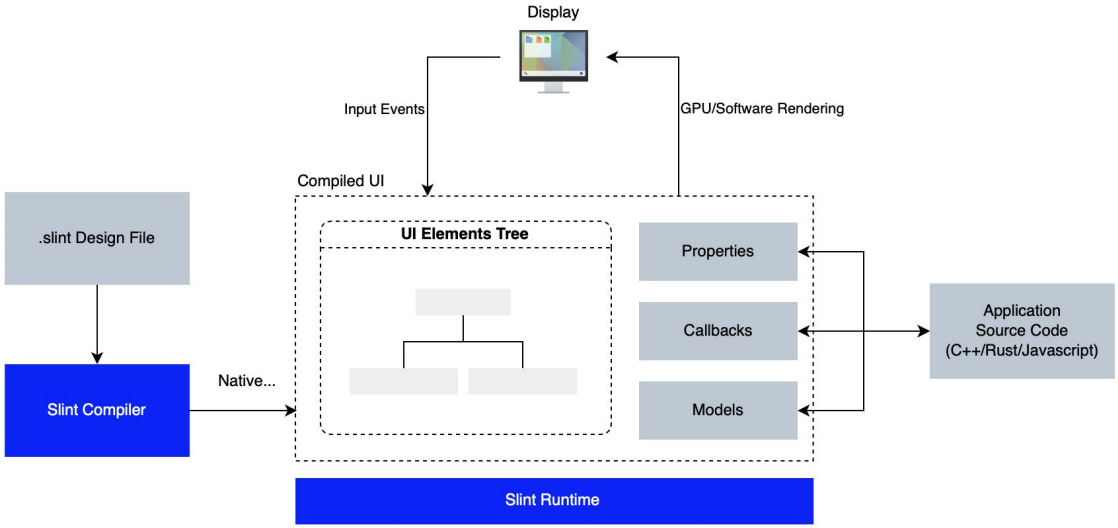
Introduction to Slint

What is Slint?

- The Slint design markup language describes extensible graphical user interfaces using the **Slint framework**
- The Slint language enforces a separation of user interface from business logic, using interfaces you can define for your project
- It only describes the user interface and it is **not a programming language**; the business logic is written in a different programming language using the **Slint API**



The .slint file

- UI components as tree elements
- Elements vs Components
- Elements with pre-defined names:
  - root -> refers to the outermost element of a component
  - self -> refers to the current element
  - parent -> refers to the parent element of the current element

```
component MyButton inherits Text {
    color: black;
    // ...
}

export component MyApp inherits Window {
    preferred-width: 200px;
    preferred-height: 100px;
    Rectangle {
        width: 200px;
        height: 100px;
        background: green;
    }
    MyButton {
        x:0;y:0;
        text: "hello";
    }
    MyButton {
        y:0;
        x: 50px;
        text: "world";
    }
}
```

Element naming

- Elements have properties which you can assign values to
- You can name elements using the `:=` syntax
- Here we assign a string constant “hello” to the first **`MyButton`’s `text`** property

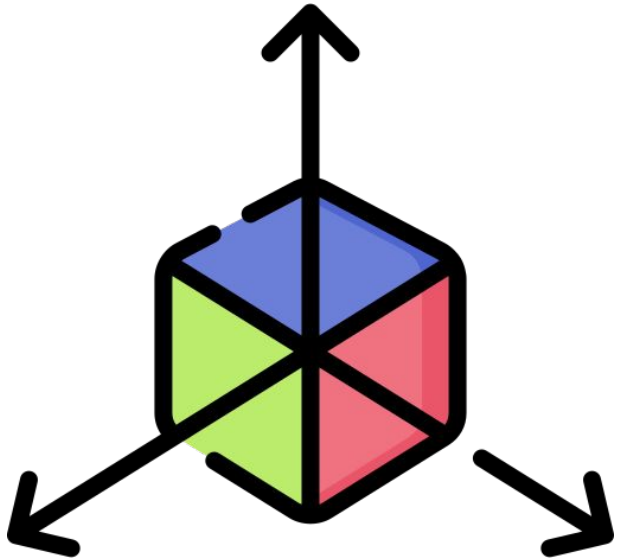
```
component MyButton inherits Text {
    // ...
}

export component MyApp inherits Window {
    preferred-width: 200px;
    preferred-height: 100px;

    hello := MyButton {
        x:0;y:0;
        text: "hello";
    }
    world := MyButton {
        y:0;
        text: "world";
        x: 50px;
    }
}
```

