How to Add a Property to A Component

How to Add a Property to a Component

Originally created by Kate Feeney

This document describes how to add a new property to an existing App Inventor component.

- 1. Adding a Property to the Properties Panel
  - 1.1 PropertyEditors
  - 1.2 Creating a New PropertyEditor
    - 1.2.1 Creating the new PropertyEditor class
    - 1.2.2 Adding the New PropertyEditor to the Properties Panel
  - 1.3. Associate the Property with the Component
- 2. Changing the Designer View When a Property Changes
  - 2.1 Change Component's Attributes
  - 2.2 Update the onPropertyChange Method
  - 2.3 Update Any Other Necessary Methods
- 3. Changing the Android Representation of the Component
  - 3.1. Button Shape Example
- 4. Update Version Numbers
  - 4.1 YaVersion
  - 4.2 YoungAndroidFormUpgrader
  - 4.3 BlockSaveFile
- 5. Deprecating methods and events and properties
- 6. Internationalization

As an example, we will show how the Shape property was added to the ButtonBase component. ButtonBase is an abstract superclass of the Button and the Picker components (ContactPicker, ImagePicker and ListPicker); therefore, all properties defined for ButtonBase are also defined for Button and Picker. Originally the ButtonBase component had no Shape property; it simply used the default shape, which uses the system's defaults and varies from device to device. When the Shape property was added, four choices (default, rounded, rectangular and oval) were included. All examples in this document show the Button component but would be the same for any of the Picker components.

The user will first see the new property in the Button component's Properties panel in the Designer. Because the choices should be restricted to the four legal values, we will create a PropertyEditor that limits the choices to these values and maps them to integers for the internal representation of the property. When the user changes the value of the Shape property, the visual representation of the Button component in the Designer must change so that the user can preview the interface. To do this we will create a method that will change the attributes of the GWT button widget that represents the Button component in the Designer. Finally, since the user ultimately wants the property changed on their Android device we will add code that changes the Button view's BackgroundDrawable depending on which Shape value is selected.

1. Adding a Property to the Properties Panel

The first step is to add the property so that it appears in the Properties panel when a Button is selected. This change is shown in the image below. Every property is associated with a PropertyEditor to allow the user to choose among legal values. Often, an existing PropertyEditor can be used, but in some cases, such as the Shape property, it will be necessary to create a new one. Finally, we will associate the property with the component so that the property will appear in the Properties panel when the component is selected.

