неизменным.

ИММУТАБЕЛЬНОСТЬ ОБЪЕКТОВ: СПРЕД ОПЕРАТОР

Также можно использовать спред оператор – новый синтаксис для создания копий (должен войти в EcmaScript 2018).

Тогда синтаксис создания копии объекта упрощается и становится похожим на синтаксис создания копии массива:

```
let user2 = {…user, name: "Дима" };
```

Теперь user2 содержит значение {name: "Дима”, isAdmin: false }

При этом объект user остался неизменным.

```
let user3 = {…user, name: "Иван", isAdmin: true};
```

Теперь у нас получился новый объект user3, при этом user остался неизменным.

Даже после повсеместного введения нового стандарта, останется много унаследованного кода, который продолжит использование метода Object.assign(). Поэтому необходимо знать и уметь применять оба варианта.

ЧИСТАЯ ФУНКЦИЯ

- Что такое чистая функция? Функция считается чистой, если она соответствует следующим утверждениям:
- При одинаковых аргументах результат вычисления будет одним и тем же. Никаких сюрпризов. Никаких побочных эффектов. Никаких вызовов API. Никаких изменений. Только вычисление.

- Примеры:

```
var values = {a: 1};
function impureFunction(items) {
   var b = 1;
   items.a = items.a * b + 2;
   return items.a;
}
var c = impureFunction(values)
```

```
var values = {a: 1};
function pureFunction(a) {
   var b = 1;
   a = a * b + 2;
   return a;
}
var c = pureFunction(values.a)
```

params − function −> return value

ПРЕИМУЩЕСТВА ИММУТАБЕЛЬНОСТИ

1. Простое и быстрое отслеживание изменений

Для того, чтобы понять, одинаковы ли объекты obj1 и obj2, необходимо провести их глубокое сравнение, что является дорогостоящей операцией.

Если же все преобразования были иммутабельны, в этом нет необходимости: если ссылки на объекты не совпадают, то и объекты – разные.

Пример: пусть у нас есть функция, ответственная за демонстрацию изменений

```
function showChanges(oldValue, newValue) {
  if (oldValue == newValue) return; // ничего не изменилось – можем ничего не делать
  // такой подход лежит в основе Virtual DOM и дает возможность экономить время

  // … логика отрисовки данных с учетом изменений
}
let oldValue = obj1; // сохраняем исходное значение
obj1.a = 10; // не иммутабельное изменение
showChanges(oldValue, obj1); //хотя объект изменился, ссылка осталась
                             // той же – в результате изменения не будут отображены
let obj1 = {…obj1, a:10}; // иммутабельное изменение obj1
showChanges(oldValue, obj1); // теперь мы знаем, что obj1 изменился
                             // и должен быть перерисован
```

ПРЕИМУЩЕСТВА ИММУТАБЕЛЬНОСТИ

2. Безопаснее использовать

Переданные в функцию данные могут быть случайно испорчены, и отследить такие ситуации очень сложно.

При использовании иммутабельности мы не боимся, что данные могут случайно измениться – если какая-то функция меняет данные, она создает их новую копию.

3. Легче тестировать

Так как чистые функции зависят только от входных параметров, мы можем протестировать различные комбинациями параметров, и быть уверенными, что функция не зависит от других частей системы.

4. Легче мемоизировать

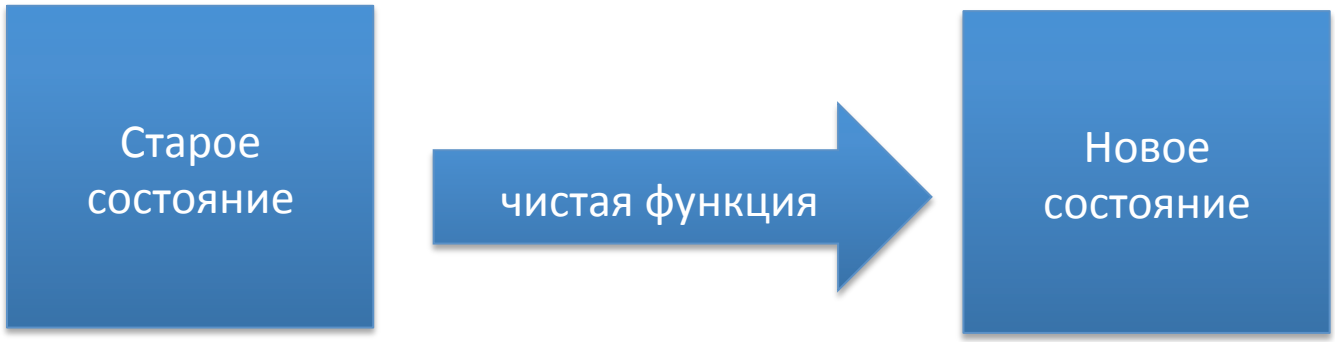
Мемоизация — сохранение результатов выполнения функций для предотвращения повторных вычислений. Это один из способов оптимизации, применяемый для увеличения скорости выполнения компьютерных программ. Перед вызовом функции проверяется, вызывалась ли функция ранее: если не вызывалась, функция вызывается и результат её выполнения сохраняется; если вызывалась, используется сохранённый результат.

