> 由 KiOii (*_EM_Cpper*)整理。 （KiOii (*_EM_Cpper*) makes this note.）

**Properties and Bindings**
- What Is a Property?
- What Is a Binding?
- Understanding Bindings Support in JavaBeans
- Understanding Properties in JavaFX
- Using Properties in JavaFX Beans
- Lazily Instantiating Property Objects
- Understanding the Property Class Hierarchy
- Handling Property Invalidation Events
- Handling Property Change Events
- Avoiding Memory Leaks in Listeners
- Handling Invalidation and Change Events
- Using Bindings in JavaFX
- Unidirectional and Bidirectional Bindings
- Understanding the Binding API
  - The Hight-level Binding API
    - Using the Fluent API
      - The Binding interface
      - The NumberBinding Interface
      - The ObservableNumberValue Interface
      - The ObservableIntegerValue Interface
      - The NumberExpression Interface
      - The StringBinding Class
      - The ObjectExpressionn and ObjectBinding Classes
      - The BooleanExpression and BooleanBinding Classes
        - Using Ternary Operation in Expressions
        - Using the Bindings Utility Class
      - Combining the Fluent API and the Bindings Class
      - Using the Low-level Binding API
- Using Bindings to Center a Circle

# Properties and Bindings

> 属性和绑定

**In this chapter**

- JavaFX中的Property是什么?
- 怎么创建property object并使用?
- The class hierarchy of properties in JavaFX（JavaFX中的属性的继承层次结构？）
- 如何处理property object中的inavalidation 和 change events
- JavaFX中的binding是什么？怎么去使用单向绑定和双向绑定（unidirectional and bidirectional）
- JavaFX的高层次和低级绑定API

## What Is a Property?

- **define:** a Java class that has public accessors, for all or part of its private fields, is known as a Java bean, and the accessors define the properties of the bean

  > Property : public accessors define the property
  >
  > Bean : class that...
  >
  > 通过属性，我们可以定制bean的状态和行为
  >
  > **accessor--->property（state,beavior)**

- Java beans are **observable**
  - 支持属性更改通知

    > 当一个public property of Java bean改变时，一个通知被发送到所有监听器

- **Java bean**
  - define reusable components (定义可重用组件)

    > 这些组件可以通过builder tool去组装然后创建一个Java application
    >
    > 因此我们可以使用第三方 java bean

  - **visual or nonvisual**

- **wraning :** A property can be read-only,write-only,ro read/write
  - read-only: getter but no setter
  - write-only: setter but no getter

    > Java ide和其他构建工具（例如，GUI布局构建器），使用内省(introspection)来获得一个bean的一个属性列表，让您在设计时操作这些属性

- **use :** Properties of a bean can be used in a builder tool or programmatically

  > 构建器和编程都可以使用属性

**JavaBeans API**

- provides a class library
- `java.beans` package
- naming conventions(命名规则)
- wraning : **用于创建 Java bean**

## What Is a Binding?

- use : binding is used in **many different contexts**
  - **context of data binding** : defines a relation between data elements

    > 定义了数据元素之间的一种关系，以此来保持它们的同步

    - example：  x = y + z 建立了x，y，z之间的绑定，当运行时，x的值同步到y和z的和

      > 当然，绑定只是这时有效，之前或者之后，不一定同步

    - 让有效期持续一段时间：  在绑定部分改变时，我们需要得到改变的通知
      - mechanism : **register listeners** with the dependencies(依赖部分)

> 当dependencies失效或者改变时，所有挂接的listeners被通知
>
> A binding may synchronize itself with its dependencies when it receives such notifications

- **eager or lazy binding**
  - eager binding:依赖部分改变时，bound variable马上被计算

    > lazy binding则不会马上执行计算，而是在下次读取时进行计算，需要再计算，表现得比eager更好
- **undirectional or bidirectional**
  - undirectional： works only in one direction

    > 依赖项的变化传播到bound variable
  - bidirectional：依赖项和bound variable互相保持同步

    > 被定义使用在两个变量之间，例如：x = y and y = x、
    >
    > 例如：The data displayed in UI widget have to be synchronized with the underlying data model and vice versa
    >
    > UI和数据模型的同步需要是双向的

## Understanding Bindings Support in JavaBeans

- Java一直支持bean property之间的绑定
- example : two properties (name and salary)

```
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;
public class Employee
{
    private String name;
    private double salary;
    private PropertyChangeSupport pcs = new PropertyChangeSupport(this);

    public Employee()
    {
        this.name = "John Doe";
        this.salary = 1000.0;
    }
    public Employee(String name, double salary)
    {
        this.name = name;
        this.salary = salary;
    }
    public String getName()
    {
        return name;
    }
    public void setName(String name)

    {
        this.name = name;
    }
    public double getSalary()
    {
        return salary;
    }
    public void setSalary(double newSalary)
    {
        double oldSalary = this.salary;
        this.salary = newSalary;
        // Notify the registered listeners about the change
        pcs.firePropertyChange("salary", oldSalary, newSalary);
    }
    public void addPropertyChangeListener(PropertyChangeListener listener)
    {
        // 监听
        pcs.addPropertyChangeListener(listener);
    }
    public void removePropertyChangeListener(PropertyChangeListener listener)
    {
        // 取消监听
        pcs.removePropertyChangeListener(listener);
    }
    @Override
    public String toString()
    {
        return "name = " + name + ", salary = " + salary;
    }
}
```