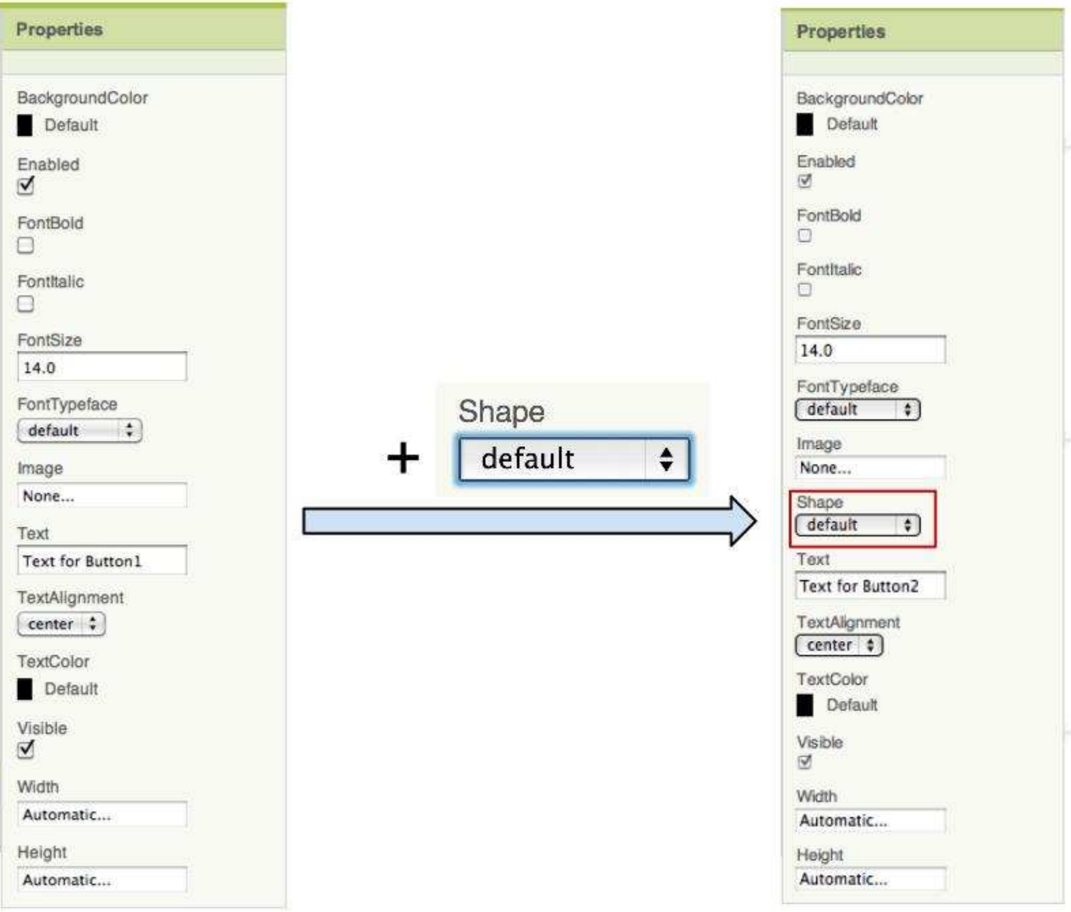

Figure 1: Button Property Panel

1.1 PropertyEditors

Each property has a PropertyEditor that controls what values can be specified in the Designer. Some existing PropertyEditors include the BooleanPropertyEditor (used by ButtonBase.Enabled), NonNegativeFloatPropertyEditor (used by ButtonBase.FontSize) and TextPropertyEditor (used by ButtonBase.Text). If a suitable PropertyEditor already exists (Check if an existing property will meet your needs.), then simply note its name and skip to Section 1.3. Otherwise a new PropertyEditor needs to be created as described in Section 1.2.

The PropertyEditor associated with the Shape property must offer a drop-down menu with the four legal shape values: default, rounded, rectangular and oval (as shown below). Since this does not already exist, a new PropertyEditor must be created.

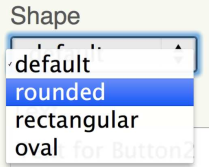

Figure 2: Shape PropertyEditor in the Designer

1.2 Creating a New PropertyEditor

1.2.1 Creating the new PropertyEditor class

The new PropertyEditor class must extend the PropertyEditor class and restrict the user inputs to only legal values. This class will also define how the PropertyEditor is displayed to the user (drop-down menu, text box, etc.).

For our example the new class will be called YoungAndroidButtonShapeChoicePropertyEditor and it will extend the ChoicePropertyEditor class, which itself extends PropertyEditor. This new class must define an array of Choice objects and pass the array to the ChoicePropertyEditor constructor, which will create the drop-down choice widget. A Choice is a static class defined in the ChoicePropertyEditor class and its constructor takes in two strings: `caption` and `value`. The `caption` string is text to be shown in the drop-down choice widget. The `value` string is the value assigned to the property if the choice is selected. The new class is defined in Figure 4.

The first step is to define the four descriptive strings (`caption` strings), which are displayed to the user and will be placed in the array passed to the ChoicePropertyEditor. This is done by adding the following code to the OdeMessages interface. The reason the strings are defined in a separate file rather than hard-coded is to support abstraction and internationalization.

```
//Used in editor/youngandroid/properties/YoungAndroidButtonShapeChoicePropertyEditor.java

 @DefaultMessage("default")       ← this is what will be displayed to the user
                                                   this string is information
 @Description("Text for button shape choice 'default'")   for a translator
                                    the name of the descriptive
 String defaultButtonShape();       string is made up at this point

 @DefaultMessage("rounded")
 @Description("Text for button shape choice 'rounded'")
 String roundedButtonShape();

 @DefaultMessage("rectangular")
 @Description("Text for button shape choice 'rectangular'")
 String rectButtonShape();

 @DefaultMessage("oval")
 @Description("Text for button shape choice 'oval'")
 String ovalButtonShape();
```

Figure 3: Define Strings in OdeMessages class

Now that the strings are defined, the YoungAndroidButtonShapeEditor class can be created as shown in Figure 4.

```
package com.google.appinventor.client.editor.youngandroid.properties;

import static com.google.appinventor.client.Ode.MESSAGES;
import com.google.appinventor.client.widgets.properties.ChoicePropertyEditor;

/**
 * Property editor for button style.
 *
 * @author feeney.kate@gmail.com (Kate Feeney)
 */
public class YoungAndroidButtonShapeChoicePropertyEditor extends ChoicePropertyEditor {

  // Button shape choices      extend ChoicePropertyEditor
  private static final Choice[] shapes = new Choice[] {
```

```
    new Choice(MESSAGES.defaultButtonShape(), "0"),
    new Choice(MESSAGES.roundedButtonShape(), "1")
    new Choice(MESSAGES.rectButtonShape(), "2"),      <- define an array of the choices
    new Choice(MESSAGES.ovalButtonShape(), "3")
  };

  public YoungAndroidButtonShapeChoicePropertyEditor() {
   super(shapes);    <- pass array to the ChoicePropertyEditor constructor
  }
}
```

