COMP7510
Internet Computing and Programming

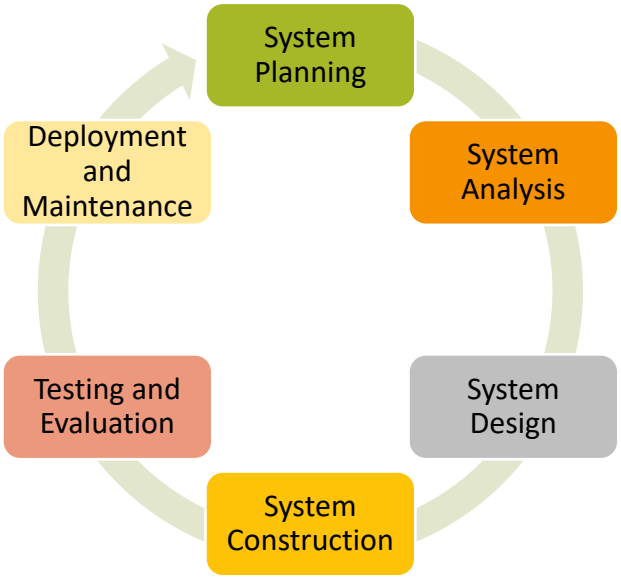
Lab Manual 1 – Introduction to Mobile Application Programming

Part 1 – Programming Methodology

Programming methodology (a.k.a. system development methodology) is designed to provide a consistent, repeatable process for developing systems. It defines the objectives and results for each stage. Systems Development Life Cycle (SDLC) is one of the system development methodologies that focuses on a large software project.

Systems Development Life Cycle

The SDLC includes all activities of the project. And, it focuses on realizing the requirements. The SDLC consists of different phases – system planning, system analysis, system design, system construction, testing & evaluation, and deployment & maintenance.



Each phase, except deployment & maintenance, may be repeated more than one time when some issues are found in its following phases. For example, the system design phase will be re-performed if the end users raise problems during the testing and evaluation phase.

System Planning

In this phase, the overall requirements for the system are defined, such as business needs, objectives, and so on.

System Analysis

In this phase, the large problem is broken down into discrete modules, diagrams, and other visual tools to analyze the situation or needs. The end users are engaged to define the definite requirements.

System Design

The system requirements are transformed into a design document. The functions and operations of the system are designed and described in the document.

System Construction

In this phase, the system design documents are transformed into program codes. The system manual and module testing are done by the developers.

Testing and Evaluation

This phase validates and confirms the developed system whether it meets all functional requirements defined in the system planning/requirements phase. A user acceptance tests are conducted by the development team.

Deployment and Maintenance

The developed system is moved to production. The end users and operational supports are all in place.

Programming

Mobile app systems (except the standalone apps) are networked computer systems. A networked computer system can be separated into different components including the frontend part including client computers and client-side programs, and the backend part including network devices, application servers, server-side programs, and database servers. The frontend part is the app installed and run inside a mobile device. The backend part is managed by the service provider. In our labs, we are going to learn how to develop the client-side program, iOS app, using some new techniques including Dart, Flutter, and Firebase.

Dart

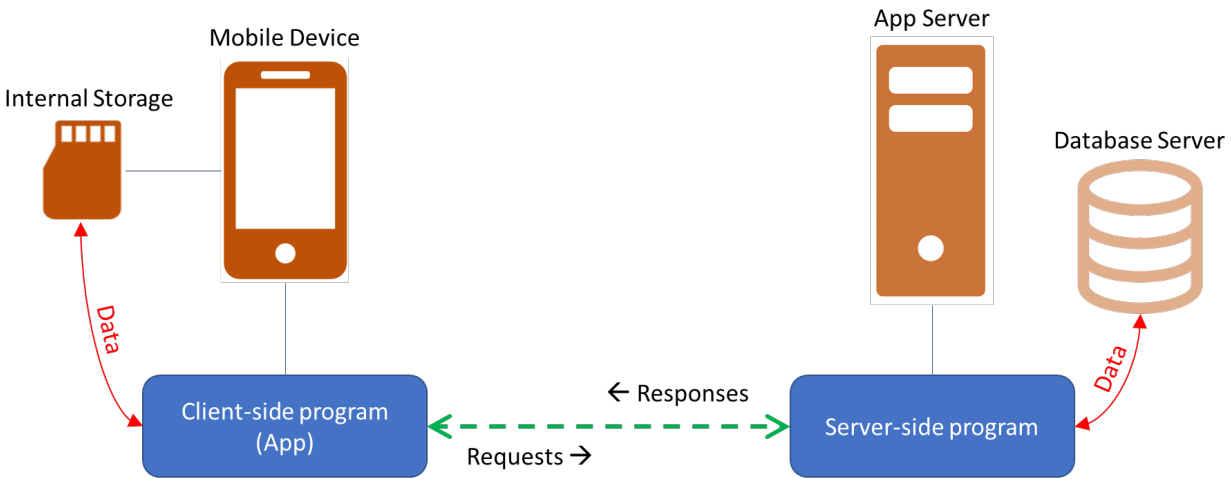
Dart is a client optimized, object-oriented programming language, originally developed by Google. It can be used to build a client-side program such as mobile apps, web apps, etc. Dart has thousands of libraries and packages for different purposes. You may browse its official website at https://www.dartlang.org/ for more details.

Flutter

Flutter is a framework and software development kit (SDK) introduced by Google for mobile app development. It provides a fast development feature – *hot reload,* which allows us to test the codes without reloading everything. Flutter has a rich set of widgets to build native interfaces that support both IOS and Android platforms. Our lab materials mainly focus on IOS platform. If you want to know more about Flutter, you may reach its official website at https://flutter.io/.

