

User Interface Training

With MICROEJ SDK

© MICROEJ



MICROEJ[®]

DISCLAIMER

All rights reserved. Information, technical data and tutorials contained in this document are proprietary under copyright Law of Industrial Smart Software Technology (MicroEJ S.A.) operating under the brand name MicroEJ®. Without written permission from MicroEJ S.A., copying or sending parts of the document or the entire document by any means to third parties is not permitted. Granted authorizations for using parts of the document or the entire document do not mean MicroEJ S.A. gives public full access rights.

The information contained herein is not warranted to be error-free.

MicroEJ® and all relative logos are trademarks or registered trademarks of MicroEJ S.A. in France and other Countries.

Java™ is Sun Microsystems' trademark for a technology for developing application software and deploying it in cross-platform, networked environments. When it is used in this site without adding the “™” symbol, it includes implementations of the technology by companies other than Sun. Java™, all Java-based marks and all related logos are trademarks or registered trademarks of Sun Microsystems Inc, in the United States and other Countries.

Other trademarks are proprietary of their respective owners.

Agenda

WHAT YOU WILL LEARN

By the end of this training, you will be able to:

- Compose a GUI using widgets
- Tune the look of widgets using style, images and fonts
- Change the rendering of widgets
- Interact with the GUI (e.g., touch or button)
- Animate a GUI

REQUIREMENTS

- IntelliJ IDEA or Android Studio
- MICROEJ SDK 6: follow the installation steps at <https://docs.microej.com/en/latest/SDK6UserGuide/install.html>
- The source code of the training, packaged with this document
- Access to internet and to the following websites:
 - <https://repository.microej.com/>
 - <https://forge.microej.com/>
 - <https://docs.microej.com/>
 - <https://github.com/MicroEJ/>

Introduction

FOREWORDS

INTENDED AUDIENCE

- Java developers designing GUIs for embedded devices

PRE-REQUISITES

- Basic knowledge of application development with MicroEJ

GOAL

- Understand the key concepts of the UI framework

OVERVIEW OF THE GUI LIBRARIES

MICROUI (*Micro User Interface*)

- Framework for operating with outputs and user-inputs (e.g., pixelated display, touch, buttons, etc.).

DRAWING

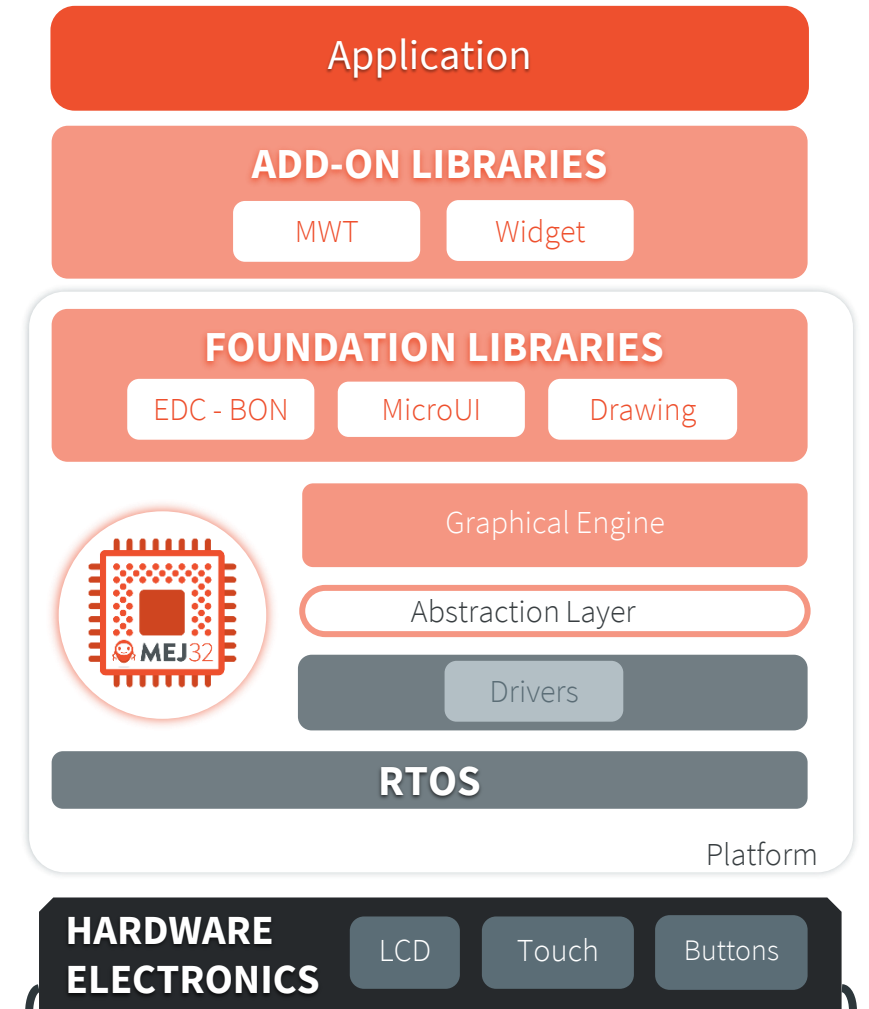
- Extension of MicroUI for advanced drawing.

MWT (*Micro Widget Toolkit*)

- Toolkit for creating widgets.

WIDGET

- Collection of widgets commonly found in user interfaces.



Code Setup

FOREWORDS

- The training will be conducted within an existing application that simulates the user interface of a basic watch.
- It is not required to have a full understanding of all the classes used in this demo application. We will focus only on the specific challenges that matter for the training.
- We will benefit from the code and resources already provided in the project (images, menus, navigation, etc.). The training's outcome will look fancier than starting from an empty project.

IMPORT THE SOURCE CODE

- Go to *File > Open ...*
- Browse to the directory with the source code of the training.
- Click *OK*.

GIT REPOSITORY SETUP

- The project is self-contained in a git repository.
- The git repository will help us moving along the steps of the training, using the **step/...** branches.

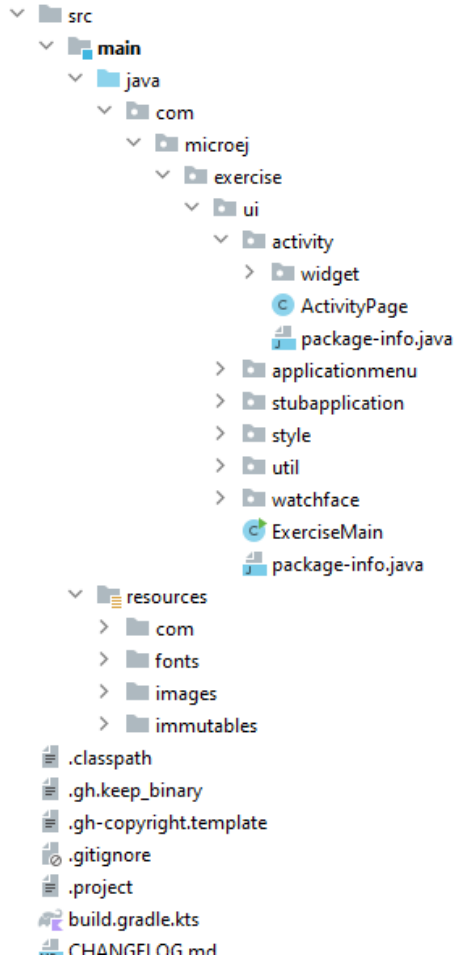
MOVING INTO STEPS

- The training is composed of 10 progressive steps.
- Each branch **step/...** of the repository is the start of a step.
- The branch **step/1** is the branch to start with.
- When the first step is complete, move to branch **step/2**, and so on.
- The branches **solution/...** contain the solution for a given step.
- For example, the branch **solution/1** contains the solution for the step 1.
- You can commit your work before moving to the next step if needed.
- To change branch and drop local changes : `git checkout -f BRANCH-NAME`
- The final branch with all steps completed is on branch **solution/10**

STEP INSTRUCTIONS

- The instructions, objective and details needed to complete a step will be given in this presentation.
- This information is also available within the project files.
- If you are lost, you can locate the step instructions by searching for the text **STEP X** in this project files (**match upper case**). For example, for the first step, look for **STEP 1**.
- **Note:** The *Find in Files* action may not find the “STEP X” anchor text after changing branch. Make sure to do *File > Reload All From Disk* to sync the files in the editor.
- The place to write code is marked with this text: `// WRITE CODE HERE`
- Every graphics resource or utility code is already provided in the project, so that the trainee can focus on the code that matters for the training.

OVERVIEW OF THE PROJECT



- The application features 4 pages:
 - A watchface
 - An application menu (list)
 - An activity monitoring application, « Activity »
 - A stub application
- The code for a page is in a well-identified package, that contains:
 - A ***Page** class, that creates and setups the content of the page
 - The widgets used in this page
- Additional packages:
 - *style*: the classes for styling management
 - *util*: utility classes for the purposes of the training

RUNNING THE APPLICATION IN THE SIMULATOR

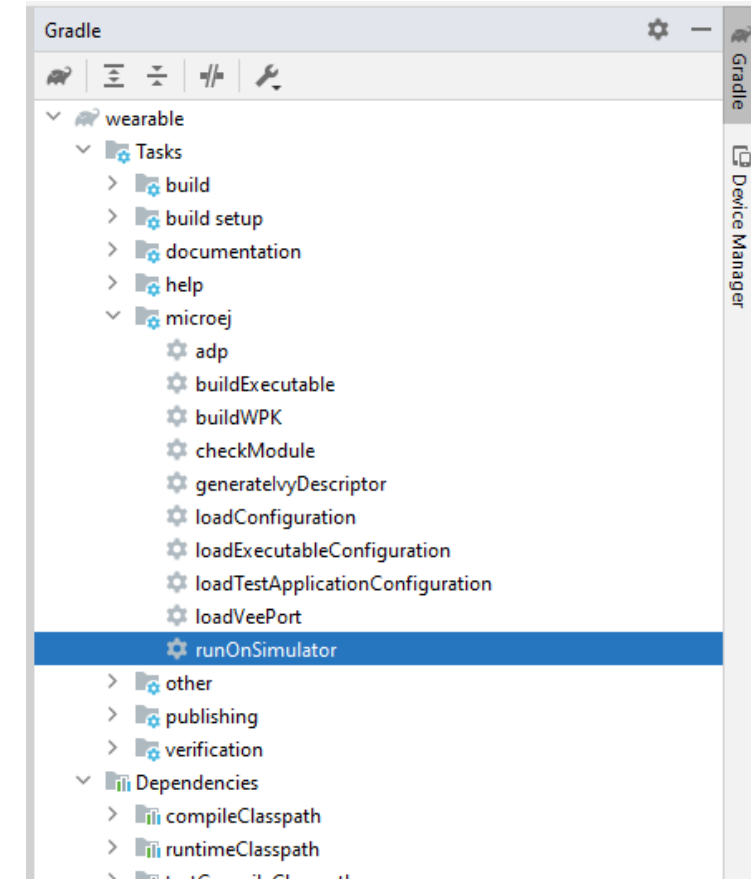
To run the application in simulation:

1. Open the Gradle pane
2. Select *Tasks > microej > runOnSimulator*

The application launches in the simulator.

Note: if a message prompts you to accept MICROEJ SDK End-User License Agreement (EULA), please refer to our documentation for details on the available options for accepting the SDK EULA.

<https://docs.microej.com/en/latest/SDK6UserGuide/licenses.html#sdk-eula-acceptation>



APPLICATION FLOW

- At *step/1*, the so-called opening page “Digital watchface” is very simple. It will be enhanced in next steps.
- You can navigate in the application using the physical buttons and the touch:
 - Button to go back and forth from the watchface to the application menu.
 - Touch to select an item in the application menu.
 - Button to exit an application.



NEED HELP?

- Check the appendix at the end of this presentation for links to the MicroEJ online documentation.

Widgets

DEFINITION

- Widget: an element of a GUI that displays an information or provides a way to interact with the user.
- Widgets are semantic elements of the GUI. They help describe the structure and function of the content.
- Widgets classes are subclasses of the class Widget, they are contained in another widget or a Desktop.
- Desktop: a top-level object that can be displayed on a Display. It contains a Widget, and at most one desktop is shown on a Display at any given time.
- Desktop automatically triggers the layout and rendering phases for itself and its children.

THE WIDGET LIBRARY

- The [Widget library](#) contains a collection of widgets commonly used in GUIs.
- To use the Widget library in an application, add the following dependency to the Gradle build file:

```
implementation("ej.library.ui:widget:5.0.0")
```

Note: the [Widget Demo](#) provides examples for these widgets.

LIST OF WIDGETS

- **Label**: a widget that displays a text.

example: `Label label = new Label("Some Text");`

- **Button**: a widget that displays a text and reacts to clicks.

example: `Button button = new Button("Some Text");`

- **ImageWidget**: a widget that displays an image.

example: `ImageWidget image = new ImageWidget("/path/to/someImage.png");`

- **ImageButton**: a widget that displays an image and reacts to clicks.