Figure 4: YoungAndroidButtonShapeEditor class

The `value` strings (e.g., “0”, “1”, “2”, “3” in Figure 4) will be assigned to the property and can be accessed via the component’s getters and setters. When the string is retrieved by the component’s getter it will be retrieved as an int for which constants should be defined in the Component class. We will define the constants in the Component class since all components inherit from it. Therefore the following code (Figure 5) is to be added.

```
  /*
   * Button Styles.
   */
  static final int BUTTON_SHAPE_DEFAULT = 0;
  static final int BUTTON_SHAPE_ROUNDED = 1;
  static final int BUTTON_SHAPE_RECT = 2;
  static final int BUTTON_SHAPE_OVAL = 3;
```

Figure 5: Define `value` Strings in the Component class

1.2.2 Adding the New PropertyEditor to the Properties Panel

The YoungAndroidPalettePanel class creates the component palette on the left side of the Designer and instantiates the components’ PropertyEditors. We need to add the logic to do this in the createPropertyEditor() method. First, we need to define a constant for the property in the propertyTypeContants class. Add the following code.

```
  /**
  * Button shapes.   * @see com.google.appinventor.client.editor.youngandroid.properties.

  * YoungAndroidButtonShapeChoicePropertyEditor
  */
  public static final String PROPERTY_TYPE_BUTTON_SHAPE = "button_shape";
```

Then add the following case to the createPropertyEditor() method inside the same class.

```
    } else if (editorType.equals(PropertyTypeConstants.PROPERTY_TYPE_BUTTON_SHAPE)) {
      return new YoungAndroidButtonShapeChoicePropertyEditor();
```

Figure 6: Addition to YoungAndroidPalettePanel.createPropertyEditor()

1.3. Associate the Property with the Component

Create a setter and a getter method for the new property in the component’s class. Both the setter and the getter methods must be marked with the SimpleProperty annotation. The SimpleProperty annotation consists

of a description, the property's category and whether or not the property is visible (visible in the BlocksEditor). The setter method must also be marked with the DesignerProperty annotation. This annotation consists of the property's editor type, which is defined in the DesignerProperty annotation, and the default value of the property.

For this example the following code needs to be added to the ButtonBase class.

This is the getter:

```
/**
 * Returns the style of the button.
 *
 * @return  one of {@link Component#BUTTON_SHAPE_DEFAULT},
 *          {@link Component#BUTTON_SHAPE_ROUNDED},
 *          {@link Component#BUTTON_SHAPE_RECT} or
 *          {@link Component#BUTTON_SHAPE_OVAL}
 */
@SimpleProperty(
    category = PropertyCategory.APPEARANCE,
    userVisible = false)
public int Shape() {
  return shape;
}
```


Figure 7: Shape Getter

This is the setter:

```
/**
 * Specifies the style the button. This does not check that the argument is a legal
 * value.
 *
 * @param shape one of {@link Component#BUTTON_SHAPE_DEFAULT},
 *          {@link Component#BUTTON_SHAPE_ROUNDED},
 *          {@link Component#BUTTON_SHAPE_RECT} or
 *          {@link Component#BUTTON_SHAPE_OVAL}
 *
 * @throws IllegalArgumentException if shape is not a legal value.
 */
@DesignerProperty(editorType = PropertyTypeConstants.PROPERTY_TYPE_BUTTON_SHAPE,
    defaultValue = Component.BUTTON_SHAPE_DEFAULT + "")
@SimpleProperty(description = "Specifies the button's shape (default, rounded," +
    " rectangular, oval). The shape will not be visible if an Image is being +
    displayed.", userVisible = false)
public void Shape(int shape) {
  this.shape = shape;
  updateAppearance();
}
```


Figure 8: Shape Setter

Then define the variable shape inside the same file with the following code.

```
// Backing for button shape
private int shape;
```

And add the following line to the ButtonBase constructor.

```
Shape(Component.BUTTON_SHAPE_DEFAULT);
```

After implementing the code in this section, the Shape property will appear in the Properties panel when a Button is selected in the designer view; however, changing the property does not yet affect the appearance of the Button.

2. Changing the Designer View When a Property Changes

The app's user interface can be viewed in two locations. The first location is in the Designer and the second is on the Android device (i.e. phone, tablet or emulator). The Designer is the view shown in the browser window and is what will be changed in this section.

Every Component has a corresponding mock class in the appengine project. The mock class is the visual representation of the component in the Designer, and the mock classes generally follow the same hierarchy as the Component classes. If your property changes visual aspects of the component (such as color), then the mock class of that component must adjust the Designer to show these visual changes.

2.1 Change Component's Attributes

There are already a number of methods written to change a component's attributes in MockComponent, MockComponentsUtil and MockVisibleComponent; most component mock classes inherit from at least one of these classes.