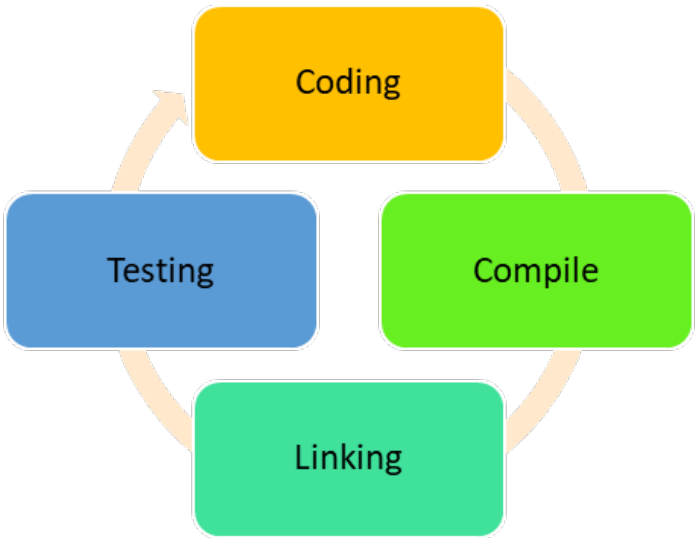
Firebase

Firebase is a service provided by Google that manages our backend data. With **Firebase**, we do not need to manage the backend systems including the server-side program, application server, database server, data backup system and network.



Constructing System

A large system is separated into smaller modules. During the construction, developers construct the system module-by-module. And, they perform program testing time-to-time. Producing a program includes the following steps – coding, compile, packages and objects linking, and testing.



Coding

Developers implement program codes for solving the problems defined in the system planning phase. The modern development tools aid in finding out the syntax error of the program codes during the code implementation.

Compile

The compile is a process of a program compiler to convert the program codes into machine codes. The compiler lists all errors if the compile faults caused syntax errors.

Linking

The linking is a process to link all compiled objects and required packages or libraries together and generates an executable program. Some modern development tools handle the objects and packages linking and output the executable program directly.

Testing

The testing process is used to find out the run-time error or logical error.

Part 2 – Preparation of App Development

In the coming labs (including this lab), we will walk through the procedures to develop a mobile app using *Dart* and *Flutter*. This mobile app is a client-side program that provides user interfaces for teachers and students exchanging information, such as the announcement by teachers, discussions among teachers and students, information sharing among student group members, and so on.

We are going to use the development tool – IntelliJ IDEA. If we use it for developing an app, we need to create a program project first. The product of the Flutter project can be run on both the iOS and Android platforms, but additional configurations are required for different platforms. In our labs, we focus on the iOS platform.

Development Environment for Our App

Since we mainly focus on the iOS platform, we require an Apple Macintosh. For the software, we need the following:

1. macOS (64-bit) 10.13.5 or later
2. Xcode 9.4.1 or later
3. IntelliJ IDEA Community 2018.1.6 or later
4. Dart 2.0
5. Flutter 0.5.1 beta or later

The software installation is done in the lab computers. If you want to install for your own computers, you may check the steps mentioned in **Appendix – Software Installation (macOS)**.

Creating Project in IntelliJ

The following procedures are used to create a new Flutter project:

1. Launch IntelliJ and click "Create New Project" on its Welcome window.

2. On the left-hand side of the New Project window, select "Flutter" and select the *Flutter SDK* path using the "…" button.
   Usually, the *Flutter SDK* is stored at */Users/XXXX/development/flutter* (where *XXXX* is the current username).

3. Then, click "Next" to continue.

4. Type the project information in as follows:

Field	Value
Project name	reach
Project location	~/Documents/reach
Description	COMP7510 project
Project Type	Application

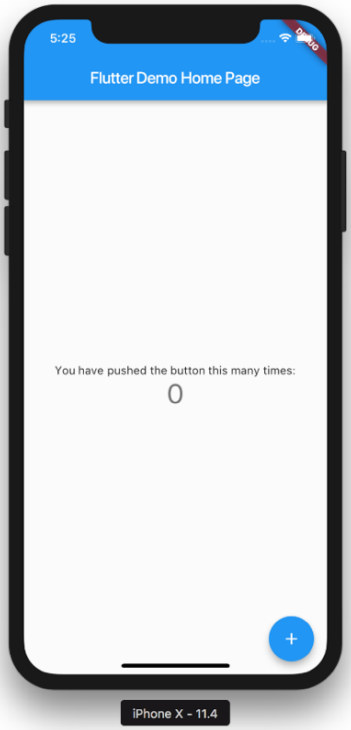
Field	Value
Organization	hk.edu.hkbu.comp
Android language	Java
iOS languages	Swift
Create project offline	☑

5. Then, click “Finish” to commit. *IntelliJ* will then prompt you for confirming the project directory creation. Click “OK” to confirm.

6. After a moment, the editor shows with the sample **main.dart** file.

7. Go to the top right corner of the editor and click the drop-down box showing “<no devices>” and select “Open iOS Simulator”.

8. Click the *Run* button on the top toolbar or menu “Run” → “Run ‘main.dart’” to run the sample program in the simulator.