> salary属性改变被监听

- **PropertyChangeSupport** class： 属性改变支持

> 支持属性修改时的：点燃属性改变、增加属性改变监听、取消属性改变监听

- **PropertyChangeListener** interface : 属性改变监听接口

> 该接口需要一个方法实现用于处理 属性改变的通知（firePropertyChange的发送）

- OK,知道了如果去使用ProperChangeSuppor让salary property变成bound property，现在我们需要知道如何相应这种改变

```
import java.beans.PropertyChangeEvent;
public class EmployeeTest
{
        public static void main(String[] args)
        {
            final Employee e1 = new Employee("John Jacobs", 2000.0);
            // Compute the tax
            computeTax(e1.getSalary());
            // Add a property change listener to e1
            e1.addPropertyChangeListener(EmployeeTest::handlePropertyChange);

            // Change the salary
            e1.setSalary(3000.00);
            e1.setSalary(3000.00); // No change notification is sent.
            e1.setSalary(6000.00);
        }
        public static void handlePropertyChange(PropertyChangeEvent e)
        {
            String propertyName = e.getPropertyName();
            if ("salary".equals(propertyName)) {
            System.out.print("Salary has changed. ");
            System.out.print("Old:" + e.getOldValue());
            System.out.println(", New:" + e.getNewValue());
            computeTax((Double)e.getNewValue());
        }
        public static void computeTax(double salary)
        {
            final double TAX_PERCENT = 20.0;
            double tax = salary * TAX_PERCENT/100.0;
            System.out.println("Salary:" + salary + ", Tax:" + tax);
        }
}
```

- PropertyChangeEvent class： 属性改变事件，涉及属性的一些信息

  > 例如：getPropertyName可以获取属性名字
  >
  > getOldValue() 可以获取旧值
  >
  > getNewValue()可以获取新值

# Understanding Properties in JavaFX

### Property

- *define :* In JavaFX,property are objects

  > 属性是对数据的封装实例对象(represent a value or a collection of values)
  >
  > 属性的抽象类(abstract class)：IntegerProperty、DoubleProperty、StringProperty、...
  >
  > 属性的具体实现类(concrete class)：有读写属性类和只读属性包装器
  >
  > 例如：
  >
  > SimpleInterProperty : int读写属性类
  >
  > ReadOnlyDoubleWrapper : double只读属性类

- *example:* `IntegerProperty counter = new SimpleIntegerProperty(100);`
- **get** and **set** :只有在primitive属性时，涉及参数和返回值均 primitive type
- **getValue** and **setValue** :涉及参数和返回值均 object type

  > 例如：对于String这种reference type property，get和getValue均String

### Working with read-only property is a bit tricky

> 使用只读属性有点棘手

- **ReadOnlyXXXWrapper** class: wraps two properties of XXX type

> one read-only and one read/write
>
> both properties are synchronized
>
> 两者是同步的

- **gerReadOnlyProperty** method: returns a **ReadOnlyXXXProperty** object

  ```
  ReadOnlyIntegerWrapper idWrapper = new ReadOnlyIntegerWrapper(100);
  ReadOnlyIntegerProperty id = idWrapper.getReadOnlyProperty();
  System.out.println("idWrapper:" + idWrapper.get());
  System.out.println("id:" + id.get());
  // Change the value
  idWrapper.set(101);
  System.out.println("idWrapper:" + idWrapper.get());
  System.out.println("id:" + id.get());
  ```

```
//prints
idWrapper:100
id:100
idWrapper:101
id:101
```
```

> 当wrapper property被修改时，read-only property object of the wrapper class自动修改，因此wrapper property一般用于private instance variable of a class



> XXXProperty
>
> ReadOnlyXXXProperty
>
> ReadOnlyXXXWrapper
```

### Information of Property object

- *wraning:* A property object wraps three pieces of information (属性包装了三个信息)
  - The reference of the bean that contains it （这边假设this的意义)
  - A name
  - A value

  > 因此，当你创建属性对象时，你可以提供以上的信息

  ```
  SimpleIntegerProperty()
  SimpleIntegerProperty(int initalValue)
  SimpleIntegerProperty(Object bean,String name)
  SimpleIntegerProperty(Object bean,String name,int initialValue);
  // 默认值取决于类型: 0、false | true 、null
  ```

  > 一个属性对象可以是bean的一部分，也可以是一个独立的（standlone）对象
  >
  > 针对standlone，一般传递实参null给bean，默认是null

- *wraning:*
    - **getBean** method : return the bean reference
    - **getName** method : return the property name

## Using Properties in JavaFX Beans

### SimpleXXXProperty

```
public class Person
{
    private StringProperty name = new SimpleStringProperty(this,"name","Li");
}

public class Book
{
   private StringProperty title = new SimpleStirngProperty(this."title","Unknonw");
    public final StringProperty titleProprty()
    {
        return title;
    }
}


public class Book
{
    private StringProperty title = new SimpleStringProperty(this, "title",  "Unknown");
    public final StringProperty titleProperty()
    {
        return title;
    }
    public final String getTitle()
    {
        return title.get();
    }
    public final void setTitle(String title)
    {
        this.title.set(title);
    }
}
```

> 按照规范，我们将getter and getter设置为final

### ReadOnlyXXXWrapper

```
public class Book
{
    private ReadOnlyStringWrapper ISBN = new ReadOnlyStringWraper(this,"ISBN","Unknonw");

    public final String getISBN()
    {
        return ISBN.get(); // value
    }
    publi final ReadOnlyStringProterty ISBNProperty()
    {
        return ISBN.gerReadOnlyProperty(); // read-only property
    }
    // More code goes here...
}
```

- Notice points
    - 使用了ReadOnlyStringWrapper代替了SimpleStringProperty
    - 没有设置setter，要设置的话必须指定为private final method，这样就不会暴露给外部
    - getter跟read/write property做了同样的工作
    - ISBNProperty返回ReadOnlyStringProperty

## Lazily Instantiating Property Objects

> 延迟实例化对象属性(因为有些高级feature没有使用到，或者只要默认值就行了)

- *power*： observable and binding features
- *define :* 延迟实例化，优化内存使用

  > 这种优化的代价是增加了几行代码

- *two use cases :*
    - property will use its default value in most of the cases

      > 属性大多数情况下使用默认值

      ```
      public class Monitor
      {
          public static final String DEFAULT_SCREEN_TYPE = "flat";
          private StringProperty screenType;
          public String getScreenType()
          {
          return (screenType == null) ? DEFAULT_SCREEN_TYPE : screenType.get();
          }
          public void setScreenType(String newScreenType)
          {
              if (screenType != null ||   !DEFAULT_SCREEN_TYPE.equals(newScreenType))
              {
                  screenTypeProperty().set(newScreenType);
              }
          }
          public StringProperty screenTypeProperty()
          {
              if (screenType == null) {
              screenType = new SimpleStringProperty(this, "screenType",
              DEFAULT_SCREEN_TYPE);
              }
              return screenType;
          }
      ```

```
}

//
Monitor m = new Monitor();
String st = m.screenTypeProperty().get(); // 实例化
//
Monitor m = new Monitor();
String st = m.getScreenType(); // 没有实例化
```