Positioning and layout of the elements

- `x` and `y` properties store the elements coordinates **relative** to their parent element
- The `width` and `height` properties store the size of visual elements
- You can create an entire GUI by placing the elements in two ways:
  - Explicitly - by setting the **`x`**, **`y`**, **`width`**, and **`height`** properties
  - Automatically - by using layout elements



Explicit placement

- Great for static scenes with few elements
- The **default** values for **x** and **y** properties are such that elements are **centered** within their parent
- The **default** values for **`width`** and **`height`** depend on the type of element
- The following elements don't have content and default to fill their parent element when they do not have children:
    - **Rectangle**
    - **TouchArea**
    - **FocusScope**
    - **Flickable**

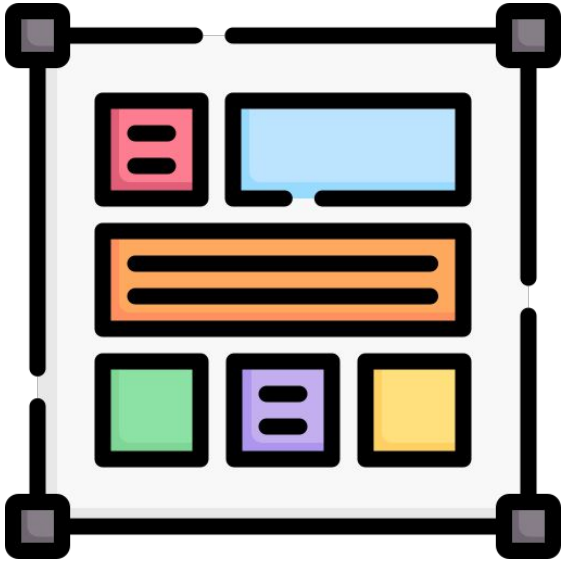
```
// Explicit positioning
export component Example inherits Window {
    width: 200px;
    height: 200px;
    Rectangle {
        x: 100px;
        y: 70px;
        width: parent.width - self.x;
        height: parent.height - self.y;
        background: blue;
        Rectangle {
            x: 10px;
            y: 5px;
            width: 50px;
            height: 30px;
            background: green;
        }
    }
}
```

Automatic placement using layouts

- Slint comes with different layout elements that automatically calculate the position and size of their children:
  - `VerticalLayout` / `HorizontalLayout`: The children are placed along the vertical or horizontal axis
  - `GridLayout`: The children are placed in a grid of columns and rows

- Each element has a **minimum**, a **maximum** size, and a **preferred** size

- The **default** value of these constraint properties may depends on the content of the element



VerticalLayout and HorizontalLayout

- These layouts place their children in a **column** or a **row**
- They stretch or shrink to take the whole space
- You can adjust the element's alignment as needed

```
// Stretch by default
export component Example inherits Window {
    width: 200px;
    height: 200px;
    HorizontalLayout {
        Rectangle { background: blue; min-width: 20px; }
        Rectangle { background: yellow; min-width: 30px; }
    }
}
```

```
// Unless an alignment is specified
export component Example inherits Window {
    width: 200px;
    height: 200px;
    HorizontalLayout {
        alignment: start;
        Rectangle { background: blue; min-width: 20px; }
        Rectangle { background: yellow; min-width: 30px; }
    }
}
```