Installing Dependencies

Then, we need to install the dependencies (required packages of external tools). Let’s follow the steps below:

1. Stop your app by clicking the *Stop* button on the top toolbar.

2. Open the **pubspec.yaml** file and add the following lines to the “dependencies” section.

```
image_picker: ^0.4.5              # for image selection
intl: ^0.15.6                     # for datetime conversion
fluttertoast: ^2.0.7              # for showing a toast message
```

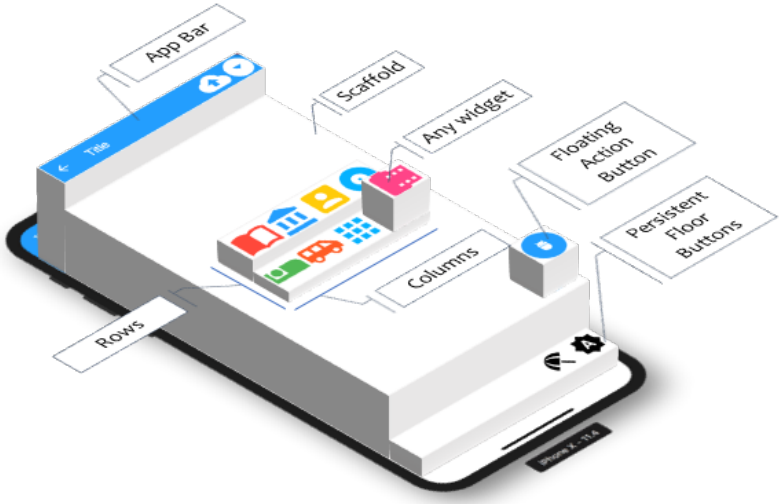
3. Click the “Packages get” link shown at the top right of the editor panel.

4. Click the **main.dart** tab, and click “Get Packages” if you see it at the top.

5. Open the **.packages** file to verify the package installation. Some installed packages may be newer than which you mentioned in the **pubspec.yaml** file, but it is fine.

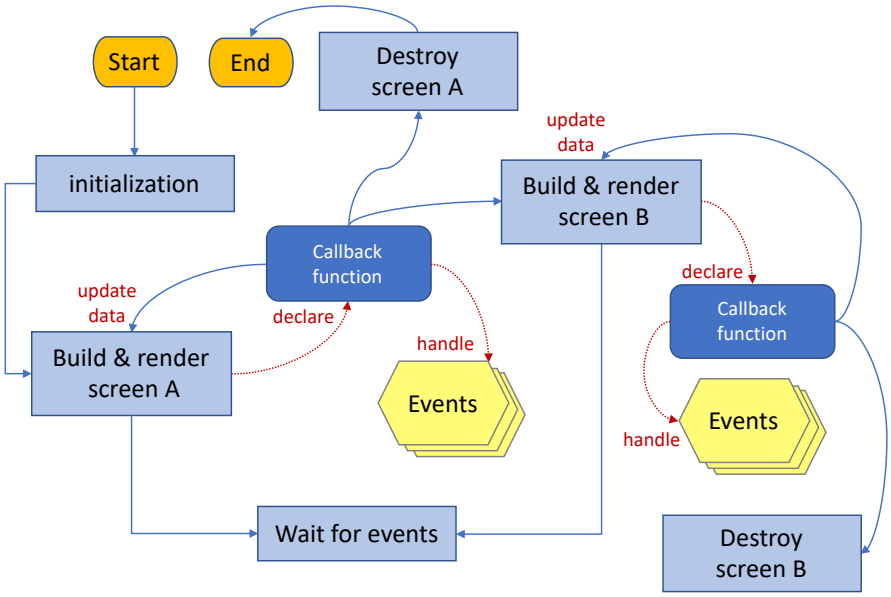
Part 3 – Coding for User Interfaces

Flutter is an SDK and framework for developing mobile apps. It provides a convenient way for building the user interfaces. In Flutter, all components on the user interfaces are widgets including the buttons, images, text items, etc. We use the widgets to construct the user interfaces – the screens you can view in the mobile app.



Flutter Mobile App Work Flow

In general, when we launch the app, it will first go into the initialization stage. In this stage, the app obtains necessary resources, such as establishing a connection to the server, downloading components, initializing subsystems, and so on. Then, the app goes into another stage for building and rendering screen. During the screen construction, some subroutines (a.k.a. callback function, event handlers) will be created for handling different events. After that, the screen will be rendered and the app waits for the event occurrence.



When an event occurs such as button pressed, data download completed, text content changed, etc., the corresponding callback function then handles the event – perform a specific action. It may update the data showing on the screen, so screen rebuild is required. Or, it may build and render a new screen. During the new screen construction, another set of the event callback functions will be created. Then, the original screen goes idle and the new screen takes place, and the app waits for the event occurrence. If the back button pressed event occurs, the current screen will be destroyed and the previous screen then takes place. If the current screen is the top level ready, the app will be terminated.

Functions

A function, called "method" in a class/object in the object-oriented programming, is a subroutine, a sequence of program instructions.

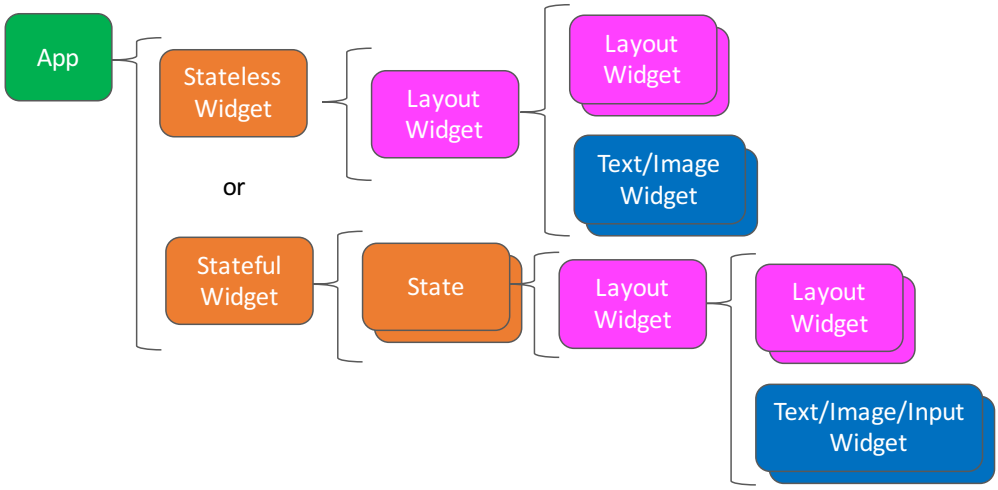
A function can be defined in a program locally as a library, or in a package that can be used in programs wherever that task should be performed. The main components of a function are the name, input parameters, function body and return value. A function must have a set of braces even if it has one statement only.