- property will not use its observable and binding features in most cases

  > 属性大多数情况下不使用observable and binding features

  ```
  public class Item
  {
      private DoubleProperty weight;
      private double _weight = 150;

      public double getWeight()
      {
          return (weight == null)?_weight:weight.get();
      }
      public void setWeight(double newWeight)
      {
          if (weight == null)
          {
              _weight = newWeight;
          }
          else
          {
              weight.set(newWeight);
          }
      }
      public DoubleProperty weightProperty()
      {
          if (weight == null)
          {
              weight = new SimpleDoubleProperty(this, "weight", _weight);
          }
          return weight;
      }
  }
  ```

## Understanding the Property Class Hierarchy

- **Observable** interface : wraps content,and it can be observed for invalidations of its content

  > Observable interface 通过两个 method 支持这个功能

  - **addListener** method : add an `InvalidationListener`
    - **invalidated** method  of InvalidationListener: 当content失效时被调用

- removeListener method : remove an `InvalidationListener`

> All JavaFX properties are observable
>
> 只有当属性从valid变成invalid时，才会产生一个失效事件
>
> valid to invalid 不是 change，例如排序ObservableList会产生invalidation event，但没有change

```
public interface Observable
{
    // ob content invalid event
    void addListener(InvalidationListener listener);
    void removeListener(InvalidationListener listener);
}
```

- **ObservableValue** interface : wraps a value,which can be observed for changes

  > inherits from the **Observable** interface

  - **getValue** method : returns the value it wraps
  - **invalidation events and change events** (扩展)
    - **invalidation** events : when the value in the ObservableValue is **no longer valid**
    - **change** events :when the **value changes**
  - addListener method and removeListener method overload
    - **changed** method :
      - three arguments : **reference** of ObservableValue,**old** value,**new** value

        > 当value被change时，listener的changed方法被调用
  - wraning : can recompute its value lazily or eagerly

    > 可以延迟计算或者立即计算出value

    - lazy： 当value变成invalid，并不知道value是否被change，直到被recomputed

      > 而且值被重新计算只有当需要read时计算，例如调用getValue method

    - eager： 当value变成invalid，虽然对change一无所知，但是还是会立即计算
    - generate invalidation event： 为了生成失效事件，你可以使用lazy or eager
    - generate change event： 为了产生change事件，要求eager

```
public interface ObservaleValue<T> extends Observable
{
    void addListener(ChangeListener<? super T> listener);
    void removeListener(ChangeListener<? super T> listener);
    T   getValue();
}
```

- **ReadOnlyProperty** interface : add two methods

  > inherits from ObservableValue

- **getBean** method : returns reference of the bean that contains the property
- **getName** method : returns the name of  the property

> A read-only proper implements this interface

```
public interface ReadOnlyProperty<T> exrtends ObservableValue<T>
{
    Object getBean();
    String getName();
}
```

- **WritableValue** interface : not herits any interface
    - **getValue** method
    - **setValue** method

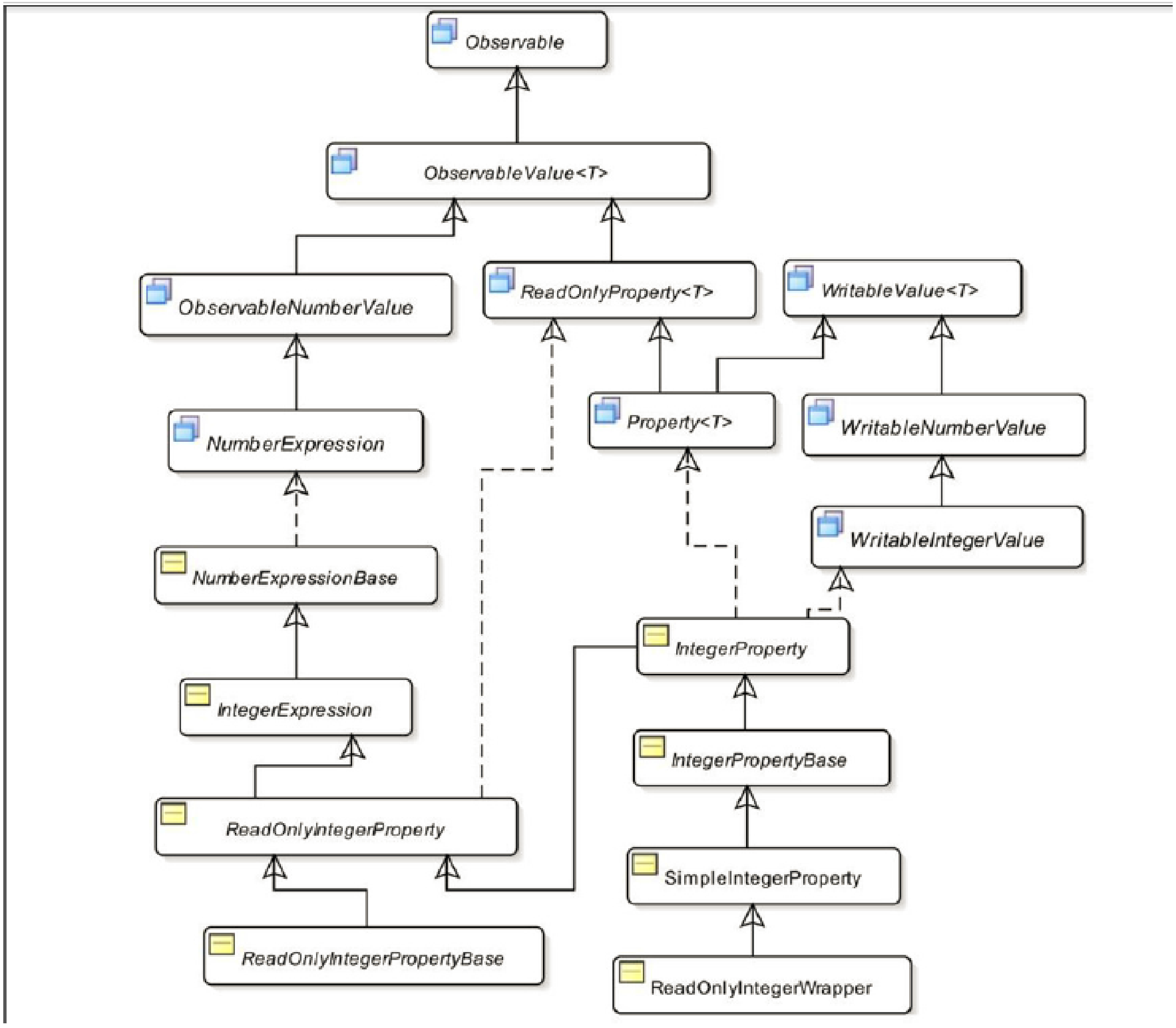
> A read/write property implements this interface

```
public interface WritableValue<T>
{
    T getValue();
    void setValue(T value);
}
```