To change the appearance in the Designer, find the mock class that corresponds to the component to which the property was added. Determine if the class inherits a method that changes the appropriate attribute. If such a method exists, then simply write a method that calls that method with the appropriate arguments, as done in the setEnabledProperty() method in the MockButtonBase class. If a method doesn't exist then follow this Shape property example.

For this example the mock class associated with the ButtonBase component is MockButtonBase. Figure 9 shows the appearances for each value of the Shape property.

Shape	Image
default	default
rounded	rounded
rectangular	rectangular

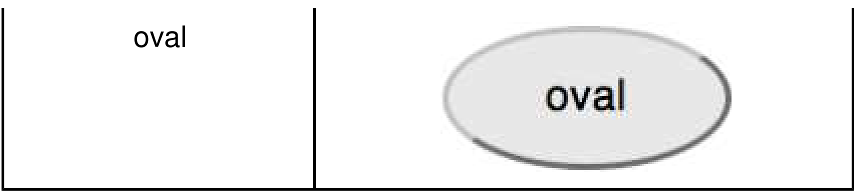

Figure 9: Mock Buttons

Mock components are built on top of GWT widgets. The MockButtonBase creates a button widget which uses the system's defaults and in order to make the above shapes the button's corner radii needs to be changed. There does not already exist a method in MockButtonBase or any of its superclasses that could change a button's corner radii, so the following method (Figure 10) needs to be added to MockButtonBase.

```
// Legal values for shape are defined in
// com.google.appinventor.components.runtime.Component.java.
private int shape;

/*
 * Sets the button's Shape property to a new value.
 */
private void setShapeProperty(String text) {
  shape = Integer.parseInt(text);
  // Android Buttons with images take the shape of the image and do not
  // use one of the defined Shapes.
  if (hasImage) {
    return;
  }
  switch(shape) {
    case 0:
      // Default Button
      DOM.setStyleAttribute(buttonWidget.getElement(), "border-radius", "0px");
      break;
    case 1:
      // Rounded Button.
      // The corners of the Button are rounded by 10 px.
      // The value 10 px was chosen strictly for style.
      // 10 px is the same as ROUNDED_CORNERS_RADIUS defined in
      // com.google.appinventor.components.runtime.ButtonBase.java.
      DOM.setStyleAttribute(buttonWidget.getElement(), "border-radius", "10px");
      break;
    case 2:
      // Rectangular Button
      DOM.setStyleAttribute(buttonWidget.getElement(), "border-radius", "0px");
      break;
    case 3:
      // Oval Button
      String height = DOM.getStyleAttribute(buttonWidget.getElement(), "height");
      DOM.setStyleAttribute(buttonWidget.getElement(), "border-radius", height);
      break;
    default:
      // This should never happen
      throw new IllegalArgumentException("shape:" + shape);
  }
}
```

Figure 10: Addition to MockButtonBase Class

2.2 Update the onPropertyChange Method

Mock component classes contain an onPropertyChange() method which is called by GWT when any of the component's properties are changed and it determines how to change the view in the Designer. The onPropertyChange() method has two string parameters, the `propertyName` and the `newValue`. The `propertyName` is a string representation of the name of the method defined in the property's getter (also the string displayed in the Properties Panel). The `newValue` is the value of the property and is passed to the method that was just created. For this example the `propertyName` would be "Shape" and the `newValue` could be "0", "1", "2" or "3".

There is a list of all the possible `propertyName` values in the MockVisibleComponent class. The new property needs to be added to the list. For this example add the following line.

```
protected static final String PROPERTY_NAME_BUTTONSHAPE= "Shape";
```

Now back in the mock component's class add logic so that if the new property is changed it will call the method that was just created.

For this example add the following statement to the onPropertyChange() method of the MockButtonBase class

```
} else if (propertyName.equals(PROPERTY_NAME_BUTTONSHAPE)){     ← this is the string just
                                                                  defined in this section
  setShapeProperty(newValue);     ← this is the method
                                    defined in Section 2.1
```

Figure 11: Additon to onPropertyChange()

2.3 Update Any Other Necessary Methods

If changing the new property doesn't affect any other properties then this section is complete. It might be a good idea to review the methods in the mock component class to confirm this. If the new property does affect other properties then update the methods called on those property changes as needed.

For this example, since the button shape only changes if there is no background image, the Shape property and the Image property affect each other. Therefore, since the setShapeProperty() method was created (or updated), the setImageProperty() method needs to be updated.