Callback Functions

Callback functions are functions that will be invoked when specific events occur. For example, the user presses a button, a corresponded callback function will be invoked for handling the event.

Layout Structures

Flutter has its own arrangement for the layouts of the user interfaces. The layout structure is as follows,



Stateless and Stateful Widgets

The **Stateless** and **Stateful** widgets are used to describe what and how widgets are showed in the user interfaces. The building process of the app continues recursively until the description of the user interface is fully concrete. The **Stateless** widget is an immutable widget. It is suitable for building a user interface with static contents. The **Stateful** widget is a mutable widget that is used when the parts of the user interface can be changed dynamically.

Layout Widgets

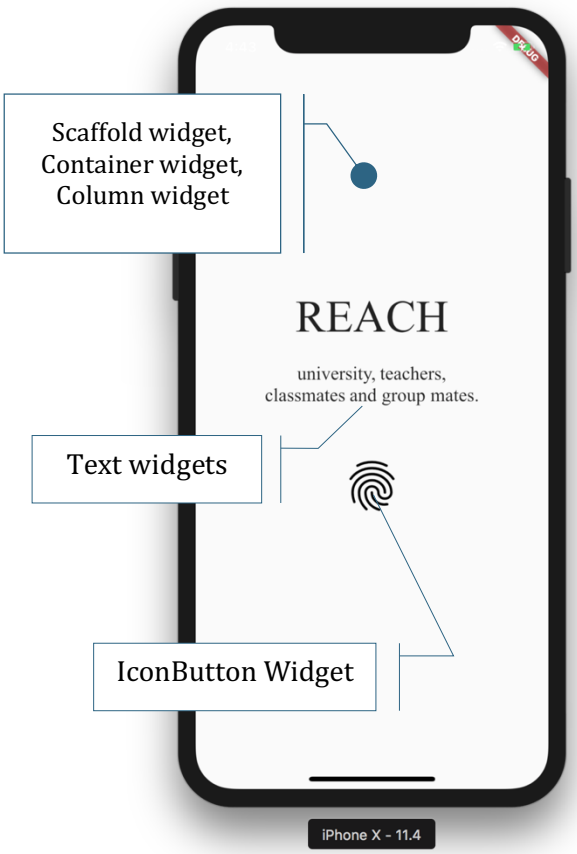
Layout widgets are the containers that contain other widgets with special position arrangements. Flutter has many layout widgets for different position arrangement needs.

Other Commonly Used Widgets

We, later in the labs, will discuss some commonly used widgets include text widgets, image widgets, and button widgets. Text widgets manipulate texts; Image widgets manipulate images or icons; Button widgets perform actions.

First User Interface – Splash Screen

We use the example codes to learn how to use different widgets. First, we are going to create our first user interface, the splash screen.



Widgets

The splash screen consists of different widgets including **Scaffold**, **Container**, **Column**, **Text** and **IconButton** widgets.

Scaffold

The **Scaffold** widget is a layout widget, a container that provides useful features including App Bar, Floating Action Button, and Persistent Footer. We will discuss them in later examples.

Container

The **Container** widget is also a layout widget, container. By default, the size of a widget is dynamic and auto-adjusted basing on its content or the sizes of child widgets. The **Container** widget has the **height** and **width** properties for fixing its size. These properties are very useful for positioning and sizing its child widgets.

Column

The **Column** widget is a container that lists its child widget vertically, row-by-row. All its child widgets are stored inside a list and the list is referred by its **children** property. With the **mainAxisAlignment** property, we can align the child widgets to the top, center, or bottom.

The **Row** widget is similar to the **Column** widget. But, it lists its child widgets horizontally, column-by-column.

Text & IconButton

The **Text** widget displays the text content with styles and colors. The **IconButton** widget is a button that triggers On-Pressed event if the user presses it.

Program Code

We now declare a new Stateful widget **HomePage** and a new State **HomeState** that will be used to create the user interface shown above.

Let's follow the steps below to start programming:

1. In IntelliJ, select the "lib" folder in Project Navigator. Then, click the menu "File" > "New" > "Dart File", and type **home.dart** in the pop-up window and press *ENTER*.

2. Open the **home.dart** file by double-clicking it in the **Project** panel, under the **lib** directory.

3. Try the following codes in the **home.dart** file:

```
1   import 'package:flutter/material.dart';
2   import 'package:fluttertoast/fluttertoast.dart';
3
4   class HomePage extends StatefulWidget {
5
6     @override
7     State createState() {
8         return HomeState();
9     }
10
11  }
12
13  class HomeState extends State {
14
15    @override
16    Widget build(BuildContext context) {
17        return splashScreen();
18    }
19
20  }
```

Constructor is a function too. It is used to create an object (instance) of the class. In our example, we do not need to implement the constructors for the classes. We only use the constructors for creating objects.

- The **import** statement is used for importing the program codes implemented outside the current code file (line 1 & 2).