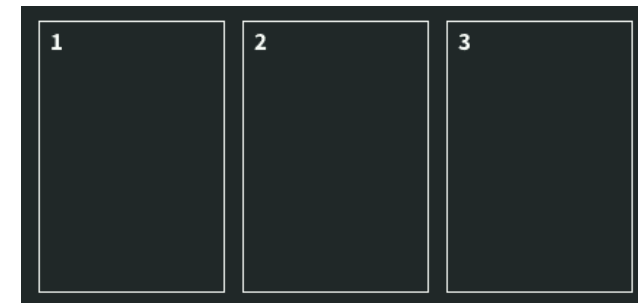
example: `ImageButton imageButton = new ImageButton("/path/to/someImage.png");`

LIST OF CONTAINERS (1)

Definition: A Container is a widget that contains other widgets.

- **List:** a container that lays out widgets horizontally or vertically, on the same row or column.

```
List list = new List(LayoutOrientation.HORIZONTAL);  
list.addChild(new Label("1"));  
list.addChild(new Label("2"));  
list.addChild(new Label("3"));
```



- **Flow:** lays out widgets horizontally or vertically, using multiple rows or columns if necessary.

```
Flow flow = new Flow(LayoutOrientation.HORIZONTAL);  
flow.addChild(new Label("Widget 1"));  
flow.addChild(new Label("Widget 2"));  
flow.addChild(new Label("Widget 3"));  
flow.addChild(new Label("Widget 4"));  
flow.addChild(new Label("Widget 5"));
```



LIST OF CONTAINERS (2)

- **Grid:** lays out widgets in a grid. All widgets have the same width and the same height.

```
Grid grid = new Grid(LayoutOrientation.HORIZONTAL, 3);
grid.addChild(new Label("1"));
grid.addChild(new Label("2"));
grid.addChild(new Label("3"));
grid.addChild(new Label("4"));
grid.addChild(new Label("5"));
grid.addChild(new Label("6"));
```



- **Dock:** stacks the widgets on the edges, in order. The center widget takes the remaining space.

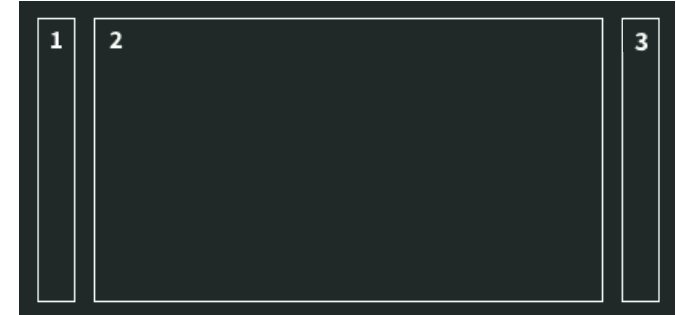
```
Dock dock = new Dock();
dock.addChildOnLeft(new Label("1"));
dock.addChildOnTop(new Label("2"));
dock.addChildOnRight(new Label("3"));
dock.addChildOnBottom(new Label("4"));
dock.addChildOnLeft(new Label("5"));
dock.setCenterChild(new Label("6"));
```



LIST OF CONTAINERS (3)

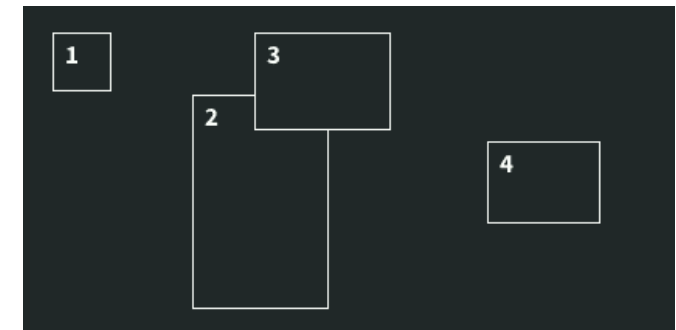
- **SimpleDock:** simple version of the Dock. Stacks at most 3 widgets: 2 on the edges, 1 in the center.

```
SimpleDock simpleDock = new SimpleDock(LayoutOrientation.HORIZONTAL);
simpleDock.setFirstChild(new Label("1"));
simpleDock.setCenterChild(new Label("2"));
simpleDock.setLastChild(new Label("3"));
```



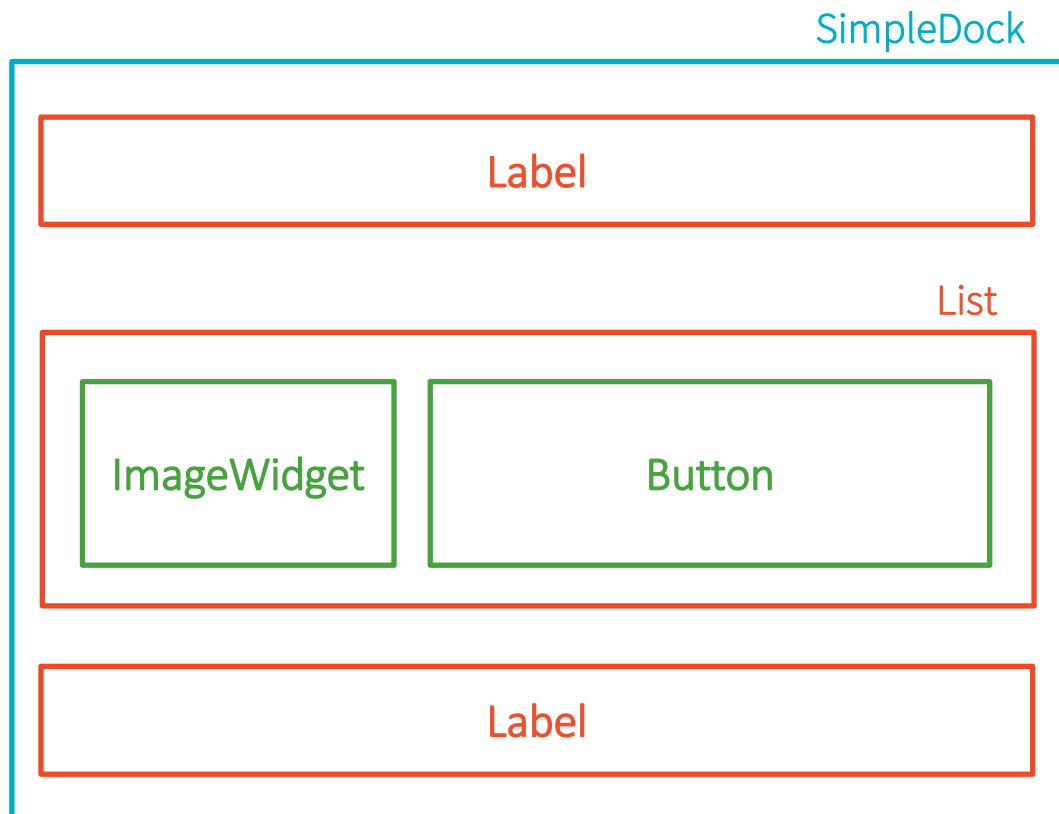
- **Canvas:** lays out the widgets at given bounds (x, y, width, height).

```
Canvas canvas = new Canvas();
canvas.addChild(new Label("1"), 20, 20, 40, 40);
canvas.addChild(new Label("2"), 120, 65, 100, 200);
canvas.addChild(new Label("3"), 170, 20, 100, 80);
canvas.addChild(new Label("4"), 350, 150, 80, 60);
```

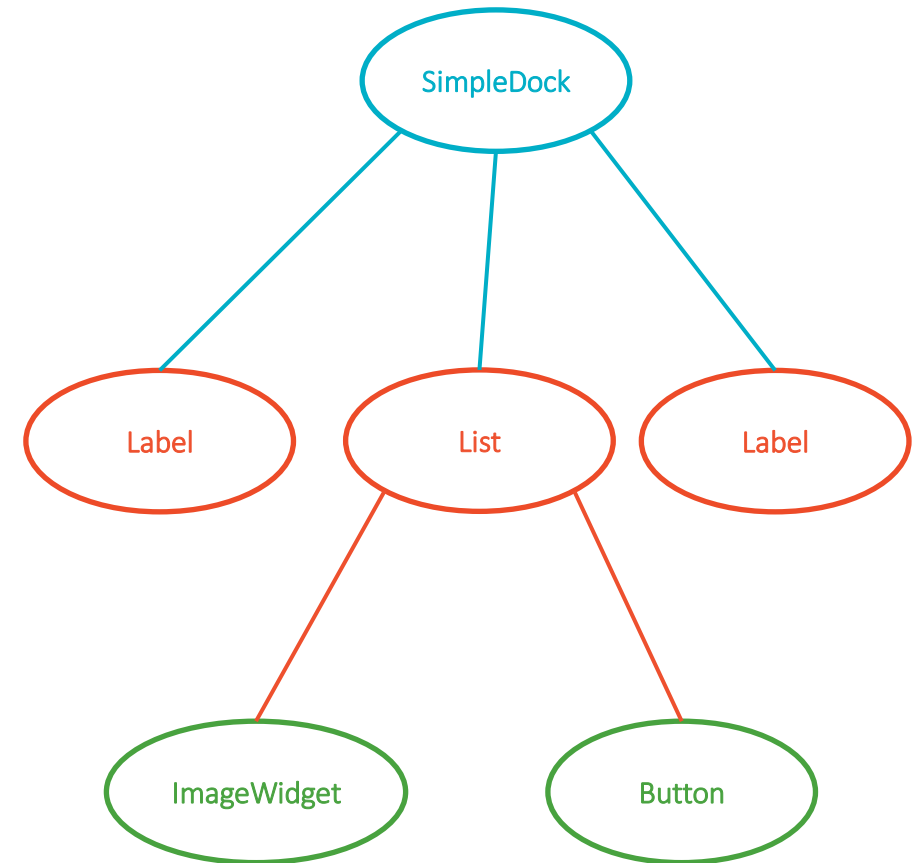


VIEW COMPOSITION

- A view is composed by assembling widgets and containers.
- Example:

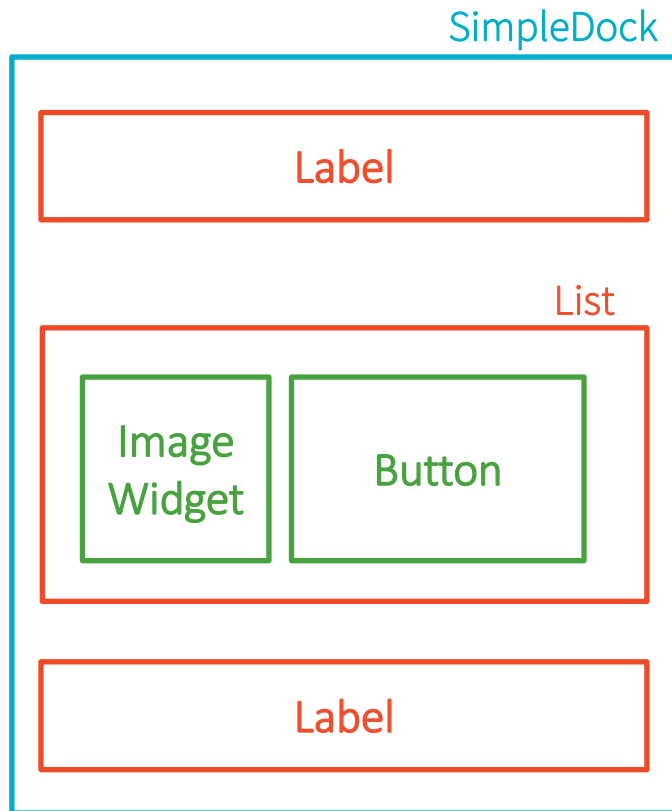


The corresponding widget tree:



VIEW COMPOSITION EXAMPLE

Let's see how it turns into code:



Code

```
SimpleDock simpleDock = new SimpleDock(LayoutOrientation.VERTICAL);

// sets the top child
simpleDock.setFirstChild(new Label("Header"));

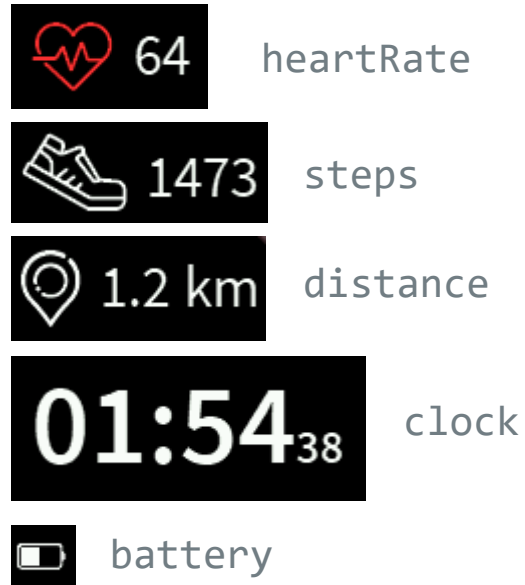
// sets the center child
List list = new List(LayoutOrientation.HORIZONTAL);
list.addChild(new ImageWidget("/images/someImage.png"));
list.addChild(new Button("Button"));
simpleDock.setCenterChild(list);

// sets the bottom child
simpleDock.setLastChild(new Label("Footer"));
```

Coding: step 1

GOAL: COMPOSE THE DIGITAL WATCHFACE

- `git checkout -f step/1`, then search for “STEP 1”.
- You will learn how to compose a view by assembling widgets together.
- It consists of 5 widgets:



For example



- The goal is to display all 5 widgets on the screen, no matter their position (see an example above).
- Note: These are not widgets from the Widget library, but custom widgets developed for this demonstration.