Basic Slint types

Type	Description
`int`	Signed integral number.
`float`	Signed, 32-bit floating point number. Numbers with a `%` suffix are automatically divided by 100, so for example `30%` is the same as `0.30`.
`bool`	boolean whose value can be either `true` or `false`.
`string`	UTF-8 encoded, reference counted string.
`color`	RGB color with an alpha channel, with 8 bit precision for each channel. CSS color names as well as the hexadecimal color encodings are supported, such as #RRGGBBAA or #RGB.
`brush`	A brush is a special type that can be either initialized from a color or a gradient specification. See the Colors and Brushes Section for more information.
`physical-length`	This is an amount of physical pixels. To convert from an integer to a length unit, one can simply multiply by `1px`. Or to convert from a length to a float, one can divide by `1phx`.
`length`	The type used for `x`, `y`, `width` and `height` coordinates. Corresponds to a literal like `1px`, `1pt`, `1in`, `1mm`, or `1cm`. It can be converted to and from length provided the binding is run in a context where there is an access to the device pixel ratio.
`duration`	Type for the duration of animations. A suffix like `ms` (millisecond) or `s` (second) is used to indicate the precision.
`angle`	Angle measurement, corresponds to a literal like `90deg`, `1.2rad`, `0.25turn`
`easing`	Property animation allow specifying an easing curve. Valid values are `linear` (values are interpolated linearly) and the four common cubiz-bezier functions known from CSS: `ease`, `ease_in`, `ease_in_out`, `ease_out`.
`percent`	Signed, 32-bit floating point number that is interpreted as percentage. Literal number assigned to properties of this type must have a `%` suffix.
`image`	A reference to an image, can be initialized with the `@image-url("...")` construct
`relative-font-size`	Relative font size factor that is multiplied with the `Window.default-font-size` and can be converted to a `length`.

Properties

- Built-in elements come with common properties such as color or dimensional properties
- In addition to the existing properties, define extra properties by specifying the name, the type, and optionally a default value

```
export component Example {
    // declare a property of type int with the name `my-property`
    property<int> my-property;

    // declare a property with a default value
    property<int> my-second-property: 42;
}
```

Access qualifiers

Annotate custom the properties with a qualifier that specifies how the property can be read and written:

- **private** (the default): The property can only be accessed from within the component
- **in**: The property is an input; it can be set and modified by the user of this component
- **out**: An output property that can only be set by the component; it's read-only for the users of the components
- **in-out**: The property can be read and modified by everyone

All properties declared at the top level of a component that aren't **`private`** are accessible from the outside when using a component as an element, or via the language bindings from the business logic.

```
export component Button {
    // This is meant to be set by the user of the component.
    in property <string> text;
    // This property is meant to be read by the user of the component.
    out property <bool> pressed;
    // This property is meant to both be changed by the user and the component itself.
    in-out property <bool> checked;

    // This property is internal to this component.
    private property <bool> has-mouse;
}
```

Bindings

- The binding expression is **automatically** re-evaluated when properties accessed in the expression change
- Internally, a **dependency** is registered for any property accessed while evaluating a binding
- When a property changes, the dependencies are notified and all dependent bindings are marked as **dirty**

```
import { Button } from "std-widgets.slint";
export component Example inherits Window {
    preferred-width: 50px;
    preferred-height: 50px;
    Button {
        property <int> counter: 3;
        clicked => { self.counter += 3 }
        text: self.counter * 2;
    }
}
```

Modules

- Components declared in a `.slint` file can be used as elements in other `.slint` files, by means of exporting and importing them
- By default, every type declared in a `.slint` file is private. The `export` keyword changes this.
- A way of exporting a component is to declare it **exported** right away
- In the event that two files export a type under the same name, we have the option of **assigning** a different name at import time

```
export component Button inherits Rectangle {
    // ...
}
```

```
import { Button } from "./button.slint";

export component App inherits Rectangle {
    // ...
    Button {
        // ...
    }
}
```

```
import { Button } from "./button.slint";
import { Button as CoolButton } from
"../other_theme/button.slint";

export component App inherits Rectangle {
    // ...
    CoolButton {} // from other_theme/button.slint
    Button {} // from button.slint
}
```

Module syntax

```
import { export1 } from "module.slint";

import { export1, export2 } from "module.slint";

import { export1 as alias1 } from "module.slint";

import { export1, export2 as alias2, /* ... */ } from "module.slint";
```

```
// Export declarations
export component MyButton inherits Rectangle { /* ... */ }

// Export lists
component MySwitch inherits Rectangle { /* ... */ }
export { MySwitch };
export { MySwitch as Alias1, MyButton as Alias2 };

// Re-export all types from other module
export * from "other_module.slint";
```