- The **HomePage** class is declared (line 4 ~ 11), and it extends the **StatefulWidget** class.
    - Its **createState()** method (line 6 ~ 9) will be invoked automatically when the object (instance) of the Home page class is just created.

- The **HomeState** class is declared (line 13 ~ 20), and it extends the State class.
    - Its **build()** method (line 15 ~ 18) returns a base widget that contains other widgets for constructing the user interface.

    - In our codes, the **build()** method does not return a widget directly. Another method named **splashScreen** is invoked instead (line 17). Then, the **build()** method returns the returned widget of the **splashScreen()** method.

Class

Class is an important component in the object-oriented programming model. *Class* is the blueprint for creating objects, provides objects the properties (member variables) and member functions/methods. It can be used as a data type.

Properties (a.k.a, member variables) are used for storing values about the object of the class. More details about the variables will be discussed in the lab manuals later.

Member method is a function, subroutine that contains a sequence of program instructions for performing specific action.

Object is a system memory space allocated for storing data. An object is constructed based on its class, so it has properties and member methods as declared in the class.

4. Add the **splashScreen()** function to the HomeState class:

```
1  Widget splashScreen() {
2    var title = 'REACH';
3    var content = '\nuniversity, teachers, \nclassmates and group mates.\n\n';
4
5    return Scaffold(
6        appBar: null,
7        body: Container(
8          width: MediaQuery.of(context).size.width,
9
10         child: Column(
11           mainAxisAlignment: MainAxisAlignment.center,
12
13           children: <Widget>[
14
15             Text(title, style: TextStyle(fontSize: 48.0, fontFamily:
16               'Times New Roman'), textAlign: TextAlign.center,),
17
18             Text(content, style: TextStyle(fontSize: 20.0, fontFamily:
19               'Times New Roman'), textAlign: TextAlign.center,),
20
21             IconButton(
22               icon: Icon(Icons.fingerprint),
23               iconSize: 64.0,
24               onPressed: ()=>Fluttertoast.showToast(msg: 'Hello'),
25
26             ),
27           ],
28         ),
29       )
30   );
31 }
```

To make a beautiful user interface, we usually have containers nested inside others. Therefore, you will see that the program code is a bit long. But, most of the codes are for positioning or sizing the widgets only.

- The **splashScreen()** function has two local variables – title and content (line 2 & 3), and returns a Scaffold object (line 5).
- The body of the scaffold contains a **Container** widget. The **Container** widget contains a **Column** widget.
- The children property of the **Column** widget associates with a list. The list contains two **Text** widgets and an **IconButton** widget.
- The first **Text** widget shows the value of variable *title*; The second **Text** widget shows the value of variable *content*.

- The **onPressed** property of the **IconButton** widget is associated with a *callback* function. When the user presses the **IconButton** widget, the *callback* function will be invoked to show a *toast* message at the bottom of the screen.

Data Types

In a system, data are stored in different data types. For example, "Hello World" will be stored as **String** data type; *1*, *2* and *3* will be stored as **Integer** data type; *3.1415* will be stored as **Double** data type.

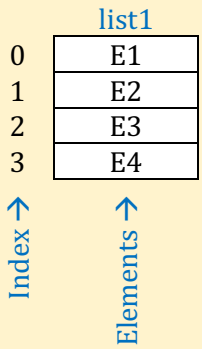
Dart has many data types for storing different kinds of data. Widget is one of the data types referring to the components of the user interface.

Variables

Variables are program elements that remember data values temporarily in the system memory. With the declaration statements, we can specify the name and data type of a variable that is used in the program. Then, a value can be stored for future use in the program.

The data type of a variable can be determined when we assign an initial value to the variable. Or, we can specify the data type when we declare a variable.

List



List is a structure for storing multiple values and these values can be referred using the same variable name.

Lists use zero-based indexing for referring the elements, where 0 is the index of the first element and *list.length - 1* is the index of the last element.

Syntax:

```
List<T> list = List<T>();
var list = <T>[ v1, v2, …, vN];
```

Examples:

```
List<String> strList = List<String>();
var widgetList = <Widget>[
  Text('Hello'),
  Text('World')
];
```

5. Open the **main.dart** file and change its content as follows.

```
1   import 'package:flutter/material.dart';
2   import 'home.dart';
3
4   void main() {
5     runApp(MyApp());
6   }
7
8   class MyApp extends StatelessWidget {
9
10    @override
11    Widget build(BuildContext context) {
12
13      return MaterialApp(
14        home: HomePage(),
15      );
16
17    }
18  }
```

- The import statements are used to import the program codes implemented outside the current code files.

- Because we declare the **HomePage** and **HomeState** classes in another dart file, we need to add the **import** statement in the **main.dart** file (line 2), so that they can be accessed by the codes declared in the **main.dart** file.

- The **main()** method calls a function named **runApp()** (line 5). The **runApp()** function needs a parameter that describes what widgets and how they are rendered in the user interface. The parameter **MyApp()**, the constructor of the **MyApp** class, returns an instance (object) of the **MyApp** class.

- The **MyApp** class is declared (line 8 ~ 17), and it extends the **StatelessWidget** class (OOP concept, inheritance), so it now is descendant of the **StatelessWidget** class.

    - Its **build()** method will be invoked automatically when the build process of the user interface starts. The **build()** returns a **MaterialApp** object. A **MaterialApp** object is an application object that uses material design.

    - The **home** property of **MaterialApp** describes the base widget – an instance of the **HomePage** class.

6. Press COMMAND+S to save the files, and re-run the app.

Second Screen – Main Menu

We are going to create a new user interface – the main menu, which is used for selecting different functions by the user.

