JAVAFX-5

##fx 属性与布局

属性与布局是一个具备gui开发能力的开发者,快速进入开发必备的知识储备,下面简单说一说常用的属性,与布局

###颜色

- 颜色

在 javafx.scene.paint.Color 类中提供了 RGB HSB WEB 等在不同应用场景下的方法,下面基本上你看看代码就可以快速应用到你的项目里面去.

```
Color c = Color.BLUE; //use the blue constant
Color c = new Color(0,0,1,1.0); //standard constructor, use 0->1.0 values, explicit alpha of 1.0

Color c = Color.color(0,0,1.0); //use 0->1.0 values. implicit alpha of 1.0
Color c = Color.color(0,0,1.0,1.0); //use 0->1.0 values, explicit alpha of 1.0

Color c = Color.rgb(0,0,255); //use 0->255 integers, implicit alpha of 1.0
Color c = Color.rgb(0,0,255,1.0); //use 0->255 integers, explicit alpha of 1.0

Color c = Color.hsb(270,1.0,1.0); //hue = 270, saturation & value = 1.0. inplicit alpha of 1.0
Color c = Color.hsb(270,1.0,1.0,1.0); //hue = 270, saturation & value = 1.0, explicit alpha of 1.0

Color c = Color.web("0x0000FF",1.0);// blue as a hex web value, explicit alpha
Color c = Color.web("0x0000FF");// blue as a hex web value, implicit alpha
Color c = Color.web("0x00F");// blue as a short hex web value, implicit alpha
Color c = Color.web("#0000FF",1.0);// blue as a hex web value, explicit alpha
Color c = Color.web("#0000FF");// blue as a hex web value, implicit alpha
Color c = Color.web("#00F");// blue as a short hex web value, implicit alpha
Color c = Color.web("0000FF",1.0);// blue as a hex web value, explicit alpha
Color c = Color.web("0000FF");// blue as a hex web value, implicit alpha
```

- FX渐变颜色

梯度绘制可以在两种或更多种颜色之间内插，这给出形状的深度。**JavaFX**提供两种类型的渐变：径向渐变(`RadialGradient`)和线性渐变(`LinearGradient`)。

要在JavaFX中创建渐变颜色，需要设置五个属性值。如下 -

- 设置开始起点的第一个停止颜色。
- 将终点设置为终止停止颜色。
- 设置`proportional`属性以指定是使用标准屏幕坐标还是单位平方坐标。
- 将循环方法设置为使用三个枚举：`NO_CYCLE`，`REFLECT`或`REPEAT`。
- 设置停止颜色数组。

- 线性梯度(LinearGradient)

属性	数据类型及描述
startX	`Double` - 设置梯度轴起点的`X`坐标。
startY	`Double` - 设置梯度轴起点的`Y`坐标。
endX	`Double` - 设置梯度轴终点的`X`坐标。
endY	`Double` - 设置梯度轴终点的`Y`坐标
proportional	`Boolean` - 设置坐标是否与形状成比例。设置为`true`时则使用单位正方形坐标，否则使用屏幕坐标系。
cycleMethod	`CycleMethod` - 设置应用于渐变的循环方法。
stops	`List<Stop>` - 设置渐变颜色指定的停止列表。

- 径向渐变 RadialGradient

ocusAngle	`Double` - 设置从渐变中心到映射第一种颜色的焦点的角度(以度为单位)。
focusDistance	`Double` - 设置从渐变中心到映射第一种颜色的焦点的距离。
centerX	`Double` - 设置渐变圆的中心点的`X`坐标。
centerY	`Double` - 设置渐变圆的中心点的`Y`坐标。
radius	`Double` - 设置颜色渐变的圆的半径。
proportional	`boolean` - 设置坐标和大小与形状成比例。
cycleMethod	`CycleMethod` - 设置应用于渐变的Cycle方法。
Stops	`List<Stop>` - 设置渐变颜色的停止列表

###域模型与属性绑定

- 我们实现登录操作的时候,绑定实体类的值,用户名,密码等 封装 get set 的时候可以使用这个字段
- JavaFX的属性包含实际值，并提供更改支持，无效支持和绑定功能。所有JavaFX属性类都位于`javafx.beans.property.*`包命名空间中。

下面的列表是常用的属性类。

- javafx.beans.property.SimpleBooleanProperty
- javafx.beans.property.ReadOnlyBooleanWrapper
- javafx.beans.property.SimpleintegerProperty
- javafx.beans.property.ReadOnlyintegerWrapper
- javafx.beans.property.SimpleDoubleProperty
- javafx.beans.property.ReadOnlyDoubleWrapper
- javafx.beans.property.SimpleStringProperty
- javafx.beans.property.ReadOnlyStringWrapper

```
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;

public class Main{
  public static void main(String[] args) {
    StringProperty password  = new SimpleStringProperty("yiibai.com");
    password.set("example.com");
    System.out.println("Modified StringProperty "  + password.get() );
  }
}
```