- **Property** interface : inherit from **ReadOnlyProperty** and **WritableValue** interface

> 继承了只读和读写 interface
>
> add five methods to support binding

- `void bind(Observable<? extends T> observable)`
    - adds a **unidirectional binding** between this `Property` and the specified `ObservableValue` (单向绑定)
- `void unbind()`
- `void bindBidirectional(Property<T> other)`
    - creates a bidirectional binding between this `Property` and the specified `Property`(双向绑定)
- `void unbindBidirectional(Property<T> other)`
- `boolean isBound()`
    - if the `Property` is bound



## Handling Property Invalidation Events

- generate an invalidation event : 从valid变成invalid，invalid再变成invalid就没有
- **InvalidationListener** interface
    - **invalidated** method
        - parameter : **Observable**

> Properties in JavaFX use lazy evaluation
>
> 当重新计算时（调用get or getValue 时），invalid 变成 valid

- example

```
// 创建 IntegerPropety 属性
IntegerProperty counter = new SimpleIntegerProperty(100);
// 添加失效监听器 InvalidationListener
counter.addListener(InvalidationTest::invalidated);

// 类似
counter.addListener(new InvalidationListener()

                   {
```

```
                            @Override
                            public void invalidated(Observable prop)
                            {
                                InvalidationTest.invalidated(prop);
                            }
                        })
```

- 你可以向属性不止添加一个invalidation listener

  > 记得调用removeListener释放，否则内存泄漏

- example

```
import javafx.beans.Observable;
import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;
public class TestProperty {
    public static void main(String[] args)
    {
        IntegerProperty counter = new SimpleIntegerProperty(100);
        // Add an invalidation listener to the counter property
        counter.addListener(TestProperty::invalidated);
        System.out.println("before changing");
        counter.set(101);
        System.out.println("after changing");

        System.out.println("before changing");
        counter.set(102);
        System.out.println("after changing");

        // make the counter property valid by calling its get()
        int value = counter.get();
        System.out.println("Counter value = "+ value);

        System.out.println("before changing");
        counter.set(102);
        System.out.println("after changing");

        System.out.println("before changing");
        counter.set(103);
        System.out.println("after changing");

    }
    public static void invalidated(Observable prop)
    {
        System.out.println(prop.getClass() + ": Counter is invalid");
    }
}
```

> set相同的值不会 fire invalidation event
>
> get使得valid

## Handling Property Change Events

- property change events : every time the value of a property changes
- **ChangeListener** interface
  - **changed** method
    - parameter :
      - **ObservableValue<? super T>** : the reference of the property object
      - T : the old value
      - T : the new value

```java
// method - 1
counter.addListener(new ChangeListener<Number>()
{
    @Override
    public void changed(ObservableValue<? extends Number> prop,
    Number oldValue,
    Number newValue)
    {
        System.out.print("Counter changed: ");
        System.out.println("Old = " + oldValue + ", new = " + newValue);
    }
});

// method - 2
counter.addListener(new ChangeListener<Object>()
{
    @Override
    public void changed(ObservableValue<? extends Object> prop,
    Object oldValue,
    Object newValue)
    {
        System.out.print("Counter changed: ");
        System.out.println("Old = " + oldValue + ", new = " + newValue);
    }
});
```

> - Calling the set() method with the name value does not fire a property change event
> - eager evaluation

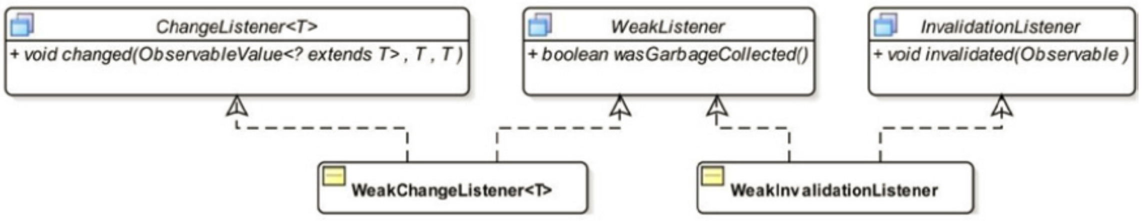
## Avoiding Memory Leaks in Listeners

> 除了RemoveListener还可以使用自动回收

- Use weak listener : grabage collected automatically
- A weak listener is an instance of the **WeakListener** interface

- JavaFX provides two implementtation classes of the **WeakListener** interface

  - `WeakInvalidationlistener` class
  - `WeakChangeListener` class



ChangeListener<T>
+ void changed(ObservableValue<? extends T>, T, T)
WeakListener
+ boolean wasGarbageCollected()
InvalidationListener
+ void invalidated(Observable)
WeakChangeListener<T>
WeakInvalidationListener

*Figure 2-3. A class diagram for WeakChangeListener and WeakInvalidationListener*

**WeakChangeListener**

- wrapper for a ChangeListener

  > 只要一个构造函数，接受一个ChangeListener的实例

```
ChangeListener<Number> cListener = ...
WeakChangeListener<Number> wListener = new WeakChangeListener(cListener);

counter.addLister(wListener);
```