The following two gray lines need to be added to the setImageProperty() method.

```
/*
 * Sets the button's Image property to a new value.
 */
private void setImageProperty(String text) {
  imagePropValue = text;
  String url = convertImagePropertyValueToUrl(text);
  if (url == null) {
    hasImage = false;
    url = "";
    setBackgroundColorProperty(backgroundColor);
    setShapeProperty(Integer.toString(shape));
  } else {
    hasImage = true;
```

```
      // Android Buttons do not show a background color if they have an image.
      // The container's background color shows through any transparent
      // portions of the Image, an effect we can get in the browser by
      // setting the widget's background color to COLOR_NONE.
      MockComponentsUtil.setWidgetBackgroundColor(buttonWidget,
          "&H" + COLOR_NONE);
      DOM.setStyleAttribute(buttonWidget.getElement(), "border-radius", "0px");
    }
    MockComponentsUtil.setWidgetBackgroundImage(buttonWidget, url);
    image.setUrl(url);
  }
```

Figure 12: Addition to setImageProperty()

Now when the Shape property is changed the button shown in the designer view will also change to reflect the user's preference. There will still be no change to the button on the Android device.

3. Changing the Android Representation of the Component

Next decide how you would like to change the visual representation of the component on the Android device. Then implement the necessary code inside the class where you defined the property's getter and setters.

For the Shape property the component's BackgroundDrawable needs to be changed to change the shape. The table below describes how the ButtonBase component is changed for each Shape.

Shape	Image	Drawable
default	default	defaultButtonDrawable or no drawable and set the background color
rounded	rounded	RoundRectShape(CornerArray, null, null) where CornerArray is and Array of 8 floats each with the value 10f.
rectangular	rectangular	RectShape() drawable
oval	oval	OvalShape() drawable

Figure 13: Shape Drawables

The process of changing the component on the Android device is very specific to both the component being changed and the property being added. Review the methods currently available to the component and components with similar properties to determine what code needs to be added or changed.

The next section details the process followed when implementing the Shape property.

3.1. Button Shape Example

In the ButtonBase class add the following constants.

```
  // Constant for shape
  // 10px is the radius of the rounded corners.
  // 10px was chosen for esthetic reasons.
```

```
private static final float ROUNDED_CORNERS_RADIUS = 10f;
private static final float[] ROUNDED_CORNERS_ARRAY = new float[] {
    ROUNDED_CORNERS_RADIUS,ROUNDED_CORNERS_RADIUS, ROUNDED_CORNERS_RADIUS,
    ROUNDED_CORNERS_RADIUS,ROUNDED_CORNERS_RADIUS, ROUNDED_CORNERS_RADIUS,
    ROUNDED_CORNERS_RADIUS,ROUNDED_CORNERS_RADIUS };

// Constant background color for buttons with a Shape other than default
private static final int SHAPED_DEFAULT_BACKGROUND_COLOR = Color.LTGRAY;
```

Replace the updateAppearance() method with the following.

```
// Update appearance based on values of backgroundImageDrawable, backgroundColor and
// shape.
// Images take precedence over background colors.
private void updateAppearance() {
  // If there is no background image,
  // the appearance depends solely on the background color and shape.
  if (backgroundImageDrawable == null) {
    if (shape == Component.BUTTON_SHAPE_DEFAULT) {
      if (backgroundColor == Component.COLOR_DEFAULT) {
      // If there is no background image and color is default,
      // restore original 3D bevel appearance.
      ViewUtil.setBackgroundDrawable(view, defaultButtonDrawable);
      } else {
        // Clear the background image.
        ViewUtil.setBackgroundDrawable(view, null);
        // Set to the specified color (possibly COLOR_NONE for transparent).
        TextViewUtil.setBackgroundColor(view, backgroundColor);
      }
    } else {
      // If there is no background image and the shape is something other than default,
      // create a drawable with the appropriate shape and color.
      setShape();
    }
  } else {
    // If there is a background image
    ViewUtil.setBackgroundImage(view, backgroundImageDrawable);
  }
}
```

Figure 14: Addition to updateAppearance()

Add the setShape() method with the following.

```
// Throw IllegalArgumentException if shape has illegal value.
private void setShape() {
  ShapeDrawable drawable = new ShapeDrawable();
  // Set color of drawable.
  drawable.getPaint().setColor((backgroundColor == Component.COLOR_DEFAULT)
                               ? shapedDefaultBackgroundColor : backgroundColor);
  // Set shape of drawable.
  switch (shape) {
    case Component.BUTTON_SHAPE_ROUNDED:
      drawable.setShape(new RoundRectShape(ROUNDED_CORNERS_ARRAY, null, null));
      break;
    case Component.BUTTON_SHAPE_RECT:
      drawable.setShape(new RectShape());
      break;
    case Component.BUTTON_SHAPE_OVAL:
      drawable.setShape(new OvalShape());
```

```
        break;
      default:
        throw new IllegalArgumentException();
    }
    // Set drawable to the background of the button.
    view.setBackgroundDrawable(drawable);
    view.invalidate();
  }
```

Figure 15: Addition to setShape()

Now when the Shape of a Button is changed both the Mock button and the Button on the Android device should change.

