[Обучение Ext JS] :: Часть 13. Повторное использование кода: Расширяем Ext JS

Мар 19 2010

В этой главе мы обсудим, как можно создавать свои собственные компоненты, расширяя библиотеку **Ext JS**. Мы поговорим о том как создать собственные пространства имен, различая наши собственные компоненты от остальных. Мы также обсудим некоторые основные объектно-ориентированные концепты (просто для того чтобы понять что мы тут забыли) и концепт архитектуры управляемого событиями приложения.

Объектно-ориентированный JavaScript
на протяжении последних нескольких лет мы видели сильные изменения связанные со скриптами клиентских браузерных веб-приложений. Фактически **JavaScript** стал стандартом клиентского скриптописания, которую поддерживает любой мало-мальски крупный браузер.
Проблема всегда заключалась в том, как каждый браузер использует объектную модель документов (**Document Object Model**). Internet Explorer Microsoft'а занимает большую часть рынка браузеров помог развиться современной **DOM**, к чему пришлось адаптироваться другим браузерам. Тем не менее после выхода 6-й версии Internet Explorer, Microsoft остановила разработку их браузера на несколько лет для того чтобы разобраться с проблемами его безопасности. Стоит сказать что это было похоже на игру, Microsoft испытывала и создавала новые стандарты. Вместо введения **JavaScript**, Internet Explorer на самом деле ввел JScript, который запускал файлы **JavaScript** но имел немного другое применение и никогда не копил движения, возможно потому что не придерживался стандартов ECMAScript. Это породило несколько проблем. Консорциум всемирной сети (The World Wide Web Consortium (W3C)) создал стандарт для объектной модели документов, которого стали придерживаться такие компании как Mozilla и Opera adhered. Впрочем, Internet Explorer (самый популярный) остался безучастным.
Эти события стали причиной очень тяжелых дней (и лет) для клиентских разработчиков, так как очень много времени уходило на создание кросс-браузерного кода. Netscape и Internet Explorer вели браузерную войну напротяжении нескольки версий, медленно, но верно все дальше расходясь в своих взглядах напринятие стандартов. Все изменился настолько, что большая часть клиентских разработок стала очень низкосортными и состояли из простейших подтверждений и нескольких картинок, поскольку разработчики были вынуждены тратить время на написание огромных, жалующихся на кроссбраузерность, приложений.
Появился **Web 2.0**. В моду вошло слово **AJAX**, или асинхронный **JavaScript + XML** технология. **AJAX** не был новой разработкой, но использовался он редко, а его назначение для многих было загадкой. Разработчики с глубокими знаниями **JavaScript**, желавшие создавать хорошие динамические веб-сайты начали применять эту технологию совместно с оживлением клиентского движения, создавая наполненные и интерактивные пользовательские интерфейсы.
С возобновленным интересом к клиентским заскриптованным приложениям, но с теми же самыми проблемами браузерной совместимости начали свою жизнь библиотеки **JavaScript** (Dojo, Prototype, Yahoo UI и так далее) с целью уменьшить количество жалоб на проблемы совместимости. Разработчики этих библиотек с их знанием **JavaScript** старались идти в ногу с изменениями в этом языке. **JavaScript** проделал многое за гранью просто языка скриптописания. При создании своих библиотек эти разработчики пользовались всеми преимуществами которые давала модель **JavaScript**, основанная на прототипах, создавали маленькие, но функциональные объекты которые можно было накладывать один на другой, что давало огромные ресурсы для многократно используемого кода.

ВАЖНО: *На последующих страницах я буду часто использовать слова объект и класс, часто меняя их местами, потому, что так проще понимать. Технически, в **JavaScript** нет классов и наши объекты встроены в модель прототипного программирования **JavaScript**.*

Объектно-ориентированное программирование с Ext JS
Ext JS является отличным примером такого сдвига. Библиотека **Ext JS** невероятно обширная коллекция пакетов классов многократно используемого кода: маленькие кусочки функционала могут использоваться сами по себе или совмещаться для создания особой магии. Дополнением к ее красоте является возможность достраивать библиотеку, создавая собственные компоненты как расширения к тем что уже есть. Это дает нам возможность создания собственного многократно используемого кода. Мы можем написать наши собственные объекты как расширения к библиотеке. Это значит, что не придется писать весь функционал, так как вся работа уже сделана за нас. Мы будем строить на предоставленном фундаменте, создавать свой функционал.

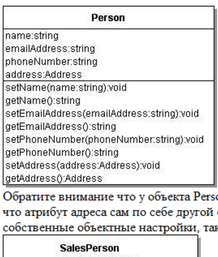
Наследование
Для того чтопы полностью понять что же нам предстоит сделать, нам надо понять ключевые моменты объектно ориентированного программирования — наследование.
Так как мы пишем наши собственные компоненты, то они обычно наследуют некоторые другие компоненты **Ext JS** расширяя функционал этих компонентов, предоставляя другие настройки и переписывая существующие методы. Мы создаем новые классы из существующего в **Ext JS** класса для определения какой метод был вызван для нашего нового класса: новый, какой-то из существующих (обычно родительский) или смешение двух предыдущих случаев.