INSTRUCTIONS

To complete the digital watchface:

1. Go to the method `createDigital()` of the class `WatchfacePage`. It is responsible for creating the layout of the digital watchface.
 2. Choose one or more containers from the Widget library (see the [provided containers](#)).
 3. Build a hierarchy by adding the widgets to the containers (look for `addChild()` and `set*Child()` methods).
 4. Return the root container of the resulting widget hierarchy.
- Note: The 5 widgets have already been instantiated in the code to make things easier.

EXAMPLE

- There are a lot of valid solutions, as long as all the 5 widgets are laid out and visible on the display.
- For example, this can be as simple as the following:

```
List list = new List(LayoutOrientation.VERTICAL);  
list.addChild(this.heartRate);  
list.addChild(this.steps);  
list.addChild(this.distance);  
list.addChild(clock);  
list.addChild(this.battery);  
return list;
```

- In this simple example, the 5 widgets are laid out in a vertical list. Another example can be found on the `solution/1` branch.



Style

SEPARATION OF CONTENT AND STYLE

- Widgets describe the structure and function of the content.
- Widgets do not define the style options (colors, fonts, margin, padding, background, etc.).
- We do not recommend to hardcode style options in the widget code. Separation of content and style help keep the GUI implementation flexible.
- The style API is inspired by CSS (Cascading Style Sheets):
 - A stylesheet
 - Selectors
 - Styles

STYLE SHEET

- The stylesheet manages the styles defined by the application.
- The stylesheet determines which style applies to widgets using a « Cascade » algorithm and a selector specificity.

```
CascadingStyleSheet stylesheet = new CascadingStyleSheet();  
desktop.setStyleSheet(stylesheet);
```

SELECTORS

- Selectors are used to select (or find) the widgets to style.
- Selectors determine the widgets to which a style applies.
- Main types of selectors:
 - ClassSelector: selects from a class ID, equivalent to *.class* in CSS.
`new ClassSelector(2) // selects class with ID = 2`
 - TypeSelector: selects from the Java class type or subtype.
`new TypeSelector(Label.class) // selects labels`
 - StateSelector: selects from a state, equivalent to CSS Pseudo-classes like *:active*
`new StateSelector(Button.ACTIVE) // selects active buttons`

STYLES

- The style describes the visual attributes of the widgets.
- Generic attributes are inspired by CSS Properties:
 - Background
 - Border
 - Color
 - Dimension
 - Font
 - Alignment (horizontal and vertical)
 - Margin
 - Padding
 - Dimension

STYLE EDITION

- First, get the style that maps with a given selector in the stylesheet (e.g., the style for labels):

```
EditableStyle style = stylesheet.getSelectorStyle(new TypeSelector(Label.class));
```

- Then, edit the style using the setters of type EditableStyle:

- setBackground(Background) : void
- setBorder(Outline) : void
- setColor(int) : void
- setDimension(Dimension) : void
- setExtraFloat(int, float) : void
- setExtraInt(int, int) : void
- setExtraObject(int, Object) : void
- setFont(Font) : void
- setHorizontalAlignment(int) : void
- setMargin(Outline) : void
- setPadding(Outline) : void
- setVerticalAlignment(int) : void

SET COLOR

- The color of a widget is set with `style.setColor()`
- Example:
`style.setColor(Colors.BLACK);`
- Equivalent to `style.setColor(0x000000)` with RGB values in hexadecimal.

SET BACKGROUND

- The background of a widget is set with `style.setBackground()`

Main types of backgrounds:

- RectangularBackground: a plain rectangular-shaped background with color
`style.setBackground(new RectangularBackground(Colors.BLACK));`
- ImageBackground: a background that displays an image
`style.setBackground(new ImageBackground(someImage));`
- NoBackground: a background that draws nothing (i.e., background is transparent)
`style.setBackground(NoBackground.NO_BACKGROUND);`

SET FONT

- The font of a widget is set with `style.setFont()`
- Example:
`style.setFont(someFont);`

SET ALIGNMENT

- The alignment within a widget is set with `style.setAlignment()`
- Example to set the content of a widget horizontally centered and vertically top-aligned:

```
style.setHorizontalAlignment(Alignment.HCENTER);  
style.setVerticalAlignment(Alignment.TOP);
```
- Horizontal alignment constants: *LEFT, RIGHT, HCENTER*
- Vertical alignment constants: *TOP, BOTTOM, VCENTER*

SET BORDER

- The border of a widget is set with `style.setBorder()`

Main types of borders:

- NoOutline: no border

```
style.setBorder(NoOutline.NO_OUTLINE);
```

- RectangularBorder: a rectangular-shaped colored border with uniform thickness

```
style.setBorder(new RectangularBorder(Colors.RED,5)); // a 5px-thick red border
```

- FlexibleRectangularBorder: a rectangular-shaped colored border with variable thickness

```
style.setBorder(new FlexibleRectangularBorder(Colors.RED,5,0,5,0)); // one thickness per side
```

- RoundedBorder: a round-shaped colored border with uniform thickness and corner radius

```
style.setBorder(new RoundedBorder(Colors.RED,10,5)); // 10px corner-radius, 5px thickness
```

SET PADDING

- The padding of a widget is set with `style.setPadding()`

Main types of padding:

- NoOutline: no padding

```
style.setPadding(NoOutline.NO_OUTLINE);
```

- UniformOutline: sets the same padding for each side

```
style.setPadding(new UniformOutline(10));           // a 10px padding
```

- FlexibleOutline: sets a different padding for each side

```
style.setPadding(new FlexibleOutline(10, 0, 10, 0)); // one padding per side
```

SET MARGIN

- The margin of a widget is set with `style.setMargin()`

Main types of margin:

- NoOutline: no margin

```
style.setMargin(NoOutline.NO_OUTLINE);
```

- UniformOutline: sets the same margin for each side

```
style.setMargin(new UniformOutline(10));           // a 10px margin
```

- FlexibleOutline: sets a different margin for each side

```
style.setMargin(new FlexibleOutline(10, 0, 10, 0)); // one margin per side
```

SET CUSTOM ATTRIBUTES

- For advanced style customization, extra style attributes for specific widgets can be created with `setExtra*(id, value)`.
- Extra style attributes can be integers, floats or objects.

- Example: imagine a widget that uses two colors and two fonts to render

```
// sets black color for the text
style.setColor(Colors.BLACK);
// sets teal color for the icon
style.setExtraInt(0, Colors.TEAL);           // uses extra field with ID = 0
// sets Roboto font for the main text
style.setFont(roboto);
// sets Noto font for the secondary text
style.setExtraObject(1, noto);              // uses extra field with ID = 1
```

GET THE STYLE

- Widgets can retrieve their style by calling the method `getStyle()`;
- All the style attributes are available with getters.
- Example:

```
Style style = getStyle();
Font mainFont = style.getFont();
int textColor = style.getColor();
// icon color is at extra field with ID = 0, and defaults to the text color if not set.
int iconColor = style.getExtraInt(0, textColor);
// secondary font is at extra field with ID = 1, and defaults to the main font if not set.
Font secondsFont = style.getExtraObject(1, Font.class, mainFont);
int verticalAlignment = style.getVerticalAlignment();
```

OVERVIEW OF THE STYLE PROCESS

1. Define the style attributes that match with the selectors.
2. Assign class selectors to widgets when necessary.
3. The widgets request the style and use the style attributes.

Coding: step 2

GOAL: CHANGE THE LOOK OF THE WATCHFACE

- `git checkout -f step/2`, then search for “STEP 2”.
- You will learn how to set style attributes to widgets.
- The method `populateStylesheet()` already defines the styles for the watchface.
- We can edit the style attributes to change the look of the watchface (colors, fonts, ...)

INSTRUCTIONS

Edit the existing styles to:

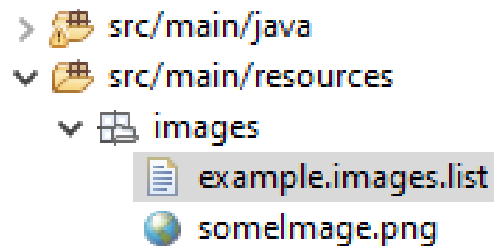
1. Go to the method `populateStylesheet()` of the class `WatchfacePage`.
It is responsible for defining the styles of the digital watchface.
 2. Change the color of the background of the watchface.
 3. Use the large font for the heart rate.
 4. Set the icon color of the heart rate to be blue (see [Colors](#)).
 5. Use the medium font for the seconds of the digital clock.
 6. Make the digital clock to be horizontally centered-aligned and vertically bottom-aligned.
 7. Set the text color of the steps and distance to be yellow by giving them a class and using a `ClassSelector` (see `addClassSelector()`).
 8. Set the battery indicator to be black.
- Note: you can go beyond these instructions and edit the styles as you want.



Loading Images

IMAGE DECLARATION

- To be displayed, the images have to be converted from their source format to a RAW format that is managed by the BSP display driver.
- The conversion is done at build-time or at runtime.
- Images have to be:
 - in the application classpath (typically in the `src/main/resources` folder).
 - declared in a `*.images.list` file.



Preview of the file example.images.list

```
/images/someImage.png:ARGB4444
```

DECLARATION SEMANTIC

`/images/someImage.png:ARGB4444`

Image path

Output format

- **Image path:** the absolute path to the image from the root of the *resources* source folder (starts with a /).
- **Output format:** specifies the format in which to embed the image in the application.
- In this example (ARGB4444), the image is converted to the output format ARGB4444 at build-time.
The application embeds a 16 bpp raw representation of the image (4 bits for **A**lpha, 4 bits for **R**ed, 4 bits for **G**reen, 4 bits for **B**lue).
- Main output formats: ARGB8888, ARGB4444, RGB888, RGB565, A8, A4, display
 - ARGBxxxx for multicolored images with transparency.
 - RGBxxx for multicolored images without transparency.
 - Ax for alpha images. Images can be colorized at runtime.
 - display: same encoding as the display.

GET THE IMAGE

- To get the image at runtime, use the method `Image.getImage()`

```
Image image = Image.getImage("/images/someImage.png");
```

- The image can now be used in a widget or as background.

- For example:

```
style.setBackground(new ImageBackground(Image.getImage("/images/someImage.png")));
```

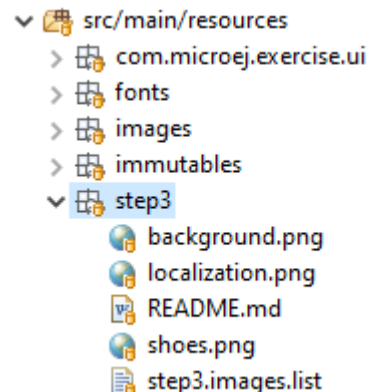
- More about images:

<https://docs.microej.com/en/latest/ApplicationDeveloperGuide/UI/MicroUI/images.html>

Coding: step 3

GOAL: A NEW DESIGN FOR THE WATCHFACE

- `git checkout -f step/3`, then search for “STEP 3”.
- The designer provided the resources and preview of the design: we have to change the code to match with the new design.
- You will use what you learned in step 1 (layout composition) and step 2 (style), to implement the given design.
- All the necessary resources for this step are in the folder `src/main/resources/step3`



NEW DESIGN

- The background is now a full-screen image.
 - The text color is now black.
 - The steps and distance icons have changed.
 - The steps and distance icon are now black.
 - The widgets positions have changed.
-
- We have to review the composition and style to adapt the code.

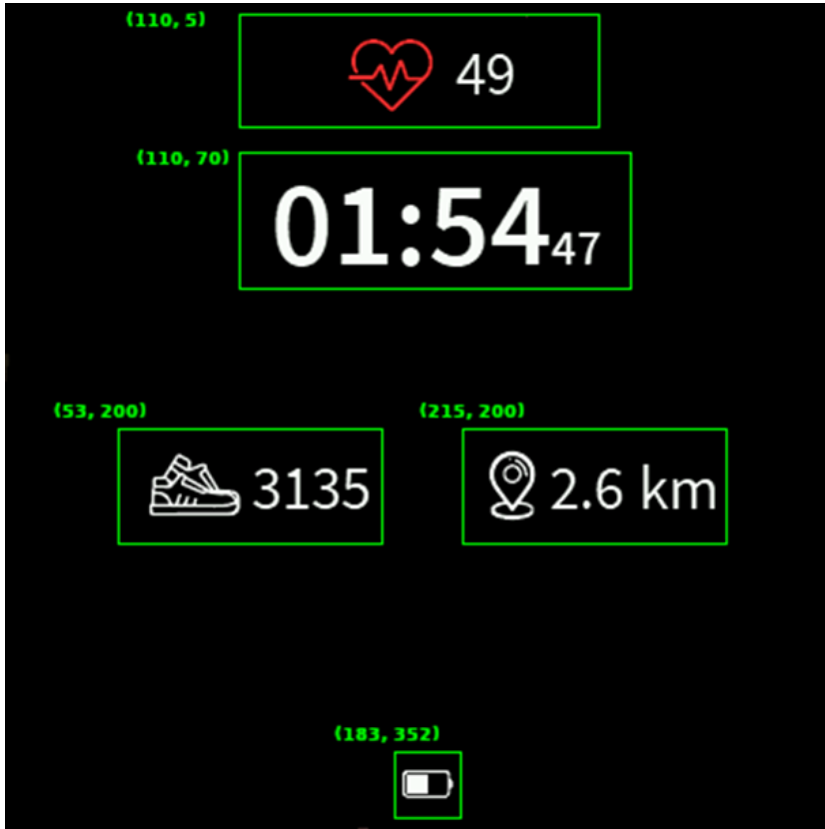


STEP 3.1 CHANGE THE LAYOUT OF THE WATCHFACE

We will use what we learned in step 1 to lay out the widgets like in the given design.