4. Update Version Numbers

4.1 YaVersion

The YaVersion class (in com.google.appinventor.components.common) defines the Young Android System version number, Blocks Language version number and Component version numbers. If the Blocks Language or any of the Components were updated in the previous sections then their version numbers and the Young Android System version number needs to be increased. There are also instructions, in the class, describing updating each of these values.

For the Button Shape example the ButtonBase component was updated, therefore the ButtonBase component version number needs to be increased along with the version numbers of components that are subclassed of ButtonBase and the Young Android System version number. Add the following gray code to the YaVersion class.

```
// ............................ Young Android System Version Number
// ............................

 // YOUNG_ANDROID_VERSION must be incremented when either the blocks language or a  \
 // component changes.
 // TODO(lizlooney) - should this version number be generated so that it is
 // automatically incremented when the blocks language or a component changes?

 // For YOUNG_ANDROID_VERSION 2:
 // - The Logger component was removed. The Notifier component should be used instead.
 // - TINYWEBDB_COMPONENT_VERSION was incremented to 2.
       :
       :
       :
 // For YOUNG_ANDROID_VERSION 54:
 // - BUTTON_COMPONENT_VERSION was incremented to 4.
 // - CONTACTPICKER_COMPONENT_VERSION was incremented to 4.
 // - IMAGEPICKER_COMPONENT_VERSION was incremented to 4.
 // - LISTPICKER_COMPONENT_VERSION was incremented to 5.
 // - PHONENUMBERPICKER_COMPONENT_VERSION was incremented to 4.
 public static final int YOUNG_ANDROID_VERSION = 54;


 // For BUTTON_COMPONENT_VERSION 2:
 // - The Alignment property was renamed to TextAlignment.
 // For BUTTON_COMPONENT_VERSION 3:
```

```
  // - The LongClick event was added.
  // For BUTTON_COMPONENT_VERSION 4:
  // - The Shape property was added.
  public static final int BUTTON_COMPONENT_VERSION = 4;

  // For CONTACTPICKER_COMPONENT_VERSION 2:
  // - The Alignment property was renamed to TextAlignment.
  // For CONTACTPICKER_COMPONENT_VERSION 3:
  // - The method Open was added.
  // For CONTACTPICKER_COMPONENT_VERSION 4:
  // - The Shape property was added.
  public static final int CONTACTPICKER_COMPONENT_VERSION = 4;

  // For IMAGEPICKER_COMPONENT_VERSION 2:
  // - The Alignment property was renamed to TextAlignment.
  // For IMAGEPICKER_COMPONENT_VERSION 3:
  // - The method Open was added.
  // For IMAGEPICKER_COMPONENT_VERSION 4:
  // - The Shape property was added.
  public static final int IMAGEPICKER_COMPONENT_VERSION = 4;

  // For LISTPICKER_COMPONENT_VERSION 2:
  // - The Alignment property was renamed to TextAlignment.
  // For LISTPICKER_COMPONENT_VERSION 3:
  // - The SelectionIndex read-write property was added.
  // For LISTPICKER_COMPONENT_VERSION 4:
  // - The method Open was added.
  // For LISTPICKER_COMPONENT_VERSION 5:
  // - The Shape property was added.
  public static final int LISTPICKER_COMPONENT_VERSION = 5;

  // For PHONENUMBERPICKER_COMPONENT_VERSION 2:
  // - The Alignment property was renamed to TextAlignment.
  // For PHONENUMBERPICKER_COMPONENT_VERSION 3:
  // - The method Open was added.
  // For PHONENUMBERPICKER_COMPONENT_VERSION 4:
  // - The Shape property was added.
  public static final int PHONENUMBERPICKER_COMPONENT_VERSION = 4;
```