Widgets

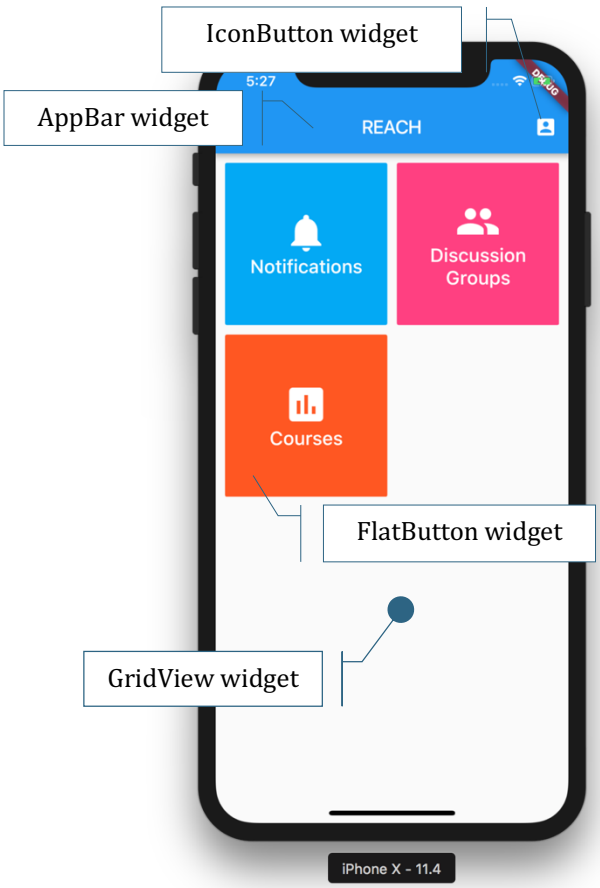
The main menu consists of buttons and an app bar. Two different types of button widgets are used – **IconButton** and **FlatButton** widgets.

FlatButton

The **FlatButton** widget is similar to **IconButton** but it allows a widget nested inside.

AppBar

The **AppBar** widget shows at the top of the screen. It allows us to add images, texts, and buttons on it.



Program Code

We declare the main menu in the **HomeState** class as well. We add the checking in the **build()** method to see which screen (splash screen / main menu) should be shown.

Let's follow the steps below to modify the **home.dart** file.

1. Add the following codes inside the HomeState class to create a new method named **menuScreen()**:

```
Widget menuScreen() {
  return Scaffold(
    appBar: AppBar(
      title: Text('REACH'),
      actions: <Widget>[
        IconButton(
          icon: Icon(Icons.account_box),
          onPressed: () => Fluttertoast.showToast(msg: 'Bye Bye'),
        ),
      ],
    ),
    body: GridView.count(
      crossAxisCount: 2,
      crossAxisSpacing: 10.0,
      mainAxisSpacing: 10.0,
      padding: EdgeInsets.all(10.0),
      children: <Widget>[

        FlatButton(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
              Icon(Icons.notifications, size: 48.0, color: Colors.white,),

              Text('Notifications',
                textAlign: TextAlign.center,
                style: TextStyle(fontSize: 20.0, color: Colors.white),
              ),

            ],
          ),
          color: Colors.lightBlue ,
          onPressed: () => Fluttertoast.showToast(msg: 'Notifications'),
        ),

      ],  // <Widget>[]
    ),    // GridView.count
  );      // Scaffold
}        // end of the menuScreen() method
```

- An **IconButton** widget is declared in the **actions** list of the **AppBar** widget (line 5 ~ 10). All widgets declared in the **actions** list will show on the right-hand side of the **AppBar** widget.

- A **GridView** widget is declared as a body of the scaffold by using the **GridView.count()** method (starting from line 12).

    - The **crossAxisCount** property is a required property (must be declared) that specify the number of columns in the view. The **crossAxisSpacing**, **mainAxisSpacing,** and **padding** properties are optional properties for specifying the column space, row space and padding space respectively.

    - All child widgets declared in the **children** list (starting from line 17) will be organized from left to right, from top to bottom.

2. Add a member variable **userID** to the **HomeState** class. It can be accessed anywhere inside the **HomeState** class.

```
1   class HomeState extends State {
2*    var userID = null;

…     …

    }
```

3. Modify the **build()** method of the **HomeState** class as follows:

```
1   @override
2   Widget build(BuildContext context) {
3*    if (userID != null)
4*      return menuScreen();
5*    else
6*      return splashScreen();
7   }
```

    - If **userID** is *null*, the **build()** method returns the returned widget of the **menuScreen()** method to the caller. Therefore, the main menu will be shown.

    - Otherwise, it returns the returned widget of the **splashScreen()** method. And, the splash screen will be shown.

4. Save all files and rerun the app. You should see the splash screen.

5. Change **userID** to a *non-null* value, say a string 'hello'. Save the file and rerun the app. You should see the main menu screen. Then, just keep the value of **userID** 'hello'.

if..then..else

By default, the program codes will be executed from the top to the bottom, statement-by-statement. For some situations, we need to change the flow, such as the user presses a button, image is just ready for display, etc.

The *if* statement is one of the program control statements. It bases on its expression to determine whether the code segment should be executed or not.

Syntax:

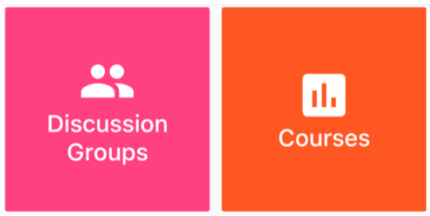
```
if (expression) {
  segment1
} else {
  segment2
}
```

If the result of the expression equals *true*, the segment1 will be executed.