1. Go to the method `createDigital()` of the class `WatchfacePage`. It is responsible for creating the layout of the digital watchface.
2. Create a new `Canvas`. It is a container from the Widget library which is very flexible because it can lay out widgets at any given coordinates.
3. Add the 5 widgets (heart rate, steps, distance, digital clock and battery) to the canvas. The bounds to use are given in the next slide.
4. Make the method `createDigital()` return the canvas.

STEP 3.1 NEW WIDGET POSITIONS



In green, the widgets bounds

To speed-up the not-very-exciting process of finding the positions of the widgets within the canvas, we provide here the bounds to use when calling the method `Canvas.addChild()`:

- Heart rate: (110, 5, 170, 54)
- Digital clock: (110, 70, 185, 65)
- Steps: (53, 200, 125, 55)
- Distance: (215, 200, 287, 55)
- Battery: (183, 352, 213, 382)

- For example:

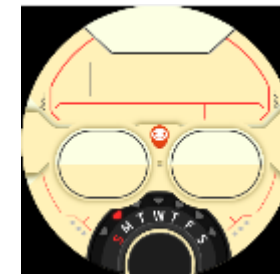
```
canvas.addChild(this.heartRate, 110, 5, 170, 54);
```

STEP 3.2 CHANGE THE STYLE OF THE WATCHFACE

We will use what we learned in step 2 to customize the style of the widgets like in the given design:

1. Go to the method `populateStylesheet()` of the class `WatchfacePage`. It is responsible for defining the styles of the digital watchface.
2. Change the existing styles accordingly to match with the design:
 - The text color of widgets is now black.
 - The background of the watchface is now an image (use [ImageBackground](#)).
3. In the class `Images`, change the path of the constants `SHOE_ICON` and `LOCALIZATION_ICON` to be the path to the new icons. These constants are used by the steps and distance widgets of the watchface.

- Note: the new images are in `src/main/resources/step3`



background.png



shoe.png



localization.png

Coding: step 4

GOAL: A NEW BACKGROUND FOR THE WATCHFACE

- `git checkout -f step/4`, then search for “STEP 4”.
- The designer provided an alternative background image for the design.
- You will learn how to load the image in the application and use it.

NEW DESIGN

- Only the background image has changed since last step.
- The image is `background.png` and is in the folder `src/main/resources/step4`

```
▼ src/main/resources
  > com.microej.exercise.ui
  > fonts
  > images
  > immutables
  > step3
  ▼ step4
    background.png
    README.md
    step4.images.list
```



INSTRUCTIONS

We have to edit the project configuration to declare the image in a `*.images.list`.

1. Edit the file `step4.images.list` in this folder.
 2. Add a line that declares the image (specify the path and output format).
 3. Go to the method `populateStylesheet()` of the class `WatchfacePage`. It is responsible for defining the styles of the digital watchface.
 4. Change the style of the digital watchface, to use the new image as background.
- Note: the output format depends on the image, its usage and the display capability:
 - Is it multicolored and opaque? Look for `RGBxxx`
 - Is it multicolored with transparency? Look for `ARGBxxxx`
 - Is it grayscale and colorized at runtime? Look for `Ax`

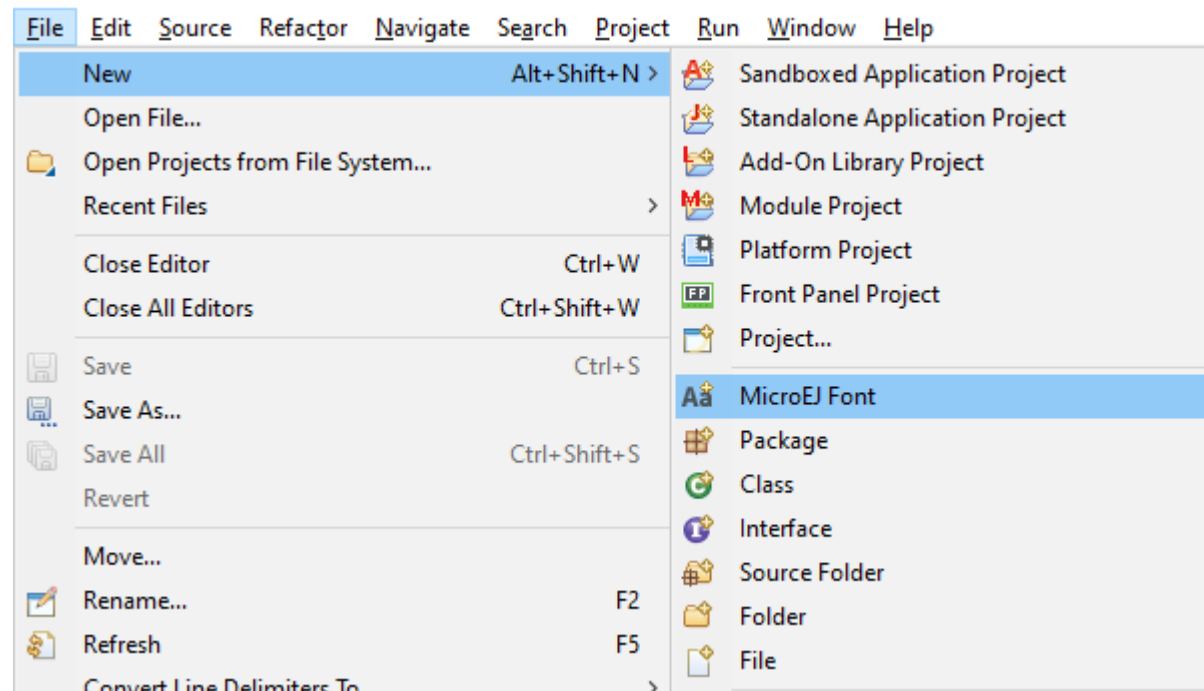
Creating and Loading Fonts

FONTs

- MicroEJ Fonts (EJF) are bitmap fonts (collection of the glyphs images).
- EJF fonts are created with a tool bundled with the SDK 5 called the Font Designer.
- The SDK 6 does not yet provide any substitution tool.
- **The SDK 5 is required for EJF font creation.** Please check the online documentation:
<https://docs.microej.com/en/latest/SDKUserGuide/installSDKDistributionLatest.html#sdk-installation-latest>

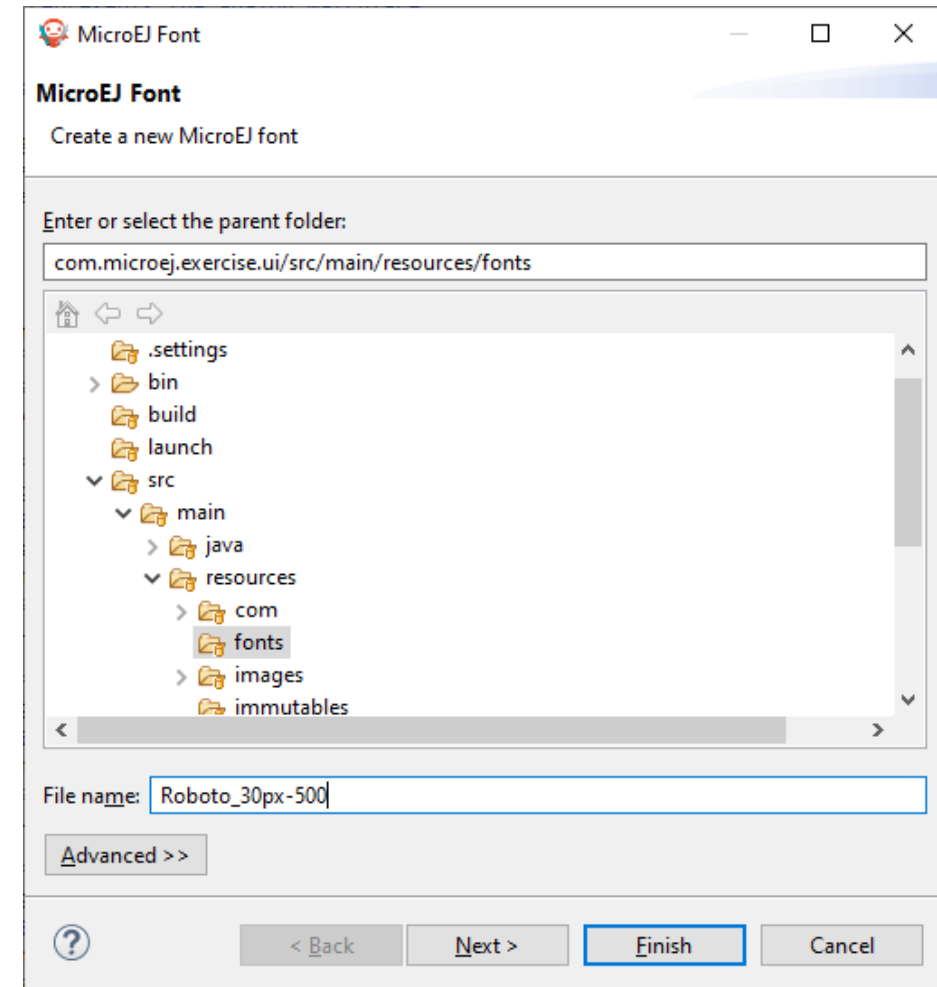
FONT CREATION (1)

- Start MicroEJ SDK 5 (see <https://docs.microej.com/en/latest/SDKUserGuide/startup.html>)
- Select *File > New > MicroEJ Font*



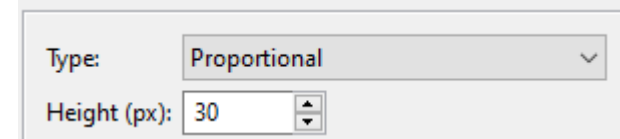
FONT CREATION (2)

- Select the *parent folder* and *file name*.
- Example:
 - We select the folder **fonts** in the **resources** source folder
 - We name it with a comprehensive name (family, height, weight)
- Click *Finish*.



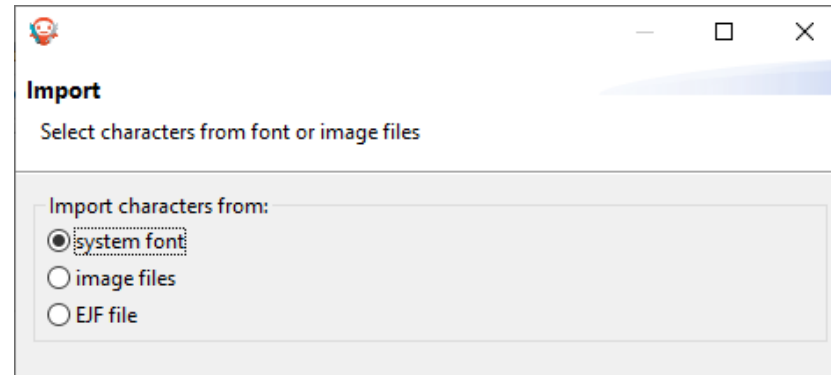
FONT CREATION (3)

- Set the font height to be the required size, in pixels. Example:
- The specified height is the height of the glyphs images in the font. It differs from the font size as expressed in the designer output or in a text editor for example.
- From a typography perspective, the height corresponds to the distance from max ascent to max descent lines. All the glyphs of the font will be fully enclosed between these two lines.
- The font size specifies the height of the text bounding box, but some glyphs of the font may extend beyond the box.