> wListener = null;
>
> System.gc();
>
> 则被自动回收

## Handling Invalidation and Change Events

- invalidation listeners perform better than change listeners
  - compute the value lazily
  - Multiple invalidations in a row fire only one invalidation event

> 按情况使用

## Using Bindings in JavaFX

- 绑定观察其依赖部分的改变，并自动重新计算其值
- JavaFX use lazy evaluation for all bindings
- example

  ```
  IntegerProperty x = new IntegerProperty(100);
  IntegerProperty y = new IntegerProperty(200);

  NumberBinding sum = x.add(y); // sum = x + y  单向绑定
  ```

- **NumberBinding** class
    - isValid method : return true if valid
    - intValue method : get int value and valid
    - longValue method : get long  value and valid
    - floatValue method : get float  value and valid
    - doubleValue method : get double value and valid
    - initValue method : compute its value and mark itself as valid

- *wraning* : A binding adds invalidation listeners to all of its dependencies

  > 任何dependencies变成invalid则绑定自己也变成invalid
  >
  > invalid不意味着值改变了，而是意味着需要重新计算值

- bind a property to a binding
  - example

    ```
    IntegerProperty x = new SimpleIntegerProperty(10);
    IntegerProperty y = new SimpleIntegerProperty(20);
    IntegerProperty z = new SimpleIntegerProperty(60);

    z.bind(x.add(y));
    ```

    > bind the property z to an expression x + y
    >
    > 现在，如果x或者y改变，则z变成invalid
    >
    > 下次获取z，就会自动计算 x.add(y) 获取新值更新z

  - **bind** method
  - **unbind** method
    - such as : z.unbind()

## Unidirectional and Bidirectional Bindings

- unidirectional binding : should bind to a expression
  - *such as* : **bind** method
- bidirectional binding : must bind to a property
  - *such as:* **bindBidirectional** method
    - between a property and another property of the same type

#### Unidirectional binding restriction

- 一旦将property绑定到表达式之后，你无法直接修改property，而是要计算更新值

  > 如果非要这么做，必须unbind再进行直接修改

```java
IntegerProperty x = new SimpleIntegerProperty(10);
IntegerProperty y = new SimpleIntegerProperty(20);
IntegerPtoperty z = new SimpleIntegerProperty(60);

z.bind(x.add(y)); // bind to an expression
z.set(7878); // will throw a RuntimeException

// 正确
z.unbind();
z.set(7878);
```

- 只能绑定到一个
  - such as : 不能 z = x+y 又 z = a+b

    > 将会 unbind the previous binding

```java
z.bind(x.add(y));  // z bind to x.add(y)
z.bind(a.add(b));  // unbind ; and z bind to a.add(b)
```

#### Bidirectional binding

> 去除了单向绑定的缺点

- multiple bidirectional bindings at the same time

```java
x = y
x = z
// 三者同步，最后全为z的值
```

- change independent

## Understanding the Binding API

- Hight-level binding API
  - using the JavaFX class library
- Low-level binding API
  - 从已经存在的绑定类中派生出其他类再编写

### The Hight-level Binding API

- Fluent API
- Bindings class

#### Using the Fluent API

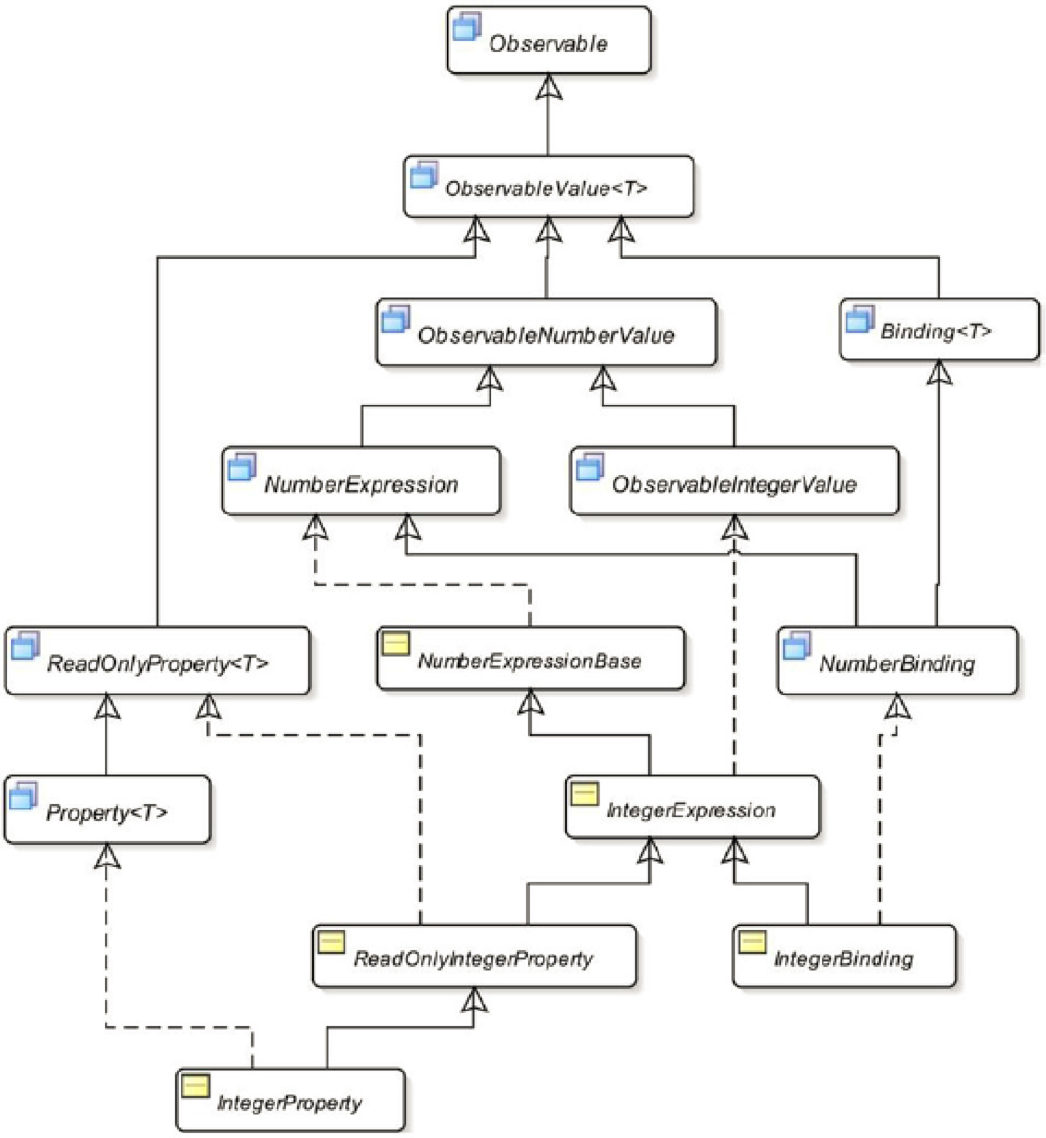
- define : consists of several methods in different interfaces and classes