Otherwise, the segment2 will be executed. The *else* block can be omitted if nothing will be done when the result of the expression equals *false*.

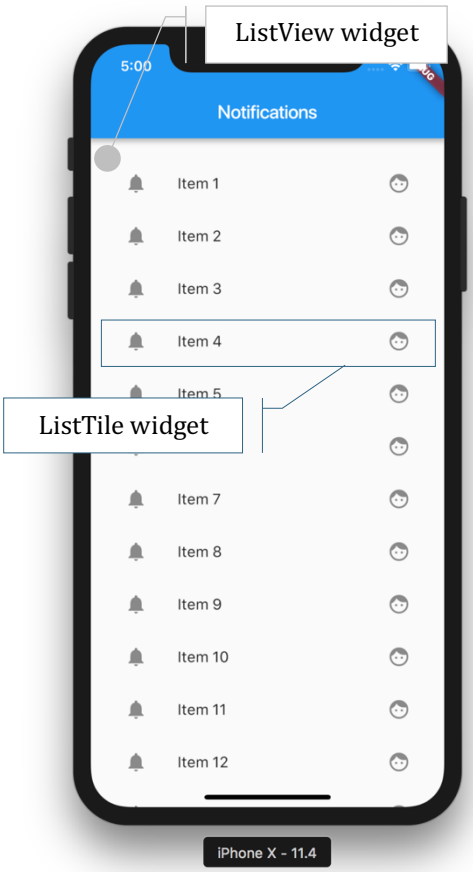
Little Challenge

Currently, your main menu page has the Notifications button only. Your task is to add two more buttons for Discussion Groups and Courses. Then, save your **home.dart** file.



Third Screen – Notification List Page

Of course, the app should have multiple screens for its different functions. Now, we create a new screen that has a list to show a list of notification items.



Widgets

The user interface contains three new widgets – **AppBar**, **ListView,** and **ListTile**.

AppBar

Again, the **AppBar** widget shows at the top of the screen.

ListView

The **ListView** widget is a container for listing items vertically with a scrolling feature.

ListTile

The **ListTile** widget is a container and the standard list item of a **ListView** widget.

Program Code

Let's create another dart file named **notification_list.dart**, and put the following codes inside it.

```
import 'package:flutter/material.dart';

class NotificationListPage extends StatefulWidget {
  @override createState() => NotificationListState();
}

class NotificationListState extends State {
  @override Widget build(BuildContext context) {

    var widgetList = <Widget>[];

    for (var i=1; i<=20; i++){
      widgetList.add(
          ListTile(
            leading: Icon(Icons.notifications),
            title: Text('Item $i'),
            trailing: Icon(Icons.face),
          )
      );
    }

    return Scaffold(
      appBar: AppBar(title: Text('Notifications'),),
      body: ListView(
        children: widgetList,
        padding: EdgeInsets.all(20.0),
      ),
    );
  }
}
```

- We create two more new classes – **NotificationListPage** and **NotificationListState**, just like **HomePage** and **HomeState**, we did previously.

- In the **build()** method of the **NotificationListState** class, we take an extra action (line 8 ~ 20) before returning the scaffold widget.

    - A widget list, **widgetList**, is created (line 10), which is used for storing list items that will be displayed inside the list view.

    - We use a *for* loop to add **ListTile** objects to the list (line 12 ~ 20).

- Finally, we return a **Scaffold** object (line 22 ~ 28), which contains a **ListView** object. This list view contains the widget list we recently created.

More about List

The **add()** method is used to add a new element to the end of the list. And, the number of the elements (*list.length*) will be increased by one.

```
list.add('Hello');
var element1 = list[0];
list[0] = 'Good morning';
print(list[0]);
```

Finally, **list[0]** stores 'Good morning' and the length of the list is *1.*

for loop

The *for* statement is a program control statement. It bases on its expressions to determine how many times the code segment should be executed.

```
for (init_expression; terminate_expression; increment_expression) {
  segment
}
```

- The init_expression is used to initialize local variable(s).
- The terminate_expression is used to determine whether the loop should continue or not.
- The increment_expression is used to increase the local variable(s).

```
for (var i = 0; i < 10; i = i + 1){
    print(i);
}
```

In the example codes above, the loop runs ten times.

1. In the first time, variable i equals 0 because the init_expression assigns 0 to variable i.
2. The second time, variable i equals 1 (i = 0 + 1).
3. The last time, variable i equals 9.

```
for (var i = 1; i <= 21; i = i + 2){
    print(i);
}
```

In the example codes above, the loop runs eleven times.

1. In the first time, variable i equals 1 because the init_expression assigns 1 to variable i.
2. The second time, variable i equals 3 (i = 1 + 2).
3. The last time, variable i equals 21.

In addition, we need to update the **main.dart** file:

1. Add an import statement at the top to import the **notification_list.dart** file.
```
import 'notification_list.dart';
```