FONT CREATION (4)

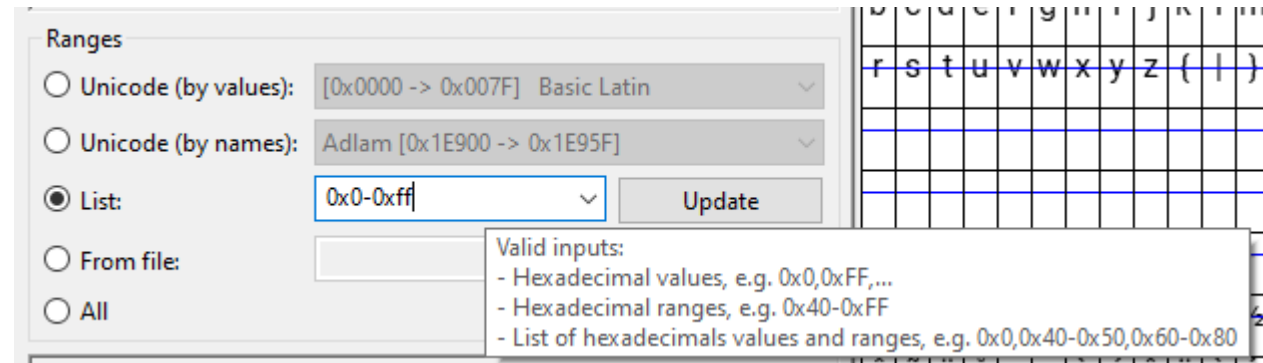
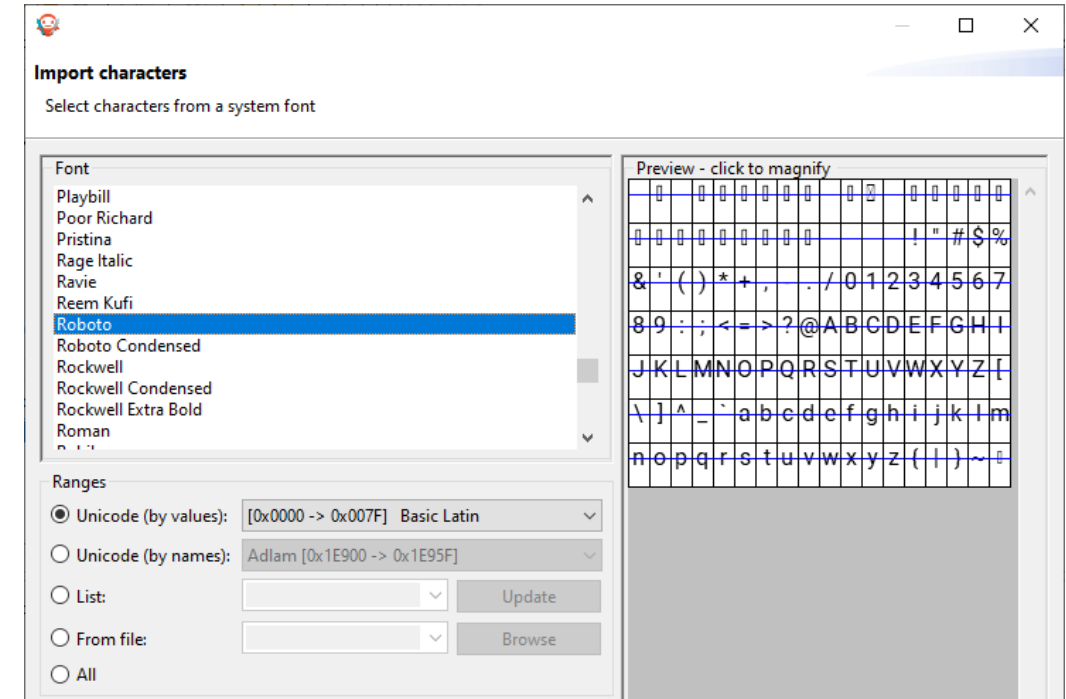
- Once you have determined the required height for the font, click on *Import...*
- Select *system font* to use a font installed on your system.



- Before starting the Font Designer, make sure that you have the font installed on your system (can be TTF, OTF, WOFF, ...)

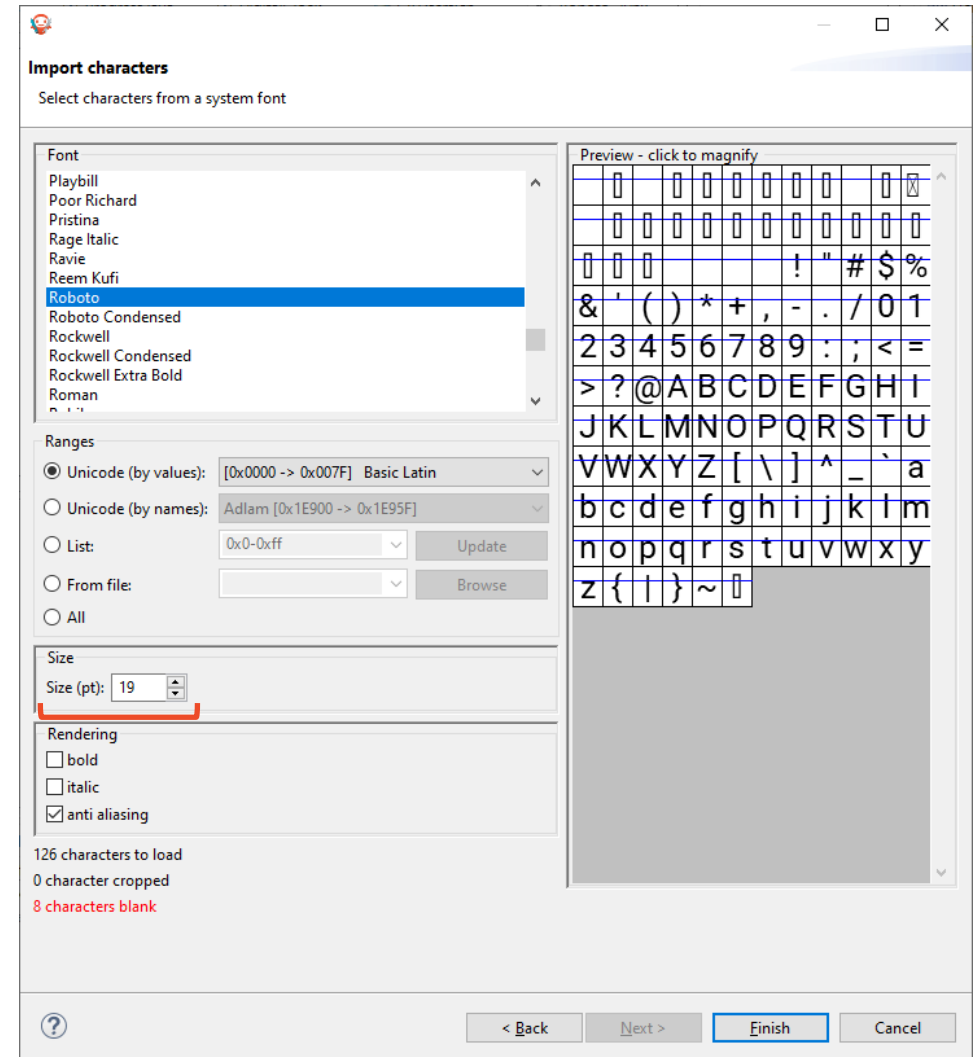
FONT CREATION (5)

- Select a font in the list of installed font.
- This automatically updates the preview on the right.
- Select the character range to import in the font.
- Multiple options:
 - From pre-defined Unicode ranges (e.g., Basic Latin)
 - From a custom list of ranges
 - All
- Example: import the custom range for Basic Latin and Latin-1 Supplement:



FONT CREATION (6)

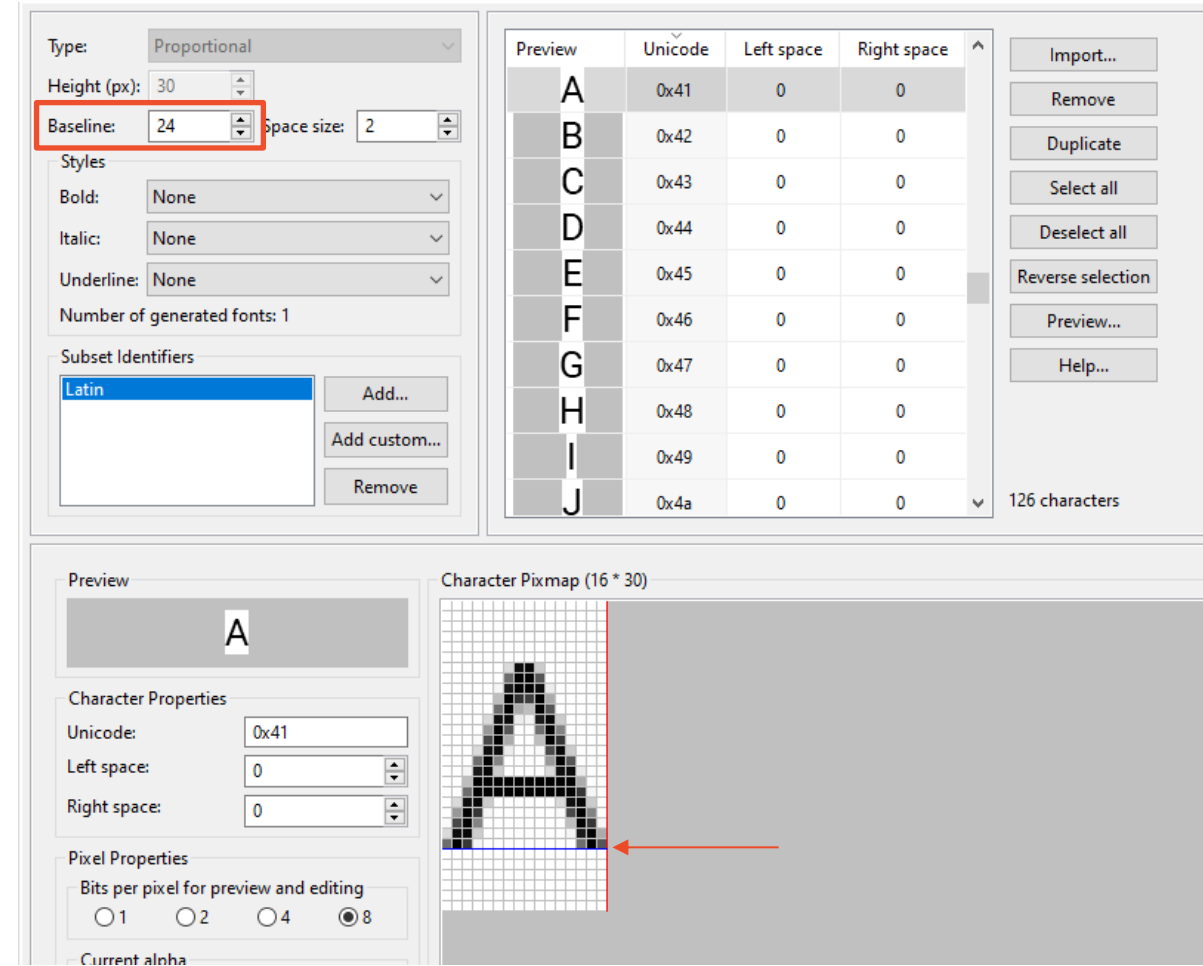
- Select the size that fits the best for the font.
- The best fit is generally when:
 - no character is cropped
 - characters look well-sized in the image box
- Click on *Finish*



FONT CREATION (7)

- Set the baseline, the line on which most characters sit.
1. Select a character that sits on the baseline, like the 'A'.
 2. Adjust the value of the field *Baseline* to align the blue line with the character, like in the picture.
 3. Save the font.
- The baseline information can be used in the code, to align two texts for example.
 - To get the baseline position for a given font:

```
int baseline = font.getBaselinePosition();
```



FONT CREATION (8)

- Set the space size.
1. Select a punctuation character with a width close to a space, like the ‘.’
 2. Adjust the value of the field *Space size* to match the width of the character, like in the picture.
 3. Save the font.
- A suitable space size is required for proper word spacing.

The screenshot shows the MICROEJ font creation interface. On the left, the 'Space size' field is highlighted with a red box and contains the value '7'. Below it, the 'Subset Identifiers' section shows 'Latin' selected. On the right, a table lists characters and their spacing properties.

Preview	Unicode	Left space	Right space
)	0x29	0	0
*	0x2a	0	0
+	0x2b	0	0
,	0x2c	0	0
-	0x2d	0	0
.	0x2e	0	0
/	0x2f	0	0
0	0x30	0	0
1	0x31	0	0
2	0x32	0	0

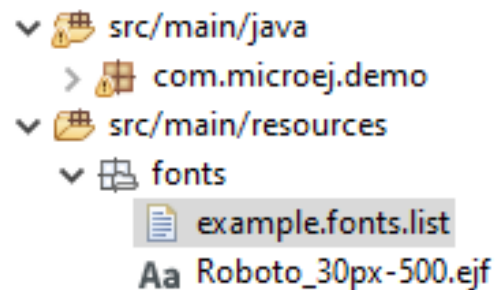
Below the table, the 'Character Pixmap (7 * 30)' section shows a grid with a red box highlighting the 'Character Properties' and 'Pixel Properties' sections. The 'Character Properties' section shows 'Unicode: 0x2e', 'Left space: 0', and 'Right space: 0'. The 'Pixel Properties' section shows 'Bits per pixel for preview and editing' with radio buttons for 1, 2, 4, and 8, where 8 is selected.

FONT CREATION (9)

- The Font Designer depends on the system native Font engine to render the glyphs images.
- The rendering is known to be different on Windows and Mac OS.
- We recommend to create fonts on a Mac for better-looking fonts.
- This task can typically be done by the designer.
- Side note: bitmap fonts support comes by default with MicroUI but we support vector fonts with our Foundation library for drawing vector graphics (MicroVG).
Please contact your MicroEJ sales representative or our support team for more information about Vector fonts support.

FONT DECLARATION

- EJB fonts are converted at build-time to a compatible format.
- Fonts have to be:
 - in the application classpath (typically in the `src/main/resources` folder).
 - declared in a `*.fonts.list` file.



Preview of example.fonts.list

```
/fonts/Roboto30px-500.ejf:0x21-0x7e:4
```

DECLARATION SEMANTIC

`/fonts/Roboto_30px-500.ejf:0x21-0x7e:4`

Font path Set Pixel depth

- **Font path:** the absolute path to the font from the root of the *resources* source folder (starts with a /).
 - **Set:** the set of characters to embed.
 - **Pixel depth:** specifies the pixel depth of the glyphs bitmaps for a font.
-
- In this example, the application embeds a 4 bpp raw representation of the font glyphs (anti-aliasing with 16 levels of opacity).
-
- Examples of valid sets:
 - `0x21-0x7e`: Basic Latin range from character ‘!’
 - `0x0-0x7e,0xc0-0xff`: Basic Latin + some accentuated characters from Latin-1 Supplement
 - `0x0-0x7e,0xb0`: Basic Latin + character ‘°’

GET THE FONT

- To get the font at runtime, use the method `Font.getFont()`

```
Font font = Font.getFont("/fonts/Roboto_30px-500.ejf");
```

- The font can now be used in a style.