  > method chaning : 一条语句中进行联合方法调用

- example

```
// 原来
x.add(y);
x.add(z);
// 后来
x.add(y).add(z)
```

---



  - **XXXExpression** classes : have the methods that used to create bingding expressions

**The Binding interface**

- define : An instance of the Binding interface represents a value that is derived from one or more sources konwn as dependencies
  - **four methods**
    - `public void dispose ()`
      - remove references to other objects (删除对其他对象的引用)
    - `public ObservableList<?> getDependencies()`

        - 这个方法用于调试目的，产品代码中不出现
    - `public void invalidate()`
        - 让一个绑定失效（invalidated a Binding）
    - `public boolean isValid()`
        - 判定是否是有效绑定

**The NumberBinding Interface**

- It is implemented by DoubleBinding, FloatBinding, IntegerBinding, and LongBinding classes

**The ObservableNumberValue Interface**

- An instance of the ObservableNumber interface wraps a numeric value of int,long,...
    - double doubleValue() method
    - float floatValue() method
    - int intValue() method
    - long longValue() method

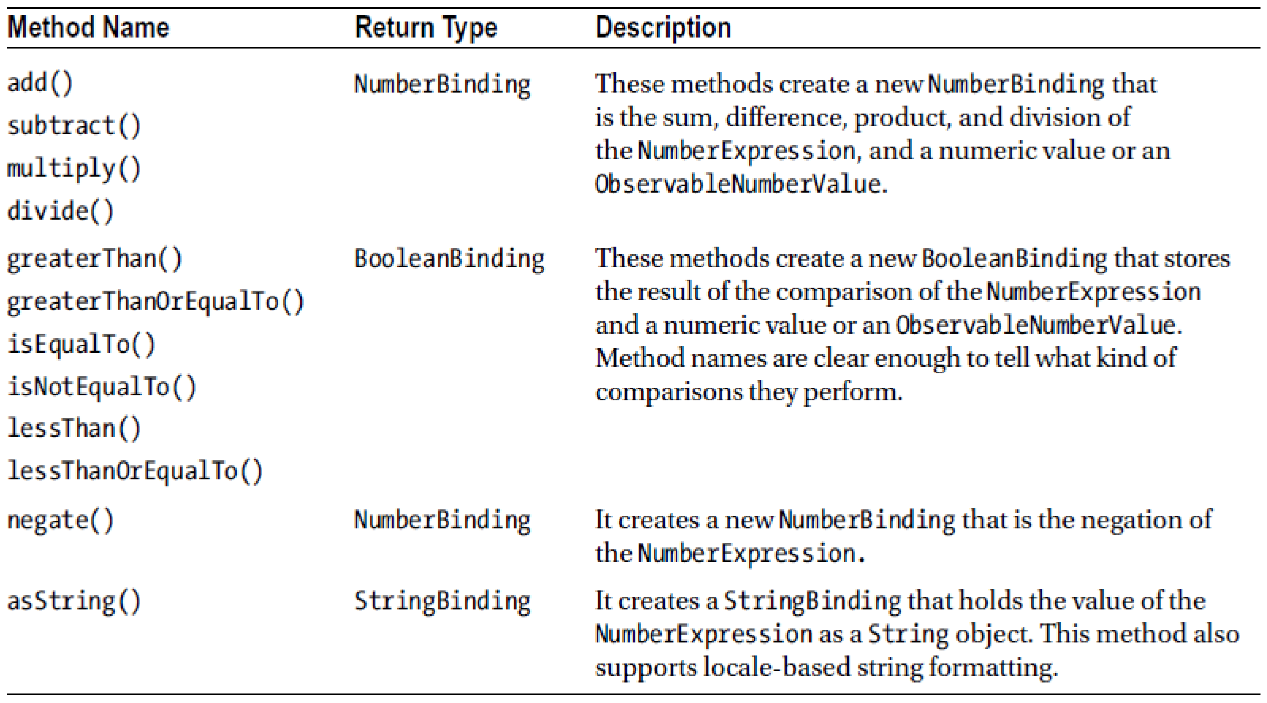
**The ObservableIntegerValue Interface**

- The ObservableIntegerValue interface defines a get() method that returns the type specific int value

**The NumberExpression Interface**

- contatin several convenience methods to create bindings using a fluent style
- return a **Bingding** type
    - such as **NumberBinding**,**BooleanBinding**,...

*Table 2-2. Summary of the Methods in the `NumberExpression` Interface*



| Method Name | Return Type | Description |
|---|---|---|
| `add()`<br>`subtract()`<br>`multiply()`<br>`divide()` | `NumberBinding` | These methods create a new `NumberBinding` that is the sum, difference, product, and division of the `NumberExpression`, and a numeric value or an `ObservableNumberValue`. |
| `greaterThan()`<br>`greaterThanOrEqualTo()`<br>`isEqualTo()`<br>`isNotEqualTo()`<br>`lessThan()`<br>`lessThanOrEqualTo()` | `BooleanBinding` | These methods create a new `BooleanBinding` that stores the result of the comparison of the `NumberExpression` and a numeric value or an `ObservableNumberValue`. Method names are clear enough to tell what kind of comparisons they perform. |
| `negate()` | `NumberBinding` | It creates a new `NumberBinding` that is the negation of the `NumberExpression`. |
| `asString()` | `StringBinding` | It creates a `StringBinding` that holds the value of the `NumberExpression` as a `String` object. This method also supports locale-based string formatting. |

- If one of the operands is a double, the result is a double.