UI Example

```
import { AboutSlint, Button, VerticalBox }from "std-widgets.slint";
export component Demo inherits Window{
 background: white;
  VerticalBox {
      alignment: start;
      Text {
          text: "Hello World!";
          font-size: 24px;
          horizontal-alignment: center;
          color: black;
      }

      Image {
         vertical-alignment: ImageVerticalAlignment.center;
         horizontal-alignment:  ImageHorizontalAlignment.center;
         source: @image-url("https://slint.dev/logo/slint-logo-full-light.svg");
      }
      HorizontalLayout {alignment: center; Button {primary: true;
          text: "OK!";} }
  }
}
```

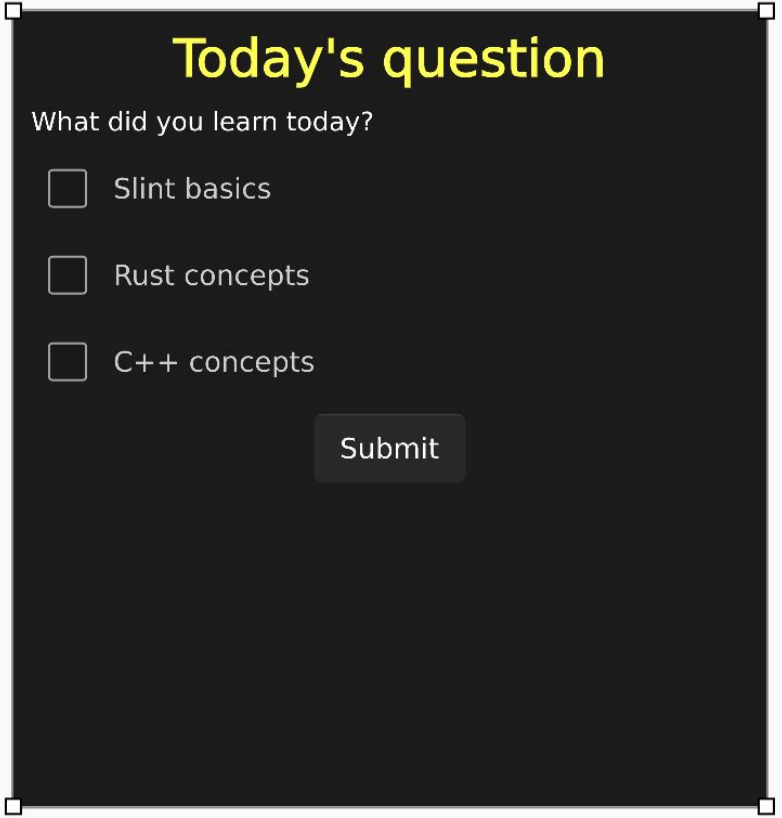
Challenge time

SlintPad: https://slintpad.com

UI pre-built elements: https://releases.slint.dev/1.1.0/docs/slint/src/builtins/elements

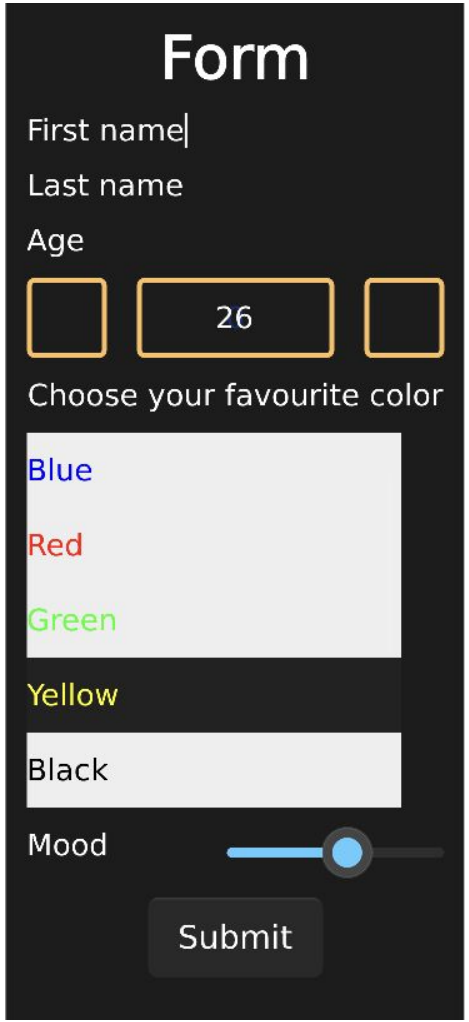
UI widgets: https://releases.slint.dev/1.1.0/docs/slint/src/builtins/widgets