- For example:

```
style.setFont(Font.getFont("/fonts/Roboto_30px-500.ejf"));
```

- More about fonts:

<https://docs.microej.com/en/latest/ApplicationDeveloperGuide/UI/MicroUI/fonts.html>

Coding: step 5

GOAL: A NEW FONT FOR THE WATCHFACE

- `git checkout -f step/5`, then search for “STEP 5”.
- You will learn how to create an EJF font and use it in the application.
- We want to change the font of the digital clock (the largest font).
- You can use any system font that you want for this step.

INSTRUCTIONS

1. Use the Font Designer to create a new EJB font. Set the height in pixels (e.g. 60px).
2. Declare the new EJB font in the file `step5.fonts.list` in the folder `src/main/resources/step5`.
3. Go to the method `populateStylesheet()` of the class `WatchfacePage`.
4. Get the font from its path with the `Font.getFont()` API.
5. Edit the style of the digital clock to replace its font with the new one.

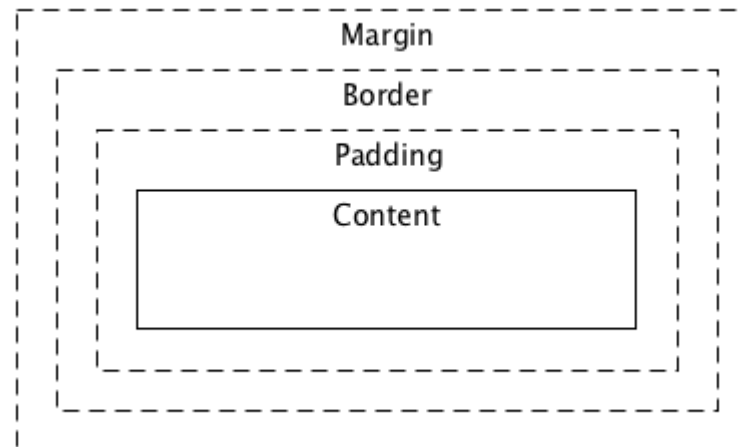
Widget Customization

CUSTOMIZATION

- Out-of-the-box widgets provide great features but you may need custom widgets for your specific needs.
- By creating new widgets or modifying existing ones, you can design a widget that displays the desired content.
- The next slides discuss the principles of widget rendering.

WIDGET RENDERING

- The method `Widget.renderContent()` is responsible for drawing the content of a widget.
- There are 3 main kinds of drawings:
 - Shapes (line, rectangle, circle, arc, ...)
 - Images
 - Texts
- This method only renders the content of the widget, without the border, padding and margin specified in the style.
- The border, padding and margin are applied before `renderContent()` is called.



THE RENDERCONTENT METHOD

- Typical implementation:
 1. Get the style of the widget
 2. Get the style attributes from the style (color, alignment, font, ...)
 3. Compute the positions of the elements within the widget content box
 4. Call MicroUI/Drawing APIs to draw the elements of the widgets

- Example:

@Override

```
protected void renderContent(GraphicsContext g, int contentWidth, int contentHeight) {  
    Style style = getStyle();  
    Font font = style.getFont();  
    // sets the color on the Graphics Context, then draw a text with the given font at (0,0)  
    g.setColor(style.getColor());  
    Painter.drawString(g, "Hello World", font, 0, 0);  
}
```

THE RENDERCONTENT METHOD

@Override

```
protected void renderContent(GraphicsContext g, int contentWidth, int contentHeight) {  
    Style style = getStyle();  
    Font font = style.getFont();  
    // sets the color of the Graphics Context, then draw a text with the given font at (0,0)  
    g.setColor(style.getColor());  
    Painter.drawString(g, "Hello World", font, 0, 0);  
}
```

- The GraphicsContext provides access to the modifiable pixel buffer. It can be used to read and write to the pixels of the display. Use the given graphics context for the drawings.
- The class **Painter** provides methods to draw basic shapes such as lines, rectangles, circles, text and images.
- The arguments **contentWidth** and **contentHeight** define the content bounds that are available for the widget content. They are used to size and position the elements within the widget.

THE COMPUTECONTENTOPTIMALSIZE METHOD

The optimal size is the minimal size that allow to show correctly the content of a widget. It is computed by `computeContentOptimalSize(Size)` method. This size is then used by the parent container to lay out (set position and size) the widget along with its siblings.

For instance, the optimal size of a label that displays a string will be the size of the string with the font defined in the style:

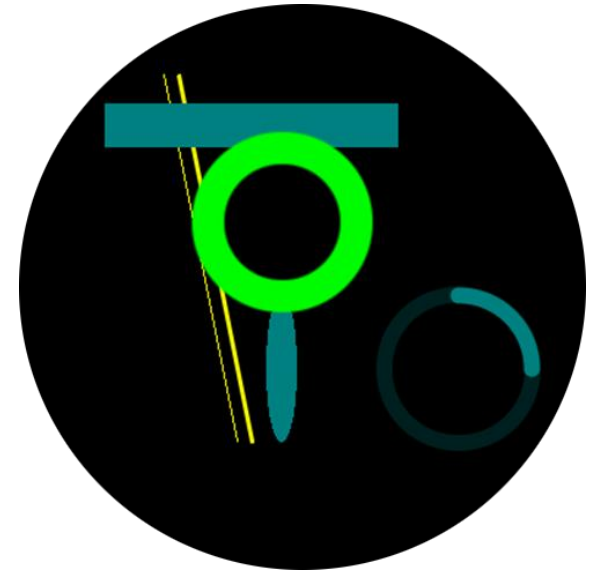
```
@Override
protected void computeContentOptimalSize(Size size) {
    Style style = getStyle();
    Font font = style.getFont();
    // retrieves the text width and height depending on the font
    int width = font.stringWidth("Some text");
    int height = font.getHeight();
    size.setSize(width, height);
}
```

DRAWINGS

- The APIs for drawing are provided by two Foundation libraries:
 - [MicroUI](#): provides the core features
 - Drawing: extends MicroUI to add more features
- The drawing methods are available in the painter classes (***Painter**):
 - [Painter](#) (MicroUI): drawing of aliased shapes, texts and images
 - [ShapePainter](#) (Drawing): drawing of anti-aliased shapes
 - [TransformPainter](#) (Drawing): drawing of transformed images and texts
- Click on the links above to have the full list of supported drawings.
- Note: The Widget library defines two more painter classes for manipulating texts and images:
 - [StringPainter](#) (Widget): utility methods for drawing texts
 - [ImagePainter](#) (Widget): utility methods for drawing images

DRAWING BASIC SHAPES (EXAMPLE)

```
g.setColor(Colors.YELLOW);  
Painter.drawLine(g, 100, 50, 150, 300);  
ShapePainter.drawThickFadedLine(g, 110, 50, 160, 300, 2, 1, Cap.PERPENDICULAR, Cap.ROUNDED);  
  
g.setColor(Colors.TEAL);  
Painter.fillRect(g, 60, 70, 200, 30);  
Painter.fillEllipse(g, 170, 200, 20, 100);  
  
g.setColor(Colors.LIME);  
ShapePainter.drawThickFadedCircle(g, 130, 100, 100, 20, 2);  
  
g.setColor(0x002020);  
ShapePainter.drawThickFadedCircle(g, 250, 200, 100, 10, 1);  
g.setColor(Colors.TEAL);  
ShapePainter.drawThickFadedCircleArc(g, 250, 200, 100, 0, 90, 10, 1, Cap.ROUNDED, Cap.ROUNDED);
```



DRAWING IMAGES (EXAMPLE)

```
Image image = Image.getImage("/images/heart.png");
```

```
g.setColor(Colors.RED);
```

```
Painter.drawImage(g, image, 100, 100);
```

```
g.setColor(Colors.GREEN);
```

```
TransformPainter.drawFlippedImage(g, image, 150, 100, Flip.FLIP_180);
```

```
g.setColor(Colors.MAGENTA);
```

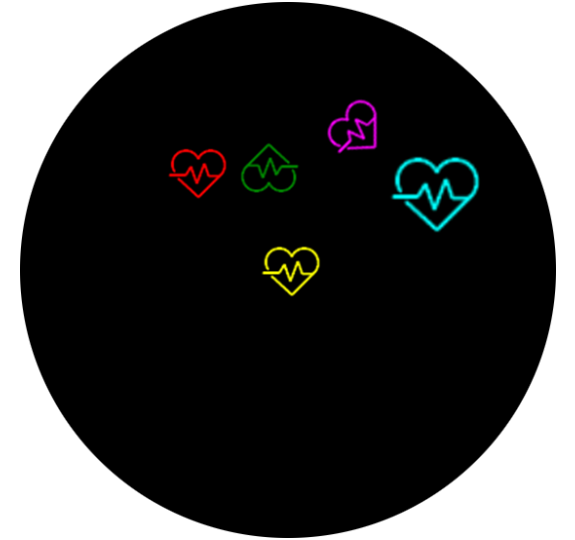
```
TransformPainter.drawRotatedImageBilinear(g, image, 200, 100, 200, 100, 45);
```

```
g.setColor(Colors.CYAN);
```

```
TransformPainter.drawScaledImageBilinear(g, image, 250, 100, 1.5f, 1.5f);
```

```
g.setColor(Colors.YELLOW);
```

```
ImagePainter.drawImageInArea(g, image, 0, 0, 390, 390, Alignment.HCENTER, Alignment.VCENTER);
```



DRAWING TEXT (EXAMPLE)

```
Font font = Font.getFont("/fonts/SourceSansPro_53px-600.ejf"); //$NON-NLS-1$
```

```
g.setColor(Colors.RED);
```

```
Painter.drawString(g, "Hello", font, 30, 80);
```

```
g.setColor(Colors.CYAN);
```

```
TransformPainter.drawScaledStringBilinear(g, "World", font, 70, 30, 1.5f, 1f);
```


```
g.setColor(Colors.LIME);
```

```
StringPainter.drawStringInArea(g, "!!!", font, 0, 0, 390, 390, Alignment.HCENTER, Alignment.VCENTER);
```



Coding: step 6

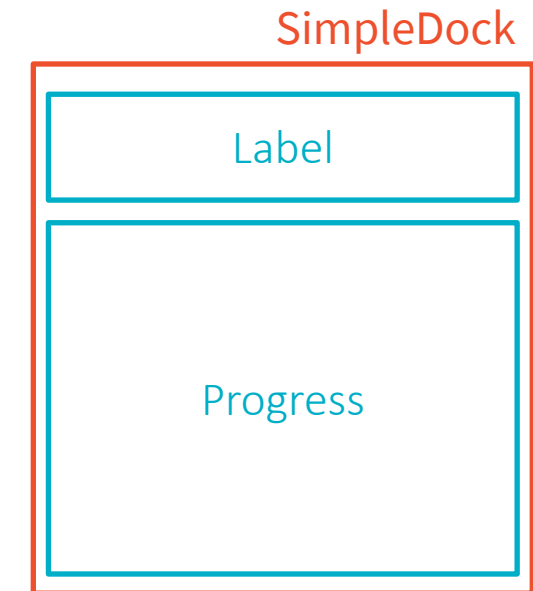
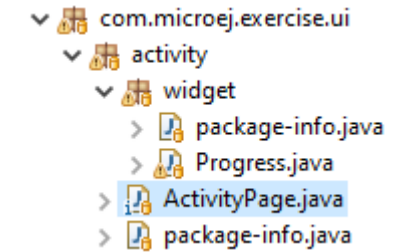
GOAL: DRAW A PROGRESS BAR

- `git checkout -f step/6`, then search for “STEP 6”.
- You will learn how to implement the rendering of widgets by using the **Painter** classes of MicroUI and Drawing.
- We will now work on the Activity entry of the application menu.
- To show Activity, push the button and select Activity in the list.
- Currently, Activity displays the step count.
- We want Activity to also display the progress towards the step goal, with a progress bar.
- The progress bar can be anything like: 



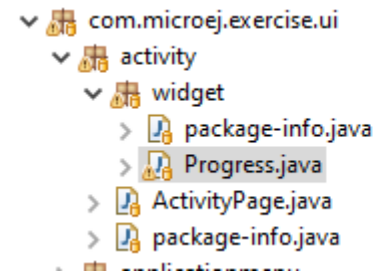
THE ACTIVITY PAGE

- First let's have a quick overview of the Activity page, `ActivityPage`.
- In a similar way to `WatchfacePage`, it defines:
 - a layout (`getWidget()`)
 - and styles (`populateStylesheet()`).
- The layout consists in a `SimpleDock` with:
 - a `Label` at the top for the title
 - a `Progress` at the center for the step count monitoring
- `Progress` is a custom widget, we will work on it to improve some of its aspects.