Разбейте чтобы сделать проще
Все еще ошеломлены? Да, надо многое осмыслить. Давайте разобьем это на простые примеры которые могут кое-что разъяснить. Предположим мы пишем для компании систему управления взаимоотношениями с клиентами и партнёрами. У этой компании есть продавцы, к которых есть клиенты (и их контактные данные), а также у компании есть торговцы, которых они используют, у которых также есть контактные данные. Итак мы определили три похожих типа объектов в нашем приложении. Если мы спустим эти объекты на базовый уровень, то обнаружим, что каждый из этих объектов является Person (Clients и Vendors скорее всего будут Companies):

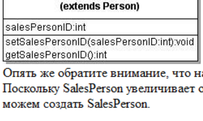
- Продавец
- Контактные данные клиента
- Контактные данные торговца

Каждый их этих объектов Person имеет общие атрибуты и методы, поскольку у каждого человека есть имя, email, телефонный номер и адрес. Мы наскоро создали диаграмму класса для визуальной демонстрации объекта Person:

ВАЖНО: *Диаграма класса и **UML** вообще, являются отличным способом для визуализации различных программных решений. Полное объяснение вне компетенции этого текста, но я настоятельно рекомендую вам ознакомиться с этим. Наши диаграммы показывают название класса, атрибуты и методы класса.*



Обратите внимание что у объекта Person имеется четыре атрибута: name, emailAddress, phoneNumber и address. Также обратите внимание, что атрибут адреса сам по себе другой объект. Мы включили для этого объекта довольно простой метод. У каждого из этих объектов есть собственные объектные настройки, такие как salesPersonID, clientContactID или vendorContactID.



Опять же обратите внимание, что на нашей диаграмме объекта SalesPerson вы не увидите никаких особых свойств или методов. Поскольку SalesPerson увеличивает объект Person все свойства и методы объекта Person стали частью объекта SalesPerson. Теперь мы можем создать SalesPerson.

```
var sp = new SalesPerson();
```
Это создает новый пример объекта SalesPerson, со всеми атрибутами и методами объекта SalesPerson, включая те что характерны для родительского объекта Person.

Круто звучит, но что все это значит?
По умолчанию объект SalesPerson является расширением Person, позволяя установить то название SalesPerson как:
SalesPerson.setName('Zig Ziggler')
Вам не надо делать ссылки методов или атрибутов вызывая родительский объект, в основном по причине наследования SalesPerson черт Person.

К чему было все это переписывание?
Через наследование все методы и атрибуты родительского объекта переходят к дочернему. Впрочем, могут быть случаи при которых вы можете захотеть изменить какие-нибудь из них. Например: представим что у объекта Person есть собственный метод validate(), который проверяет все атрибуты возвращается в собственый массив ошибок, но у вашего объекта SalesPerson также есть пара дополнительных атрибутов для проверки. Определяя метод validate() в объекте SalesPerson, вы переписываете родительский объект Person.

```
validate: function() {
  // Some validation code here
}
```
Но в этом случае вы захотите всеобщую проверку атрибутов. И у внутренних объектных свойств и тех что есть у родительского объекта. Так что придется вызывать метод validate() для родительского объекта:

```
validate: function() {
  var errorArr = ourObjects.salesperson.superclass.validate.
call(this);
  // The salesperson specific validate stuff, appending the errorArr
  return errorArr;
}
```
Это основные принципы ОО которые необходимо понять прежде чем начать создавать собственные классы с **Ext JS**. Ну, почти...

Понимание пакетов, классов и пространств имен
Нам понадобятся несколько заключительных кусочков объектно ориентированной головоломки для четкого понимания наших целей. Мы говорили о том как такая большая коллекция объектов **Ext JS** помогает расширять другие объекты, но также важно понять несколько других частей терминологии ОО (Объектно-Ориентированной) и как они помогут все организовать.

Пакеты
Пакетами называют собрания классов у которых есть что-то общее. Например, пакет Ext.data это коллекция классов имеющих дело с данными, различными типами хранилищ, читателей и записей. Пакет Ext.grid является коллекцией классов для разнообразны объектов таблиц, включая различные типы таблиц и модели выбора. Подобным образом пакет Ext.form содержит классы отвечающие за создание форм, включая все эти классы в различные типы поля.

Классы
Класс, это то как мы называем определенный объект JavaScript, определяя его отрибуты и методы. Возвращаясь к предыдущим примерам, Person и SalesPerson будут записаны как объекты класса и оба, возможно, будут частью одного пакета.

Пространства имен
Классы являются частью пакетов, а пакеты, в основном, имеют собственные пространства имен. Пространства имен это контейнеры логически сгруппированных пакетов и объектов класса. В качестве примера, библиотека **Ext JS** разделяется на пространства имен **Ext**. Формы в **Ext JS** разделяются на пространства имен Ext.forms, где формы это пакеты различных классов использованных для создания форм. Это иерархические отношения, где используется знак "точка" для разделения пространства имени пакета от класса. Перенесенные из одного пространства имени должны передаваться в другое, таким образом применение пространства имени помогает сжимать информацию. Ext.grid, Ext.form, и Ext.data являются пользовательскими пространствами имен.

Что дальше?
Теперь, покончив с терминологией и ознакомившись с основами объектно-ориентированных концептов мы переходим к применению полученных знаний в контексте создания собственных компонентов **Ext JS**.