Figure 16: Addition to YaVersion Class

4.2 YoungAndroidFormUpgrader

As stated in the instructions in YaVersion, if a component version is updated code must be added to the YoungAndroidFormUpgrade (in com.google.appinventor.client.youngandroid). For the Button Shape example the following gray code needs to be added to the YoungAndroidFormUpgrader class.

```
private static int upgradeButtonProperties(Map<String, JSONValue> componentProperties,
    int srcCompVersion) {
  if (srcCompVersion < 2) {
    // The Alignment property was renamed to TextAlignment.
    handlePropertyRename(componentProperties, "Alignment", "TextAlignment");
    // Properties related to this component have now been upgraded to version 2.
    srcCompVersion = 2;
  }
  if (srcCompVersion < 3) {
    // The LongClick event was added.
```

```
      // No properties need to be modified to upgrade to version 3.
      srcCompVersion = 3;
    }
    if (srcCompVersion < 4) {
      // The Shape property was added.
      // No properties need to be modified to upgrade to version 4.
      srcCompVersion = 4;
    }
    return srcCompVersion;
  }

  private static int upgradeContactPickerProperties(Map<String, JSONValue>
      componentProperties, int srcCompVersion) {
    if (srcCompVersion < 2) {
      // The Alignment property was renamed to TextAlignment.
      handlePropertyRename(componentProperties, "Alignment", "TextAlignment");
      // Properties related to this component have now been upgraded to version 2.
      srcCompVersion = 2;
    }
    if (srcCompVersion < 3) {
      // The Open method was added.  No changes are needed.
      srcCompVersion = 3;
    }
    if (srcCompVersion < 4) {
      // The Shape property was added.
      // No properties need to be modified to upgrade to version 4.
      srcCompVersion = 4;
    }
    return srcCompVersion;
  }

  private static int upgradeImagePickerProperties(Map<String, JSONValue>
      componentProperties, int srcCompVersion) {
    if (srcCompVersion < 2) {
      // The Alignment property was renamed to TextAlignment.
      handlePropertyRename(componentProperties, "Alignment", "TextAlignment");
      // Properties related to this component have now been upgraded to version 2.
      srcCompVersion = 2;
    }
    if (srcCompVersion < 3) {
      // The Open method was added.  No changes are needed.
      srcCompVersion = 3;
    }
    if (srcCompVersion < 4) {
      // The Shape property was added.
      // No properties need to be modified to upgrade to version 4.
      srcCompVersion = 4;
    }
    return srcCompVersion;
  }

  private static int upgradeListPickerProperties(Map<String, JSONValue>
      componentProperties, int srcCompVersion) {
    if (srcCompVersion < 2) {
      // The Alignment property was renamed to TextAlignment.
      handlePropertyRename(componentProperties, "Alignment", "TextAlignment");
      // Properties related to this component have now been upgraded to version 2.
```

```
      srcCompVersion = 2;
    }
    if (srcCompVersion < 3) {
      // The SelectionIndex property was added.  No changes are needed.
      srcCompVersion = 3;
    }
    if (srcCompVersion < 4) {
      // The Open method was added.  No changes are needed.
      srcCompVersion = 4;
    }
    if (srcCompVersion < 5) {
      // The Shape property was added.
      // No properties need to be modified to upgrade to version 5.
      srcCompVersion = 5;
    }
    return srcCompVersion;
  }

  private static int upgradePhoneNumberPickerProperties(Map<String, JSONValue>
      componentProperties, int srcCompVersion) {
    if (srcCompVersion < 2) {
      // The Alignment property was renamed to TextAlignment.
      handlePropertyRename(componentProperties, "Alignment", "TextAlignment");
      // Properties related to this component have now been upgraded to version 2.
      srcCompVersion = 2;
    }
    if (srcCompVersion < 3) {
      // The Open method was added.  No changes are needed.
      srcCompVersion = 3;
    }
    if (srcCompVersion < 4) {
      // The Shape property was added.
      // No properties need to be modified to upgrade to version 4.
      srcCompVersion = 4;
    }
    return srcCompVersion;
  }
```

Figure 17: Addition to YoungAndroidFormUpgrader Class

4.3 BlockSaveFile

The BlockSaveFile class (in openblocks.yacodeblocks) must also be updated. If any component version numbers were increased then add code to upgradeComponentBlocks().