THE PROGRESS WIDGET

- The widget `Progress` displays a value.



```
20  
21 /**  
22  * A widget that shows a progress.  
23  *  
24  * <p>  
25  * The widget shows a value and the p  
26  */  
27 public class Progress extends Widget  
28
```

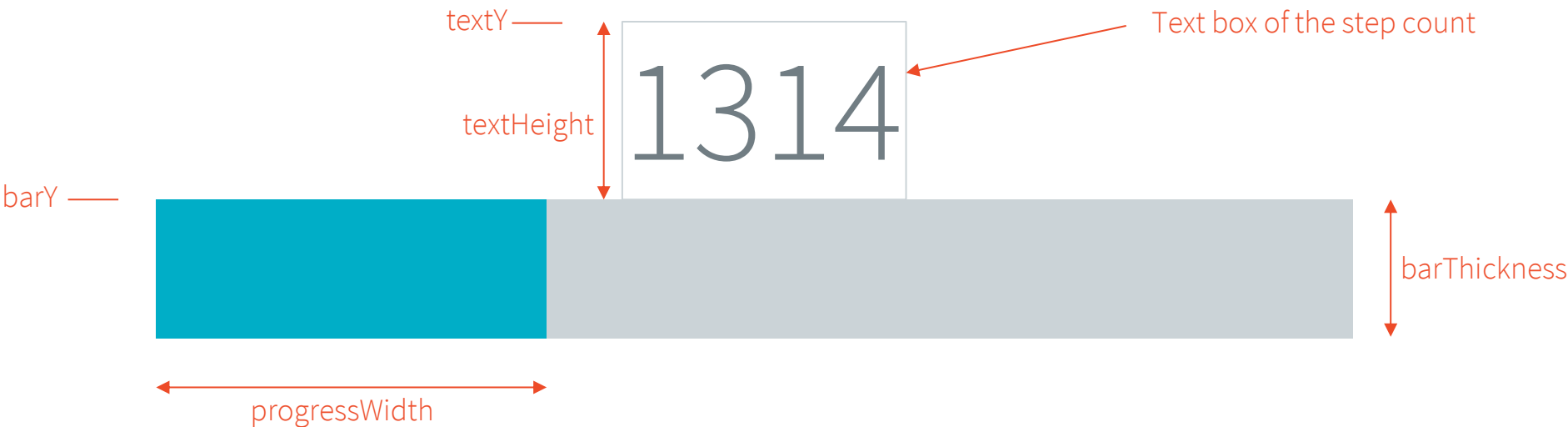
- In the method `renderContent()`, it gets the style and draws the value as a text (`Painter.drawString`).
- To draw the goal progress, we will now append the code that draws the shapes of the bar.
- The simpler approach is to draw an horizontal progress bar with rectangles or thick lines.



INSTRUCTIONS

1. Browse the API of class Painter and class ShapePainter and look for good candidates for drawing a progress bar.
 2. Get the progress value [0,1] with the method `getProgress()`.
 3. Get the style for the progress bar (colors, bar thickness).
 4. Draw the shapes that makes the progress bar.
- If you choose to draw an horizontal progress bar, see methods `Painter.fillRect()` and `ShapePainter.drawThickLine()`.
 - The next two slides are a cheat sheet to visualize the design of an horizontal bar.

CHEAT SHEET



- accentColor
- secondaryColor

CHEAT SHEET

- To save you some time finding the correct style attributes and progress variables, the code of `renderContent()` already provides some values:

```
int accentColor = style.getExtraInt(EXTRA_FIELD_BAR_ACCENT_COLOR, textColor);
int secondaryColor = style.getExtraInt(EXTRA_FIELD_BAR_SECONDARY_COLOR, 0xcccccc);
int barThickness = style.getExtraInt(EXTRA_FIELD_BAR_THICKNESS, DEFAULT_THICKNESS);
float progress = getProgress();
int progressWidth = (int) (progress * contentWidth);
int barY = textY + textHeight;
```

- `accentColor` defines the progress bar color.
- `secondaryColor` defines the color of the remaining progress bar.
- `barThickness` defines the thickness of the bar.
- `progress` is the progress value between 0 and 1.
- `progressWidth` is the width of the bar for the current progress. It is a ratio of the width of the widget.
- `barY` is the y coordinate of the bottom of the text. It can be used as the y for the bar.

Updating the content of a widget

REFLECT CONTENT UPDATE

- The widget content can change (e.g., a value changed)
- To reflect the change, call the method `requestRender()` to make the widget render again

- For example, imagine a `Label` widget that displays some text:

```
Label label = new Label("Some Text");
```

- Later in the execution of the application, the text changes:

```
label.setText("A new text");
```

- To update the content on the display, call `requestRender()` afterwards:

```
label.requestRender();
```

Coding: step 7

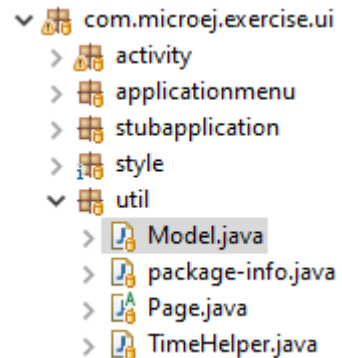
GOAL: UPDATE THE PROGRESS VALUE DYNAMICALLY



- `git checkout -f step/7`, then search for “STEP 7”.
- You will learn how to update the content of a widget and request it to render again, to reflect the changes on the display.
- The application manages a simple data model that simulates sensing data (HR, step count) and battery level.
- The model updates its data every 5 seconds and notifies the pages.
- Currently, the Activity page does nothing when the model updates: the displayed step count value remains the same.
- We will make the Activity page responsive to changes by updating the value of the **Progress** widget.

MODEL

- For the purposes of the demonstration, we implemented a simple data model in the class `Model`.



- Pages are notified when the model data changes: the model calls the method `update()` of the current page.
- The method `update()` of the `ActivityPage` is a good place for writing the update code.

INSTRUCTIONS

1. Locate the method `update()` of the class `ActivityPage`.
2. Set the new step count value to the widget `this.activityProgress` (use `Progress.setValue()`).
3. Request a new render of the activity progress widget to update the display.

Interacting with widgets

EVENTS

- MicroUI exposes the user inputs to the application layer:
 1. MicroUI processes the data from input-sensors
 2. MicroUI generates integer-based events which encode the sensor type and action
 3. The event is sent to the application's event handler
 4. The event handler processes the event
- MicroUI defines 4 generic types of events:
 - Pointer: actions on touch screens (press, move, release, etc.)
 - Buttons: actions on physical buttons (press, release, long press, etc.)
 - Command: actions with application-level logic (Start, Cancel, Pause, Back, Up, Down, etc.)
 - States: actions on stateful devices such as switches

TOUCH EVENTS

- Touch events are often used in modern GUI.
- The main events are:
 - Buttons.*PRESSED* : the user pressed the touch screen
 - Pointer.*DRAGGED* : the user moved its pointer/finger
 - Buttons.*RELEASED* : the user released its pointer/finger from the touch screen

THE HANDLE EVENT METHOD

- The method `Widget.handleEvent()` is responsible for handling the events received by a widget.
- The `handleEvent()` method returns a **boolean** that indicates whether the event has been consumed or not. Once an event is consumed, it is not dispatched to other widgets: the widget that consumed the event is the last one to receive it.
- The general guideline is to consume an event only if it triggers an action from the widget.

THE HANDLE EVENT METHOD

- Typical implementation:
 1. Get the type and action of the event.
 2. Update the widget state given the type and action.
 3. Return **true** to consume the event, **false** otherwise.

- Example:

```
@Override
public boolean handleEvent(int event) {
    int type = Event.getType(event);
    if (type == Pointer.EVENT_TYPE) {
        int action = Buttons.getAction(event);
        if (action == Buttons.RELEASED) {
            handleClick();
            return true;
        }
    }
    return super.handleEvent(event);
}
```

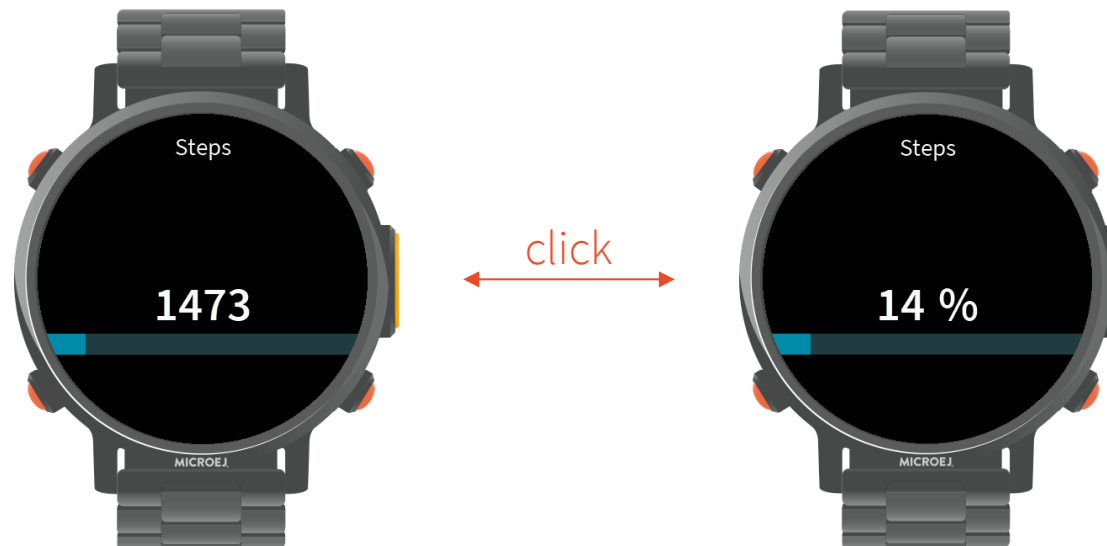
RECEIVING EVENTS

- Widgets have to be « enabled » to receive events: the default event dispatch policy forwards the incoming events to enabled widgets only.
- Widgets can be enabled:
 - By construction (constructor `Widget(boolean enabled)`)
 - On demand (setter `setEnabled(boolean enabled)`)

Coding: step 8

GOAL: HANDLE THE USER CLICKS

- `git checkout -f step/8`, then search for “STEP 8”.
- You will learn how to handle events, to interact with widgets using touch or physical buttons.
- Currently, the Activity page shows the step count and the progress towards the daily goal.
- We want to make it also display the progress in percent (e.g., 43 %)
- Clicking on the progress widget should toggle between the step count and the progress in percent.



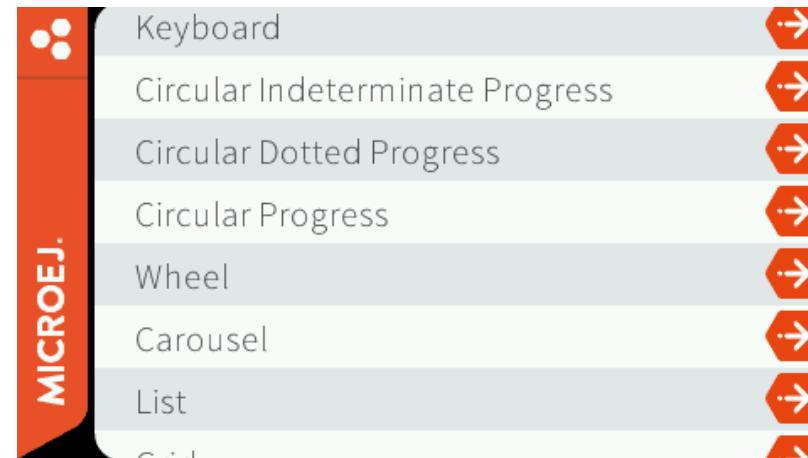
INSTRUCTIONS

1. Locate the method `handleEvent()` of the class `Progress`.
2. When the event `action` is `Buttons.RELEASED`, toggle the value of the boolean `showValue`.
 - When `showValue` is `true`, the widget shows the value (here the step count)
 - When `showValue` is `false`, the widget shows the progress in percent
3. Request a new render to update the display.
4. Return `true` to indicate that the event has been consumed by this widget.

Animation

ANIMATIONS

- Animations make a graphical interface engaging and visually appealing.
- The animation API allows for creating animations.
- Animation examples:



MOTION ANIMATION

- The class `MotionAnimation` defines an animation from:
 - An `Animator`
 - A `Motion`
 - A `MotionAnimationListener`
- The `Animator` is the instance that will execute the animation. It schedules the steps of the animation.
- The `Motion` describes the parameters of the animation:
 - The start and end value of the animation
 - The rate of change of the value over time (easing)
 - The duration of the animation
- The `MotionAnimationListener` is an instance that will be notified at each frame of the animation, it will typically update the widget with the current value of the animation.

ANIMATION EXAMPLE

- Imagine a `Label` that displays an integer value.
- We animate this value from 0 to 100 over 2000 ms, with a Quart easing out
- When notified, the `MotionAnimationListener` updates the value of the `Label`

```
int value = 0;
final Label label = new Label(String.valueOf(value));
final Animator animator = getDesktop().getAnimator();
Motion motion = new Motion(QuartEaseOutFunction.INSTANCE, 0, 100, 2000);
final MotionAnimation animation = new MotionAnimation(animator, motion, new MotionAnimationListener() {

    @Override
    public void tick(int value, boolean finished) {
        label.setText(String.valueOf(value));
        label.requestRender();
    }

});
animation.start();
```

EASING FUNCTIONS

- Easing functions specify how a value changes over time.
- Help controlling the motion of graphics to achieve the desired effect (acceleration, bounce, etc.).
- Easing functions available:
 - Linear
 - Ease in, ease out, ease in-out for quad, quart, quint, cubic, bounce, elastic, ...
 - Full list [here](#)
- Example of linear function: `Function function = LinearFunction.INSTANCE;`
- You can preview these easing functions at <https://easings.net/>

ANIMATOR

- The animator schedules the animation “as fast as possible”: the next frame of the animation is drawn as soon as the drawing of the previous frame is finished.
- An instance of the **Animator** is available for widgets from the root Desktop.
 - Within the widget code, call `getDesktop().getAnimator()`
 - Only one **Animator** can be running at any time.