- If none of the operands is a double and one of them is a float, the result is a float.
- If none of the operands is a double or a float and one of them is a long, the result is a long
- Otherwise, the result is an int

```
IntegerProperty x = new SimpleIntegerProperty(1);
IntegerProperty y = new SimpleIntegerProperty(2);
NumberBinding sum = x.add(y);
int value = sum.intValue();


IntegerProperty x = new SimpleIntegerProperty(1);
IntegerProperty y = new SimpleIntegerProperty(2);
// Casting to IntegerBinding is safe
IntegerBinding sum = (IntegerBinding)x.add(y);
int value = sum.get();
```

### NumberExpressionBase class

- implementation of the **NumberExpression**

### IntegerExpression class

- extends the **NumberExpressionBase** class
- overrride methods in its superclass to provide a type-specific return type

```
public class CircleArea
{
    public static void main(String[] args)
    {
        DoubleProperty radius = new SimpleDoubleProperty(7.0);
        // Create a binding for computing arae of the circle
        DoubleBinding area = radius.multiply(radius).multiply(Math.PI);
        System.out.println("Radius = " + radius.get() +
        ", Area = " + area.get());
        // Change the radius
        radius.set(14.0);
        System.out.println( "Radius = " + radius.get() +
        ", Area = " + area.get());
        // Create a DoubleProperty and bind it to an expression
        // that computes the area of the circle
        DoubleProperty area2 = new SimpleDoubleProperty();
        area2.bind(radius.multiply(radius).multiply(Math.PI));
        System.out.println("Radius = " + radius.get() +
        ", Area2 = " + area2.get());
    }
}
```

### The StringBinding Class

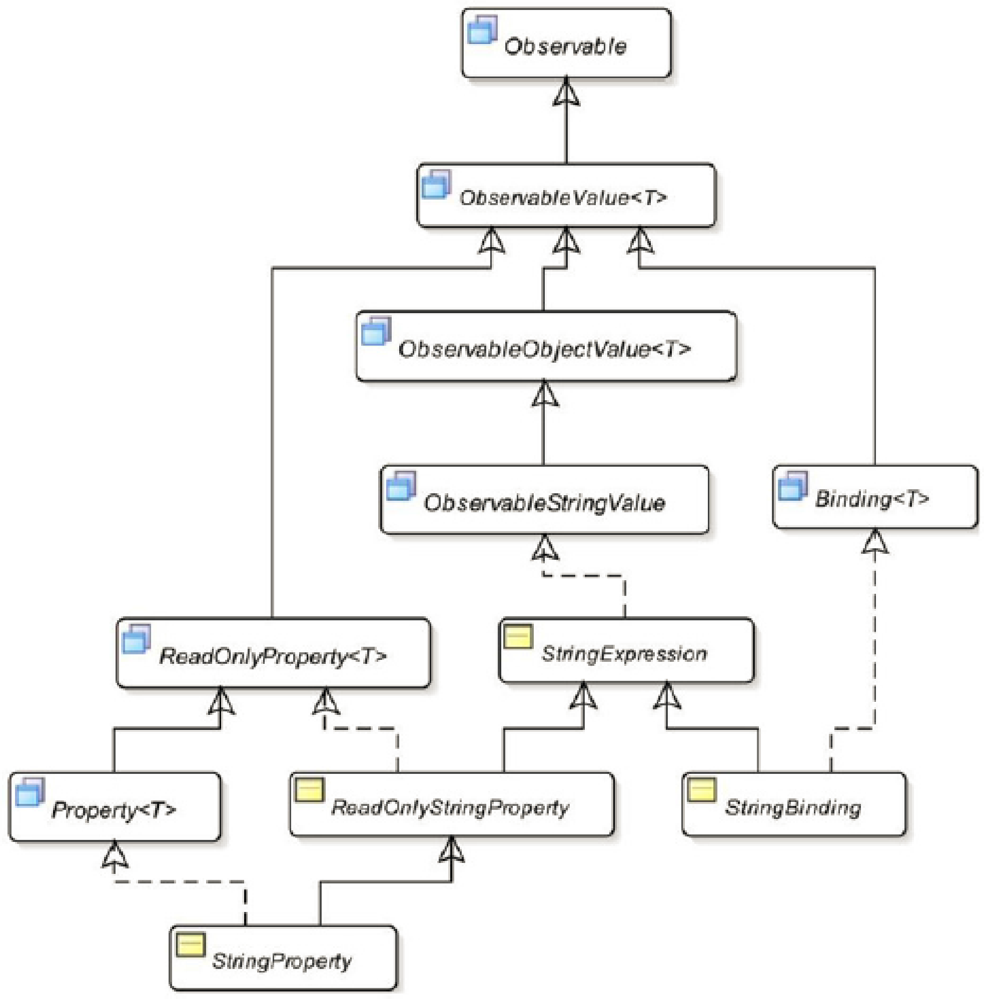


*Figure 2-5. A partial class diagram for `StringBinding`*

### ObservableStringValue interface

- get () method whose return type is String

### StringExpression class

- let you create binding using a fluent style
- methods 连接对象到StringExpression
  - compare two strings
  - check for null
  - others
  - getValue method
  - getValueSafe method : return empty String if current value is null

```
public class StringExpressionTest
{
    public static void main(String[] args)
    {
        DoubleProperty radius = new SimpleDoubleProperty(7.0);
        DoubleProperty area = new SimpleDoubleProperty(0);
        StringProperty initStr = new SimpleStringProperty("Radius = ");
        // Bind area to an expression that computes the area of the circle
        area.bind(radius.multiply(radius).multiply(Math.PI));
        // Create a string expression to describe the circle
        StringExpression desc = initStr.concat(radius.asString())
                                       .concat(", Area = ")
                                       .concat(area.asString(Locale.US, "%.2f"));
        System.out.println(desc.getValue());
        // Change the radius
        radius.set(14.0);
        System.out.println(desc.getValue());
    }
}
```