Add the following gray code to upgradeComponentBlocks() for the Button Shape example.

```
  private int upgradeButtonBlocks(int blkCompVersion, String componentName) {
    if (blkCompVersion < 2) {
      // The Alignment property was renamed to TextAlignment.
      handlePropertyRename(componentName, "Alignment", "TextAlignment");
      // Blocks related to this component have now been upgraded to version 2.
      blkCompVersion = 2;
    }
    if (blkCompVersion < 3) {
      // The LongClick event was added.
```

```
      // No blocks need to be modified to upgrade to version 3.
      blkCompVersion = 3;
    }
    if (blkCompVersion < 4) {
      // The Shape property was added.
      // No blocks need to be modified to upgrade to version 4.
      blkCompVersion = 4;
    }
    return blkCompVersion;
  }

  private int upgradeContactPickerBlocks(int blkCompVersion, String componentName) {
    if (blkCompVersion < 2) {
      // The Alignment property was renamed to TextAlignment.
      handlePropertyRename(componentName, "Alignment", "TextAlignment");
      // Blocks related to this component have now been upgraded to version 2.
      blkCompVersion = 2;
    }
    if (blkCompVersion < 3) {
      // The Open method was added, which does not require changes.
      blkCompVersion = 3;
    }
    if (blkCompVersion < 4) {
      // The Shape property was added.
      // No blocks need to be modified to upgrade to version 4.
      blkCompVersion = 4;
    }
    return blkCompVersion;
  }

  private int upgradeImagePickerBlocks(int blkCompVersion, String componentName) {
    if (blkCompVersion < 2) {
      // The Alignment property was renamed to TextAlignment.
      handlePropertyRename(componentName, "Alignment", "TextAlignment");
      // Blocks related to this component have now been upgraded to version 2.
      blkCompVersion = 2;
    }
    if (blkCompVersion < 3) {
      // The Open method was added, which does not require changes.
      blkCompVersion = 3;
    }
    if (blkCompVersion < 4) {
      // The Shape property was added.
      // No blocks need to be modified to upgrade to version 4.
      blkCompVersion = 4;
    }
    return blkCompVersion;
  }

  private int upgradeListPickerBlocks(int blkCompVersion, String componentName) {
    if (blkCompVersion < 2) {
      // The Alignment property was renamed to TextAlignment.
      handlePropertyRename(componentName, "Alignment", "TextAlignment");
      // Blocks related to this component have now been upgraded to version 2.
      blkCompVersion = 2;
    }
    if (blkCompVersion < 3) {
```

```
    // The SelectionIndex property was added, which does not require changes.
    blkCompVersion = 3;
  }
  if (blkCompVersion < 4) {
    // The Open method was added, which does not require changes.
    blkCompVersion = 4;
  }
  if (blkCompVersion < 5) {
    // The Shape property was added.
    // No blocks need to be modified to upgrade to version 5.
    blkCompVersion = 5;
  }
  return blkCompVersion;
}

private int upgradePhoneNumberPickerBlocks(int blkCompVersion, String componentName) {
  if (blkCompVersion < 2) {
    // The Alignment property was renamed to TextAlignment.
    handlePropertyRename(componentName, "Alignment", "TextAlignment");
    // Blocks related to this component have now been upgraded to version 2.
    blkCompVersion = 2;
  }
  if (blkCompVersion < 3) {
    // The Open method was added, which does not require changes.
    blkCompVersion = 3;
  }
  if (blkCompVersion < 4) {
    // The Shape property was added.
    // No blocks need to be modified to upgrade to version 4.
    blkCompVersion = 4;
  }
  return blkCompVersion;
}
```

Figure 18: Addition to BlockSaveFile Class

5. Deprecating methods and events and properties

Sometimes, in ongoing development, it's necessary to deprecate an existing event, method or property. This removes the element from the blocks palette and drawer. If an existing project that contains the block is read in, the block will be outlined in red and made inactive, indicating that the user should remove it.

To deprecate a block add the @Deprecated annotation to the component java file. In addition, if this is a designer property, comment out the @Designerproperty annotation. For example, here's how we deprecated the Camera.useFront property (both the getter and the setter):

```
/**
  * Returns true if the front-facing camera is to be used (when available)
  *
  * @return {@code true} indicates front-facing to be used, {@code false} by default
  */
 @Deprecated
 @SimpleProperty(category = PropertyCategory.BEHAVIOR)
 public boolean UseFront() {
       return useFront;
```

```
  }

  /**
   * Specifies whether the front-facing camera should be used (when available)
   *
   * @param front
   *              {@code true} for front-facing camera, {@code false} for default
   */
  @Deprecated
  // Hide the deprecated property from the Designer
  //  @DesignerProperty(editorType = PropertyTypeConstants.PROPERTY_TYPE_BOOLEAN, defaultValue =
"False")
  @SimpleProperty(description = "Specifies whether the front-facing camera should be used (when
available). "
        + "If the device does not have a front-facing camera, this option will be ignored "
        + "and the camera will open normally.")
  public void UseFront(boolean front) {
        useFront = front;
  }
```

Also update versioning.js to record that the deprecation was done -- although “no upgrade was necessary”. Here’s how that was done in for Camera.useFront for version 3 of the Camera component.

```
 "Camera": {
      ...

     // AI2: The UseFront property was removed
     3: "noUpgrade"

      ...

 }, // End Camera upgraders
```

6. Internationalization

Names of methods and events and properties need to internationalized so that they can appear in several languages. Don’t use hard-coded strings for these names. Instead, add variables in OdeMessages.java. Similarly for component descriptions.

Add an entry for each new property/event/method into OdeMessages.java iff a property with that name doesn't already exist (so if you are adding a property that has the same name as another property in a different component, (even in a different language) you don't do it a second time). For example, to add the "Foo" property you would add:

```
@defaultMessage("Foo")
@description("This the name of the wonderful Foo property")
String FooProperties();
```

The localized strings for the different languages are in the /appengine/src/com/google/appinventor /client.OdeMessages<language>.properties files. The system is designed so that if there is no translation, then English will be used. So you don’t need to supply translations when you implement the properties; these can be added later.

If you edit the description of a component (but not yet a property,method or event of that component) you must also find and update the description in OdeMessages.java

Published by Google Drive – Report Abuse – Updated automatically every 5 minutes