При иммутабельных изменениях мемоизация становится тривиальной.

ПРЕИМУЩЕСТВА ИММУТАБЕЛЬНОСТИ

5. Легче отлаживать

Отладка – очень существенный вопрос для больших систем. Когда данные иммутабельны, гораздо легче отслеживать изменения состояний системы:



При этом мы можем видеть и отслеживать все изменения.
Таким образом, если происходит ошибка, ее гораздо проще отследить, потому что у нас есть вся полнота знаний о состоянии и преобразования, которые не содержат побочных эффектов.

ПРАВИЛА ХОРОШЕГО КОДА ПРИ НАПИСАНИИ ФУНКЦИЙ

ОГРАНИЧЕНИЕ КОЛИЧЕСТВА АРГУМЕНТОВ

Ограничение количества параметров функции невероятно важно, поскольку оно упрощает тестирование функции. Наличие более чем трёх аргументов приводит к комбинаторному взрыву, когда вам приходится перебирать массу различных случаев с каждым отдельным аргументом.

Плохо:

```
function createMenu(title, body, buttonText, cancellable) { … }
```

Хорошо:

```
const menuConfig = {
  title: 'Foo',
  body: 'Bar',
  buttonText: 'Baz',
  cancellable: true
};

function createMenu(config) { return config.title+","+config.body+","+config.buttonText }
createMenu(menuConfig);
```

Еще лучше – используем синтаксис ES2015:

```
function createMenu({title, body, buttonText, cancellable}) { return title + body + buttonText }
createMenu(menuConfig);
```

ФУНКЦИЯ ДОЛЖНА РЕШАТЬ ОДНУ ЗАДАЧУ

Когда функции решают более одной задачи, их труднее сочетать, тестировать и понимать. Как только вы сможете свести каждую функцию к выполнению только одного действия, их станет значительно проще рефакторить, а ваш код станет гораздо более читаемым.

Плохо:

```
function emailClients(clients) {
  clients.forEach((client) => {
    const clientRecord = database.lookup(client);
    if (clientRecord.isActive()) {
      email(client);
    }
  });
}
```

Хорошо:

```
function emailClients(clients) {
  clients
    .filter(isClientActive)
    .forEach(email);
}

function isClientActive(client) {
  const clientRecord = database.lookup(client);
  return clientRecord.isActive();
}
```

НАЗВАНИЯ ФУНКЦИЙ ДОЛЖНЫ ОПИСЫВАТЬ ИХ НАЗНАЧЕНИЕ

Плохо:

```
function addToDate(date, month) {
  // ...
}

const date = new Date();

// По имени функции трудно сказать,
// что именно добавляется
addToDate(date, 1);
```

Хорошо:

```
function addMonthToDate(month, date) {
  // ...
}

const date = new Date();
addMonthToDate(1, date);
```

ИЗБАВЛЯЙТЕСЬ ОТ ДУБЛИРОВАННОГО КОДА

Старайтесь избегать дублированного кода. Дублированный код вреден тем, что подразумевает наличие более чем одного места, в которое придется вносить правки, если логика действий изменится.

Зачастую дублированный код возникает в тех случаях, когда требуется реализовать два или более незначительно различающихся действия, которые в целом очень схожи, но их различия вынуждают вас завести две или более функции, делающих практически одно и то же. В этом случае избавление от дублированного кода будет означать создание абстракции, которая сможет представить все различия в виде одной функции, класса или модуля.

НЕ ИСПОЛЬЗУЙТЕ ФЛАГИ В КАЧЕСТВЕ ПАРАМЕТРОВ ФУНКЦИИ

Флаги говорят, что функция совершает более одного действия. Функция должна решать одну задачу. Разделяйте функции, если они исполняют различные варианты кода на основе логического значения.