Replicate the following UI



Replicate this UI while following these guidelines:

- “First name” and “Last name” fields are TextInputs
- Age counter is represented by a SpinBox
- The text inside the SpinBox needs to be represented by a separate component defined by you
- The color list needs to be a Listview



Expressions

- A powerful way to declare relationships and connections in your user interface
- When the properties change, the expression is automatically re-evaluated and a new value is assigned to the property the expression is associated with
- Arithmetic in expression with numbers works like in most programming language with the operators ***, +, -, /**
- String concatenation is made with **+**
- Access an element's properties by using its name, followed by a **.** and the property name

```
export component Example {
    in-out property <int> p: 1 * 2 + 3 * 4; // same as (1 * 2) + (3 * 4)
}
```

```
export component Example {
    foo := Rectangle {
        x: 42px;
    }
    x: foo.x;
}
```

Functions

- Declare helper functions with the **`function`** keyword
- Functions are private by default, but can be made public with the **`public`** annotation

```
export component Example {
    in property <int> min;
    in property <int> max;
    public function inbound(x: int) -> int {
        return Math.min(root.max, Math.max(root.min, x));
    }
}
```

Callbacks

- **Callbacks** are invoked by “calling” them like you would call a function
- You react to callback invocation by declaring a handler using the `=>` arrow syntax
- The built-in **`TouchArea`** element declares a **`clicked`** callback, that’s invoked when the user touches the rectangular area covered by the element, or clicks into it with the mouse

```
export component Example inherits Rectangle {
    // declares a callback with a return value
    callback hello(int, int) -> int;
    hello(aa, bb) => { aa + bb }
}
```

```
export component Example inherits Rectangle {
    // declare a callback
    callback hello;

    area := TouchArea {
        // sets a handler with `=>`
        clicked => {
            // emit the callback
            root.hello()
        }
    }
}
```

Repetition

- Use the **`for-in`** syntax to create an element multiple times
- The name will be available for lookup within the element and is going to be like a pseudo-property set to the value of the model
- The model can be of the following type:
  - an integer, in which case the element will be repeated that amount of time
  - an array type or a model declared natively, in which case the element will be instantiated for each element in the array or model

`Syntax:`

```
for name[index] in model : id := Element { ... }
```

```
export component Example inherits Window {
    preferred-width: 300px;
    preferred-height: 100px;
    for my-color[index] in [ #e11, #1a2, #23d ]: Rectangle {
        height: 100px;
        width: 60px;
        x: self.width * index;
        background: my-color;
    }
}
```

Conditional element

- The `if` construct instantiates an element only if a given condition is **true**
- Syntax: `if condition: id := Element { ... }`

```
export component Example inherits Window {
    preferred-width: 50px;
    preferred-height: 50px;
    if area.pressed : foo := Rectangle { background: blue; }
    if !area.pressed : Rectangle { background: red; }
    area := TouchArea {}
}
```

Slint components in Rust

- A component is instantiated using the **fn new() -> Self** function
- After instantiating the component, call **ComponentHandle::run()** on show it on the screen and spin the event loop to react to input events
- To show multiple components simultaneously, call **ComponentHandle::show()** on each instance
- For each top-level property of the components we have a getter and a setter:
  - `fn get_<property_name>(&self) -> <PropertyType>`
  - `fn set_<property_name>(&self, value: <PropertyType>)`

Threading and Event loop

- For platform-specific reasons, the event loop must run in the **main thread**
- All the components must be created in the **same thread** as the thread the event loop is running or is going to run
- The minimum amount of work should be performed in the main thread and **delegate** the actual logic to another thread to avoid blocking animations
- Use the **invoke_from_event_loop** function to communicate from your worker thread to the UI thread

Bonus: https://releases.slint.dev/1.0.2/docs/tutorial/rust/