Coding: step 9

GOAL: ANIMATE WHEN THE VALUE CHANGE

- `git checkout -f step/9`, then search for “STEP 9”.
- You will learn how to use the animation API to make the user interface fancier.
- Currently, the Activity page updates the step count every 5 seconds without any animation.
- We would like to make it a bit nicer by animating the value and progress bar when the value changes.



INSTRUCTIONS

1. Locate the method `setValue()` of the class `Progress`. A call to this method will trigger the animation.
2. Create a new `Motion`: specify the easing function, the start value, the stop value and the duration of the animation:
 - Easing function: any implementation of `Function` (for example `QuartEaseOutFunction`)
 - Start value: the current widget value (`this.value`)
 - End value: the target value (the argument `newValue`)
 - Duration: a value in milliseconds
3. Create a new `MotionAnimation` instance to define the animation:
 - Animator: use the field `this.animator`
 - Motion: use the created motion
 - Animation listener: use `this` (the widget implements `MotionAnimationListener`)
4. Call the method `startAnimation()` to start the motion animation.
5. Locate the method `tick()` and update the value when the motion value changes.
 - Update `this.value` with the motion value
 - Request a new render

Coding: step 10

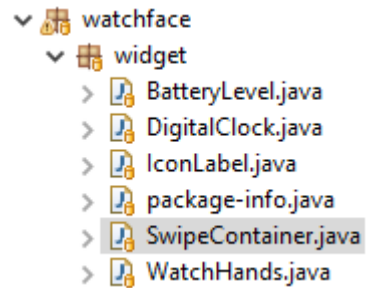
GOAL: ADD SWIPE NAVIGATION

- `git checkout -f step/10`, then search for “STEP 10”.
- You will learn how to add swipe transitions when navigating between views.
- The application provides only one watchface: the digital watchface.
- We would like to have two watchfaces: a digital and an analog.
- We will introduce the analog watchface and add swipe navigation between the watchfaces.



NEW MATERIAL

- We provide the code for the analog watchface and the swipe handling:
 - **WatchHands**: a widget that draws an analog clock, using images for the hands.
 - **SwipeContainer**: a widget that can swipe between its children widgets.



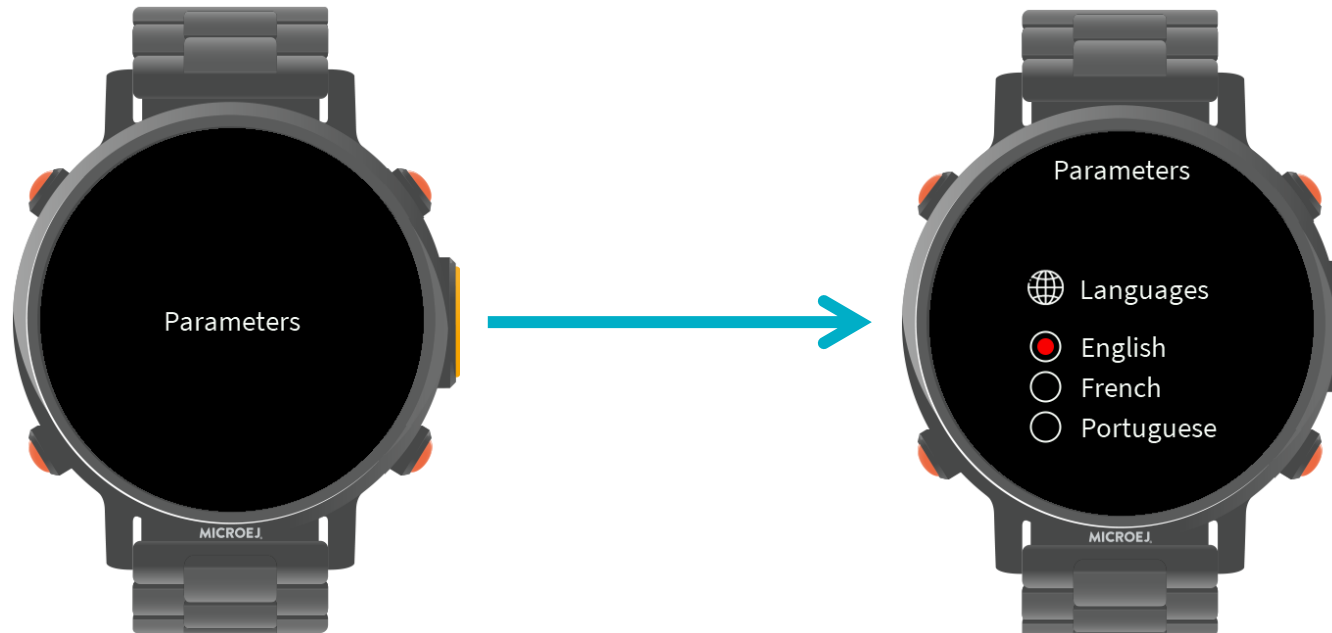
INSTRUCTIONS

1. Locate the method `getWidget()` of the class `WatchfacePage`. It defines the layout of the page.
2. Instantiate a new `SwipeContainer`.
3. Create the digital watchface widget with the method `createDigital()`.
4. Add it to the swipe container.
5. Create the analog watchface widget with the method `createAnalog()`.
6. Add it to the swipe container.
7. Return the swipe container.

Coding: step 11

GOAL: CREATE A NEW PAGE

- `git checkout -f step/11`, then search for “STEP 11”.
- In this step, you will learn how to add a new page to the application.
- The new page will use the `RadioButton` widget provided by the [Example-Java-Widget](#) demo.
- Currently, the Parameters page is a blank page. We want to implement it to match this design:



INSTRUCTIONS (1/2)

The page `ParametersPage` has been created but its methods need to be implemented.

1. Go to the Example-Java-Widget demo and look at the [RadioButton example](#). Add the classes `RadioButton` and `RadioButtonGroup` to your project. (Open the files on GitHub and click on “Download raw file” in the top right corner of the file. Then, move the files to the correct package in your project `com.microej.exercise.ui.parameters.widget`)
2. In the method `getWidget()` of the `ParametersPage`, create a `RadioButtonGroup` and three `RadioButton`. You can add them to a `List` or another `Container`. Look at the [RadioButtonPage](#) example in the Example-Java-Widget demo for the correct use of the widget.
3. Check the first `RadioButton`, using `group.setChecked(radioButton);`

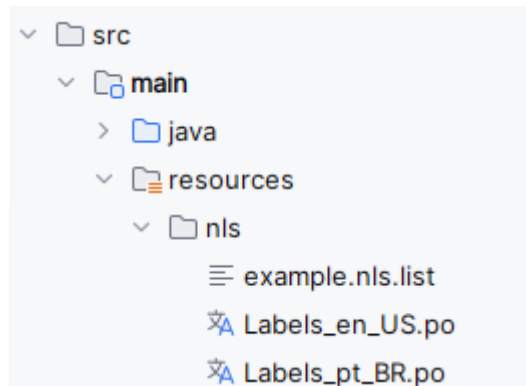
INSTRUCTIONS (2/2)

4. Create a new `SimpleDock` for the “Languages” subtitle. Create an `ImageWidget` and a `Label`, add them inside the new `SimpleDock`. The icon is declared in the `step11.images.list` resource file. Add the `SimpleDock` at the top of your `List`.
5. Add the `List` (or other `Container`) to the **root** `SimpleDock` as the `lastChild` and the page title should be the `centerChild`.
6. Now that the layout is complete, add style in the correct method (note that you may need to create new entries in the `ClassIdentifiers.java` class):
 - Apply style to the title `Label`. Refer to the `ActivityPage` if needed. Don’t forget to use `ClassIdentifiers`!
 - Add some margins to your `List` to center it: `style.setMargin(new FlexibleOutline(0, 0, 70, 100));`
 - Align the three `RadioButton` horizontally
 - Define the inner color of the `RadioButton` with the extra int field `RadioButton.CHECKED_COLOR_FIELD`
 - Align the title `Label` vertically at the top
 - Align the subtitle `SimpleDock` horizontally on the left and give it a 20px bottom padding
 - Add a right margin of 15px to the `ImageWidget`

Internationalization

INTERNATIONALIZATION

- Internationalization can be achieved with the Native Language Support (NLS) library.
- Localization source files can be internal (this tutorial uses PO files) or loaded as external resources.
- Internal PO files have to be:
 - in the application classpath (typically in the `src/main/resources` folder).
 - declared in a `*.nls.list` file.



Preview of example.nls.list

```
com.example.generated.Labels
```

This will retrieve all the translations from files named Labels*

Note: The project might need to be rebuilt in order to generate the sources

USAGE

- The `binary-nls` module must be added to the Application project build file `build.gradle.kts`:

```
implementation("com.microej.library.runtime:binary-nls:3.1.0")
```

- Manage the locales:

- Set the locale

```
Labels.NLS.setCurrentLocale("en_US");
```

- Check available locales

```
for (String locale : Labels.NLS.getAvailableLocales()) {  
    System.out.println(locale);  
}
```

- Retrieve the current locale

```
Labels.NLS.getCurrentLocale();
```

- Apply to a text:

```
Label label = new Label(Labels.NLS.getMessage(Labels.LabelId));
```

Coding: step 12

GOAL: ADD INTERNATIONALIZATION

- `git checkout -f step/12`, then search for “STEP 12”.
- You will learn how to implement internationalization by using the NLS library.
- Currently the application is only available in English. This step will show you how to change the `ParametersPage` locale dynamically.

NEW MATERIAL

In order to implement internationalization, some changes have been made:

- Import for the NLS Library has been added to the `build.gradle.kts` file.
- Localization source files have been added, they can be found in the `resources/step12` folder. Each file corresponds to a different locale.
- The fonts range in `resources/fonts/exercises.fonts.list` has been updated to accommodate accents in the new languages :
`/fonts/*.ejf:0x21-0x7a:4` → `/fonts/*.ejf:0x0-0x7e,0xc0-0xff:4`
- In the `RadioButton` class, a new attribute has been created to store the locale in the button.

ADAPTING NLS TO THE PROJECT ARCHITECTURE

We will use the existing `Model` class to handle the current locale changes. When a `RadioButton` is clicked, it will update the `Model`, which will then notify the `ParametersPage`.

The `Model` class uses the **NLS methods** to update or retrieve the current locale.

The `Model` is an `Observable`, and its `setCurrentLocale()` method has been configured to trigger an update of its observers (in our case, the `ParametersPage`) which allows the propagation of the locale change through the application.

INSTRUCTIONS (1/2)

1. Create the NLS list file `step12.nls.list` in the `resources/step12` folder. Provide the Fully Qualified Name of the Java interface that will be generated and used in the application (e.g. `com.mycompany.myapp.Labels`).
2. Build the project in order to generate the sources: open the Gradle pane and select *Tasks > build > clean* then *Tasks > build > build*.
3. In the `Model` class, update the `getCurrentLocale()` and `setCurrentLocale()` methods to use NLS.
4. Set the current locale in the `ParametersPage.getWidget()` method by calling the `Model` method you just updated.

INSTRUCTIONS (2/3)

5. Replace the `RadioButton` creations in the `ParametersPage.getWidget()` method to use `Labels.NLS.getAvailableLocale()`. This will make future changes easier if you decide to add/remove a language to the application.

```
// creates one RadioButton per available locale
for (String locale: Labels.NLS.getAvailableLocales()) {
    RadioButton radio = new RadioButton(locale, Labels.NLS.getDisplayName(locale), group);
    radio.addClassSelector(ClassIdentifiers.RADIO_BUTTON);
    if(locale.equals(Model.getInstance().getCurrentLocale())) {
        group.setChecked(radio);
    }
    list.addChild(radio);
}
```

This also checks the `RadioButton` corresponding to the current locale.

INSTRUCTIONS (2/2)

6. Update both `Labels` instantiation in the `ParametersPage.getWidget()` method to use NLS and allow translation:

```
myLabel = new Label(Labels.NLS.getMessage(Labels.msgid));
```
7. Create and override the `public void update()` method of the `ParametersPage`:
 - Set the text of the title and subtitle `Label` widgets using the corresponding `msgid` found in the `Labels*.po` files, for example:

```
this.title.setText(Labels.NLS.getMessage(Labels.Parameters));
```
 - Request a render of the `SimpleDock` at the **root** of the page to update it and its children.
8. In the `RadioButton.handleEvent()` method, set the current locale every time a button is clicked by calling the correct `Model` method.

Appendix

USEFUL DOCUMENTATION

- Application Developer Guide, UI section:
<https://docs.microej.com/en/latest/ApplicationDeveloperGuide/UI/ui.html>
- Demo Widget: shows the widgets and containers of the Widget library and many others in action:
<https://github.com/MicroEJ/Example-Java-Widget>
- MWT examples: <https://github.com/MicroEJ/ExampleJava-MWT>
- Getting Started MicroUI in <https://github.com/MicroEJ/How-To/>
- MicroEJ Libraries APIs: https://repository.microej.com/javadoc/microej_5.x/apis/index.html
- General MicroEJ documentation and resources:
 - <https://docs.microej.com/en/latest/>
 - <https://forum.microej.com/>

THANK YOU

for your attention !



MICROEJ[®]