### The ObjectExpressionn and ObjectBinding Classes

- create bingdings of any type of objects
- The ObjectExpression class has methods to compare objects for equality and to check for null values

```
public class ObjectBindingTest
{
    public static void main(String[] args)
    {
        Book b1 = new Book("J1", 90, "1234567890");
        Book b2 = new Book("J2", 80, "0123456789");
        ObjectProperty<Book> book1 = new SimpleObjectProperty<>(b1);
        ObjectProperty<Book> book2 = new SimpleObjectProperty<>(b2);
        // Create a binding that computes if book1 and book2 are equal
        BooleanBinding isEqual = book1.isEqualTo(book2);
        System.out.println(isEqual.get());
        book2.set(b1);
        System.out.println(isEqual.get());
    }
}
```

### The BooleanExpression and BooleanBinding Classes

- The BooleanExpression class contains methods
    - and
    - or
    - not
    - isEqualTo
    - isNotEqualTo

```
public class BooelanExpressionTest
{
    public static void main(String[] args)
    {
        IntegerProperty x = new SimpleIntegerProperty(1);
        IntegerProperty y = new SimpleIntegerProperty(2);
        IntegerProperty z = new SimpleIntegerProperty(3);
        // Create a boolean expression for x > y && y <> z
        BooleanExpression condition = x.greaterThan(y).and(y.isNotEqualTo(z));
        System.out.println(condition.get());
        // Make the condition true by setting x to 3
        x.set(3);
        System.out.println(condition.get());
    }
}
```

### Using Ternary Operation in Expressions

- when-then-otherwise (三元运算符)

```
new When(condition).then(value1).otherwise(value2);
// condition must be an ObservableBooleanValue
// the type of value1 and value2 must be the same
// Values may be constants or instances of ObservableValue
```

### Using the Bindings Utility Class

- The **Bindings** class is a helper class to create simple bindings
- more 150 static methods

```
StringExpression desc = Bindings.concat(
    "Radius = ",radius.asString(Locale.US,"%.2f"), // java.util.Locale
    ",Area = ",area.asString(Locale.US,"%.2f")
);
```

### Using the selectXXX() method

```
public class Address
{
    private StringProperty zip = new SimpleStringProperty("36106");
    public StringProperty zipProperty()
    {
        return zip;
    }
}

public class Person
{
    private ObjectProperty<Address> addr = new SimpleObjectProperty(new Address());
    public ObjectProperty<Address> addrProperty()
    {
        return addr;
    }

}
```

```
ObjectProperty<Person> p = new SimpleObjectProperty(new Person());
StringBinding zipBinding = Bindings.selectString(p,"addr","zip");
```

> create a binding for a nested property

#### Combining the Fluent API and the Bindings Class

```
DoubleProperty radius = new SimpleDoubleProperty(7.0);
DoubleProperty area = new SimpleDoubleProperty(0);

// Combine the Fluent API and Bindings class API
area.bind(Bindings.multiply(Math.PI, radius.multiply(radius)));
```

#### Using the Low-level Binding API

- The high-level binding API is not sufficient in all cases
- step
  - Create a class the extends one of the binding classes
    - if you want to create a **DoubleBinding**, you need to extend the **DoubleBingding** class
    - Call the **bind()** method of the superclass to bind all dependencies
    - Override the **computeValue()** method of the superclass to write the logic for your binding
- example

  ```
  final DoubleProperty radius = new SimpleDoubleProperty(7.0);
  DoubleProperty area = new SimpleDoubleProperty(0);
  DoubleBinding areaBinding = new DoubleBinding()
  {
      {
          this.bind(radius);
      }
      @Override
      protected double computeValue()
      {
          double r = radius.get();
          double area = Math.PI * r * r;
          return area;
      }
  };

  area.bind(areaBinding);
  ```

  > - calls the bind() method passing the reference of the radius property
  > - the computeValue() method computes and returns the area of the circle
  > - final radius property, because it used inside the anonymous class

- example

```java
import java.util.Formatter;
import java.util.Locale;
import javafx.beans.binding.DoubleBinding;
import javafx.beans.binding.StringBinding;
import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;

public class LowLevelBinding
{
    public static void main(String[] args)
    {
        final DoubleProperty radius = new SimpleDoubleProperty(7.0);
        final DoubleProperty area = new SimpleDoubleProperty(0);
        DoubleBinding areaBinding = new DoubleBinding()
        {
            {
                this.bind(radius);
            }
        @Override
        protected double computeValue()
        {
            double r = radius.get();
            double area = Math.PI * r *r;
            return area;
        }
        };
        //Bind area to areaBinding
        area.bind(areaBinding);
        // Create a StringBinding
        StringBinding desc = new StringBinding()
        {
            {
                this.bind(radius, area);
            }
            @Override
            protected String computeValue()
            {
                Formatter f = new Formatter();
                f.format(Locale.US, "Radius = %.2f, Area = %.2f",
                radius.get(), area.get());
                String desc = f.toString();
                return desc;
            }
            @Override
            public ObservableList<?> getDependencies()
            {
                return FXCollections.unmodifiableObservableList(
                FXCollections.observableArrayList(radius, area));

            }
```

```
                @Override
                public void dispose()
                {
                    System.out.println("Description binding is disposed.");
                }
                @Override
                protected void onInvalidating()
                {
                    System.out.println("Description is invalid.");
                }
            };
            System.out.println(desc.getValue());
            //Change the radius
            radius.set(14.0);
            System.out.println(desc.getValue());
        }
    }
```

## Using Bindings to Center a Circle

```
package com.javafx;
import javafx.application.Application;
import javafx.beans.binding.Bindings;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.Group;
import javafx.scene.shape.Circle;

public class CenteredCircle extends Application
{
    public static void main(String... args)
    {
        Application.launch(args);
    }
    @Override
    public void start(Stage stage)
    {
        Circle c = new Circle();
        Group root = new Group(c);
        Scene scene = new Scene(root,100,100);

        c.centerXProperty().bind(scene.widthProperty().divide(2));
        c.centerYProperty().bind(scene.heightProperty().divide(2));
        c.radiusProperty().bind(Bindings.min(scene.widthProperty(),
                scene.heightProperty()).divide(2));
        stage.setTitle("Binding in JavaFX");
        stage.setScene(scene);
        stage.sizeToScene();
        stage.show();

    }
```

```
  }
```