Плохо:

```
function createFile(name, temp) {
  if (temp) {
    fs.create(`./temp/${name}`);
  } else {
    fs.create(name);
  }
}
```

Хорошо:

```
function createFile(name) {
  fs.create(name);
}

function createTempFile(name) {
  createFile(`./temp/${name}`);
}
```

Альтернатива – использование Map из функций:

```
// конфигурируем разные варианты
createFileMap = { true: createTempFile, false: createFile };
// получаем ссылку на функцию в зависимости от параметра temp
let createFileConfigured = createFileMap[temp];
// вызываем уже сконфигурированную функцию
createFileConfigured();
```

ОТДАВАЙТЕ ПРЕДПОЧТЕНИЕ ФУНКЦИОНАЛЬНОМУ ПРОГРАММИРОВАНИЮ НАД ИМПЕРАТИВНЫМ

Функциональные языки чище и их проще тестировать. Применяйте функциональный стиль программирования при возможности.

Плохо:

```
let totalOutput = 0;

for (let i = 0; i < programmerOutput.length; i++) {
  totalOutput += programmerOutput[i].linesOfCode;
}
```

Хорошо:

```
const totalOutput = programmerOutput
  .map((programmer) => programmer.linesOfCode)
  .reduce((acc, linesOfCode) => acc + linesOfCode, 0);
```

```
const programmerOutput = [
  {
    name: 'Uncle Bobby',
    linesOfCode: 500
  }, {
    name: 'Suzie Q',
    linesOfCode: 1500
  }, {
    name: 'Jimmy Gosling',
    linesOfCode: 150
  }, {
    name: 'Gracie Hopper',
    linesOfCode: 1000
  }
];
```

ИНКАПСУЛИРУЙТЕ УСЛОВИЯ

Плохо:

```
if (fsm.state === 'fetching' && isEmpty(listNode)) {
  // ...
}
```

Хорошо:

```
function shouldShowBanner(fsm, listNode) {
  return fsm.state === 'fetching' && isEmpty(listNode);
}

if (shouldShowBanner(fsmInstance, listNodeInstance)) {
  // ...
}
```

ИЗБЕГАЙТЕ УСЛОВНЫХ КОНСТРУКЦИЙ

Плохо:

```
class Airplane {
 // ...
 getCruisingAltitude() {
  switch (this.type) {
   case '777':
    return this.getMaxAltitude() -
              this.getPassengerCount();
   case 'Air Force One':
    return this.getMaxAltitude();
   case 'Cessna':
    return this.getMaxAltitude() -
              this.getFuelExpenditure();
  }
 }
}
```

Хорошо:

```
class Airplane {
 // ...
}

class Boeing777 extends Airplane {
 // ...
 getCruisingAltitude() {
  return this.getMaxAltitude() - this.getPassengerCount();
 }
}

class AirForceOne extends Airplane {
 // ...
 getCruisingAltitude() {
  return this.getMaxAltitude();
 }
}

class Cessna extends Airplane {
 // ...
 getCruisingAltitude() {
  return this.getMaxAltitude() - this.getFuelExpenditure();
 }
}
```

ИЗБЕГАЙТЕ ПРОВЕРКИ ТИПОВ

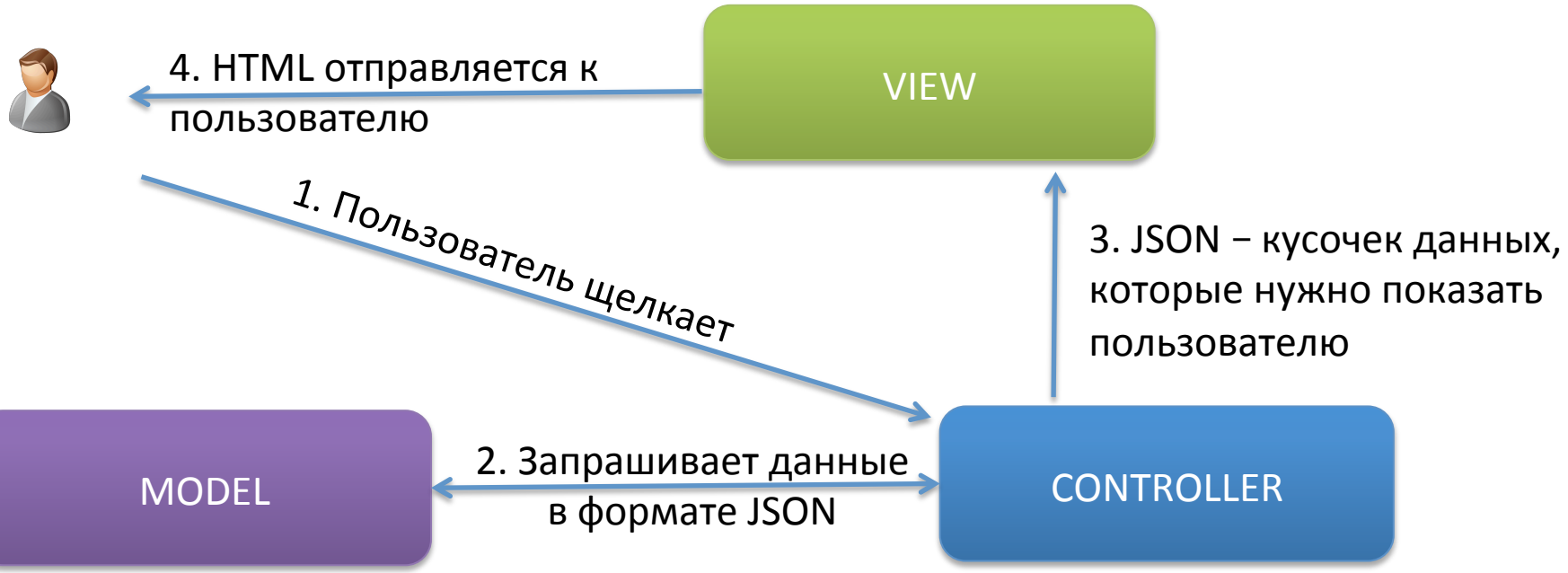
Плохо:

```
function travelToTexas(vehicle) {
  if (vehicle instanceof Bicycle) {
    vehicle.peddle(this.currentLocation, new Location('texas'));
  } else if (vehicle instanceof Car) {
    vehicle.drive(this.currentLocation, new Location('texas'));
  }
}
```