- javabean具体代码(忽略get set)

```
class User {
  private final static String USERNAME_PROP_NAME = "userName";
  private final ReadOnlyStringWrapper userName;
  private final static String PASSWORD_PROP_NAME = "password";
  private StringProperty password;

  public User() {
    userName = new ReadOnlyStringWrapper(this, USERNAME_PROP_NAME,"fake user");
    password = new SimpleStringProperty(this, PASSWORD_PROP_NAME, "");
  }
  public final String getUserName() {
```

- 属性更改事件

属性可以通知值更改的事件处理程序，以便在属性更改时进行响应处理相关操作。JavaFX属性对象包含一个`addListener()`方法，它接受两种类型的功能接口：`ChangeListener(改变值)`和`invalidationListener(初始化值)`

- 代码:

```
SimpleIntegerProperty xProperty = new SimpleIntegerProperty(0);

// Adding a change listener with anonymous inner class
xProperty.addListener(new ChangeListener<Number>() {
  @Override
  public void changed(ObservableValue<? extends Number> ov, Number oldVal,
      Number newVal) {
    System.out.println("old value:"+oldVal);
    System.out.println("new value:"+newVal);
  }
});
```

```
SimpleIntegerProperty xProperty = new SimpleIntegerProperty(0);

// Adding a invalidation listener (anonymous inner class)
xProperty.addListener(new InvalidationListener() {
  @Override
  public void invalidated(Observable o) {
    System.out.println(o.toString());
  }
});
```

- 他们都实现了`ObservableValue`和`Observable`接口

实际上也可以这样写:

```
// Adding a change listener with lambda expression
xProperty.addListener((ObservableValue<? extends Number> ov, Number oldVal,
    Number newVal) -> {
      System.out.println("old value:"+oldVal);
      System.out.println("new value:"+newVal);
});
```

####绑定

- 双向绑定

双向绑定绑定相同类型的属性，并同步两侧的值。当使用`bindBidirectional()`方法双向绑定时，需要两个属性都必须是可读/写的。

```
public static void main(String[] args) {
    User contact = new User("Jame", "Bind");
    StringProperty fname = new SimpleStringProperty();
    fname.bindBidirectional(contact.firstNameProperty());

    contact.firstNameProperty().set("new value");
    fname.set("新绑定名称值");

    System.out.println("firstNameProperty = " + contact.firstNameProperty().get());
    System.out.println("fname = " + fname.get());
```

- 高级别绑定

`multiply()`，`divide()`，`subtract()`，`isEqualTo()`，`isNotEqualTo()`，`concat()`。 请注意，方法名称中没有`get`或`set`。当链接**API**在一起时可以写代码，就像类似于写英文句子，例如，`width().multiply(height()).divide(2)`。

```
IntegerProperty width = new SimpleIntegerProperty(10);
IntegerProperty height = new SimpleIntegerProperty(10);
NumberBinding area = width.multiply(height);
System.out.println(area.getValue());
```

- 低级别绑定

当对`NumberBinding`类进行子类化时，使用低级别绑定，例如`Double`类型的`DoubleBinding`类。

在`DoubleBinding`类的子类中，我们覆盖它的`computeValue()`方法，以便可以使用运算符(例如`*`和`-`)来制定复杂的数学方

程计算

```
public static void main(String[] args) {
  DoubleProperty width = new SimpleDoubleProperty(2);
  DoubleProperty height = new SimpleDoubleProperty(2);
  DoubleBinding area = new DoubleBinding() {
    {
      super.bind(width, height); // initial bind
    }

    @Override
    protected double computeValue() {
      return width.get() * height.get();
    }
  };
  System.out.println(area.get());
}
```

####值的跟踪传递

- `javafx.collections`包

接口

接口	描述
ObservableList	允许跟踪更改的列表
ListChangeListener	接收更改通知的接口
ObservableMap	允许跟踪更改的映射
MapChangeListener	从`ObservableMap`接收更改通知的接口

类

类	描述
FXCollections	实用程序类映射到`java.util.Collections`
ListChangeListener.Change	表示对`ObservableList`所做的更改
MapChangeListener.Change	表示对`ObservableMap`所做的更改

- list map 操作方法是一样的

```
List<String> list = new ArrayList<String>();

ObservableList<String> observableList = FXCollections.observableList(list);
observableList.addListener(new ListChangeListener() {
    @Override
    public void onChanged(ListChangeListener.Change change) {
        System.out.println("有修改操作!");
    }
});
observableList.add("item one");
list.add("item two");
System.out.println("Size: " + observableList.size());
```