1. Что такое Свойства зависимости? Для чего они нужны?

**Свойства** **зависимостей** расширяют функциональные возможности базовых **свойств** среды выполнения Windows, предоставляя глобальное внутреннее хранилище для резервных копий **свойств** **зависимостей** в приложении во время выполнения. **Свойства** **зависимостей** — это альтернативный способ стандартного резервирования **свойства** при помощи частного поля (частного в классе определения **свойств**). Dependency

2. Как создать DependencyProperty?

Они фактически скрывают **свойства зависимостей** или **dependency property**. Без свойств зависимостей были бы невозможны многие ключевые особенности WPF, как привязка данных, стили, анимация и т.д.

Рассмотрим, как они определяются. Возьмем, к примеру, элемент TextBlock, у которого есть свойство **Text**:

Статическое свойство TextProperty является свойством зависимостей, представляя объект **System.Windows.DependencyProperty**. По соглашениям по именованию все свойства зависимостей представляют статические публичные поля (public static) с суффиксом *Property*.

Затем в статическом конструкторе класса происходит регистрация свойства с помощью метода **DependencyProperty.Register()**, в который передается ряд параметров:

* **имя свойства** (в данном случае "Text"). Как правило, соответствует названию свойства зависимостей без суффикса *Property*
* **тип свойства** (в данном случае string)
* **тип, который владеет свойством** - собственно тот тип, в котором свойство определено или в данном случае тип TextBlock
* Необязательный параметр **FrameworkPropertyMetadata** устанавливает дополнительные настройки свойства
* В качестве пятого необязательного параметра может использоваться ссылка на метод, который производит валидацию свойства. В данном случае этот параметр опущен.

3. Для чего и как используют делегат ValidateValueCallback?

* **ValidateValueCallback**: делегат, который возвращает true, если значение проходит валидацию, и false - если не проходит

4. Для чего и как используют делегат CoerceValueCallback?

* **CoerceValueCallback**: делегат, который может подкорректировать уже существующее значение свойства, если оно вдруг не попадает в диапазон допустимых значений

5. Какие типы маршрутизируемых событий есть в WPF (поясните каждый)?

**Маршрутизируемое событие** — это тип события, которое может вызывать обработчики для нескольких слушателей в дереве элементов, а не только для объекта, вызвавшего событие. Это в основном событие CLR, которое поддерживается экземпляром класса Routed Event. Он зарегистрирован в системе событий WPF. RoutedEvents имеют три основные стратегии маршрутизации, которые заключаются в следующем —

* Прямое событие
* Бублинг событие

Туннельное Событиеак, маршрутизируемые события делятся на три вида:

* **Прямые** (direct events) - они возникают и отрабытывают на одном элементе и никуда дальше не передаются. Действуют как обычные события.
* **Поднимающиеся** (bubbling events) - возникают на одном элементе, а потом передаются дальше к родителю - элементу-контейнеру и далее, пока не достигнет наивысшего родителя в дереве элементов.
* **Опускающиеся, туннельные** (tunneling events) - начинает отрабатывать в корневом элементе окна приложения и идет далее по вложенным элементам, пока не достигнет элемента, вызвавшего это событие.

6. Как создать RoutedEvent?

7. Поясните концепцию Command в WPF? В чем ее преимущества?

**Команды** представляют механизм выполнения какой-нибудь задачи, например, копирования текста - когда мы нажимаем Ctrl+C, то мы копируем текст в буффер. В процессе копирования выполняется ряд действий, и все вместе эти действия объединяются в одну команду. Использование команд помогает нам сократить объем кода и использовать одну и ту же команду для нескольких элементов управления в различных местах программы. Таким образом, команды позволяют абстрагировать набор действий от конкретных событий конкретных элементов.

В некотором роде команды в WPF являются реализацией общераспространенного паттерна [Команда](http://metanit.com/sharp/patterns/3.3.php).

Модель команд в WPF состоит из четырех аспектов:

* Сама **команда**, которая представляем выполняемую задачу
* **Привязка команд**, которая связывает команду с определенной логикой приложения
* **Источник команды** - элемент пользовательского интерфейса, который запускает команду (например, кнопка, по нажатию который выполняется команда)
* **Цель команды** - элемент интерфейса, на котором выполняется команда

8. Как используются команды?

9. Как создать RoutedUICommand

В WPF определено много команд, но даже их может оказаться недостаточно. Поэтому нередко разработчики создают свои собственные команды. Наиболее простой способ создания команды - использование готовых классов **RoutedCommand** и **RoutedUICommand**, в которых уже реализован интерфейс ICommand.