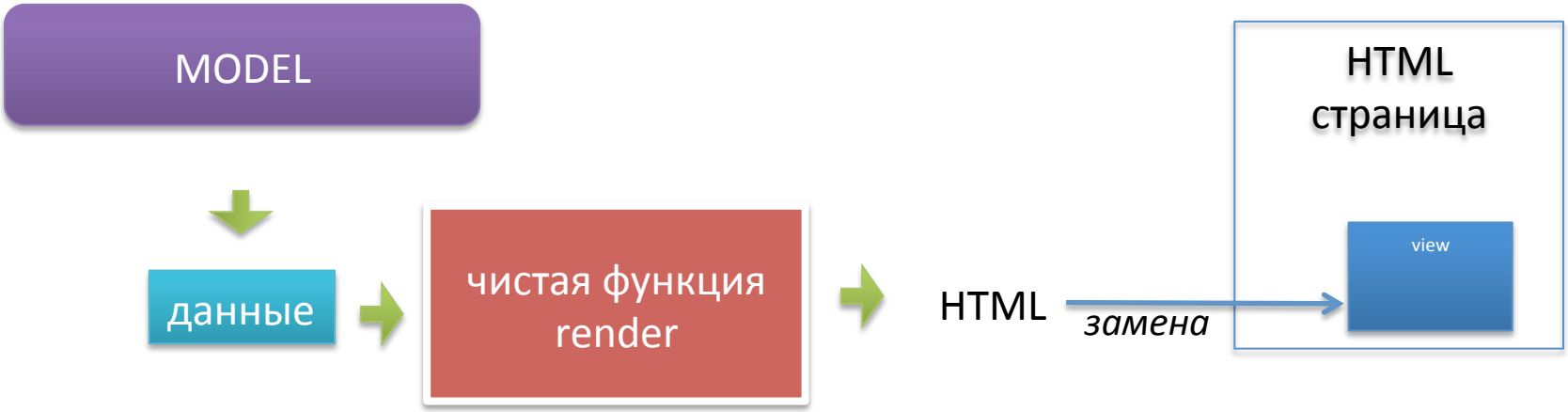
Хорошо:

```
function travelToTexas(vehicle) {
  vehicle.move(this.currentLocation, new Location('texas'));
}
```

МОДЕЛЬ MVC


4. HTML отправляется к пользователю
VIEW
1. Пользователь щелкает
3. JSON – кусочек данных, которые нужно показать пользователю
MODEL
2. Запрашивает данные в формате JSON
CONTROLLER

ИММУТАБЕЛЬНЫЕ ФУНКЦИИ ДЛЯ VIEW



employee -> HTML

```
function renderEmployee(employee) {
  return `<li>${employee.name} ${employee.surname}
            <button onclick="removeEmployee(${employee.id})">удалить</button>
          </li>`;
}
```

employees -> HTML

```
function renderEmployees(employees) {
  return "<ul>"+employees.map(e=>renderEmployee(e)).join('')+"</ul>";
}
```

ПРИМЕР: УДАЛЕНИЕ ИЗ СПИСКА СОТРУДНИКОВ