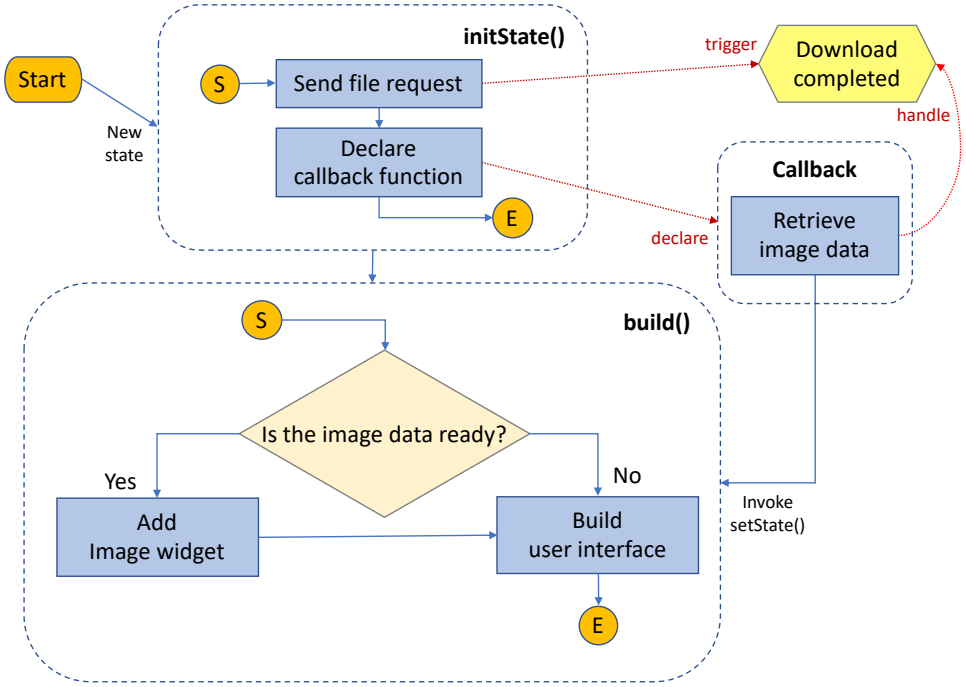
2. Change the **home** property of **MaterialApp** to **NotificationListPage()** as follows:

```
    return MaterialApp(
//      home: HomePage()
      home: NotificationListPage()
    );
```

3. Save all files and re-run the app. Then, you should see the new screen with a list.

Fourth Screen – Notification View Page

Next, we create a new screen to show the details of a notification. The notification view page includes an Image widget that shows an image. Before we create the notification view page, let's see the flow of downloading and showing the image in the page.



1. In the initial state, we send a request for downloading an image.

    a. The **http.get()** function is used to download a file (any file format) through the specific URL. It is an asynchronized function, the app will not wait for the completion of the file downloading.

    b. The **callback** function handles the download data once the downloading is completed.

2. After finishing the **initStat()** method, the **build()** method will then be invoked. We check whether the image data is ready or not. If it is ready, we add an Image widget with the image data. Otherwise, we just build the user interface without the Image widget.

3. When the image downloading completes, the **callback** function will be invoked. It retrieves the image data from the response of the server and calls the **setState()** method. Then, the screen will be rebuilt by invoking the **build()** method.

Widgets

In the sample output screen, you can see that there are two new things added – image and footer.

Image

The Image widget is used for showing an image.

Scaffold – persistenFooterButtons

The **persistentFooterButtons** property of the **Scaffold** widget is used to show different items in the footer.

Program Code

1. Now, we create a new dart file named **notification_view.dart**. Then, put the following codes in the file.

```
1   import 'package:flutter/material.dart';
2   import 'package:http/http.dart' as http;
3   import 'dart:async';
4
5   class NotificationViewPage extends StatefulWidget {
6     @override createState() => NotificationViewState();
7   }
8
9   class NotificationViewState extends State {
10
11    var imgBytes = null;
12
13    @override
14    Widget build(BuildContext context) {
15      var childWidgets = <Widget>[];
16
17      childWidgets.add(Text('Here is the content of the notification...'));
18
19      if (imgBytes != null)
20        childWidgets.add(Image.memory(imgBytes,
21          width: MediaQuery.of(context).size.width - 120)
22        );
23
24      return Scaffold(
25        appBar: AppBar(
26          title: Text(' Notification 1'),
27        ),
28        body: ListView(
29          padding: EdgeInsets.all(20.0),
30          children: <Widget>[
31            Column(
32              children: childWidgets,
33            ),
34          ],
35        ),
36        persistentFooterButtons: <Widget>[Text('Created by ...'),],
37      );
38    }
39  }
```

- At the beginning of the **build()** method of the **NotificationViewState** class, we declare a widget list (line 15) which will be used to store a **Text** widget and an **Image** widget.

- We check whether the image data is ready in the member variable **imgBytes** or not (line 19 ~ 22). If it is ready, an Image widget will be added to the widget list.

- The **persistentFooterButtons** property of the **Scaffold** widget is used to build a footer (line 36). It refers to a widget list. All widgets declared in the list will be displayed in the footer.

2. Add the **initState()** method to the NotificationViewState class:

```
1   @override void initState() {
2       super.initState();
3
4       var url = 'http://www.comp.hkbu.edu.hk/~mandel/comp7510/comp7510.jpg';
5       http.get(url).then((response) {
6         print('download complete!');
7         setState(() => imgBytes = response.bodyBytes);
8       });
9     }
```

- The **initState()** method downloads an image from the Internet using the **http.get()** method with the specified URL (line 4 ~ 8).

- Once the image downloading completes, the callback function will be invoked. The **setState()** statement updates the value of **imgBytes** and initiates a new state.

✪ *The **super.initState()** statement must be put as a first line in the **initState()** method to invoke the **initState()** method of the superclass.*

3. Open the main.dart file, and add an import statement to import the **notification_view.dart** file at the top of the **main.dart** file.

```
import 'notification_view.dart';
```

4. Change the **home** property of **MaterialApp** to **NotificationViewPage()** as follows:

```
    return MaterialApp(
//      home: HomePage()
//      home: NotificationListPage()
      home: NotificationViewPage()
    );
```

5. Save all files and re-run the app. The notification view page screen will be displayed.

Working Together

Normally, in a mobile app, when a new screen is built, it will put over the previous screen. The top screen will be destroyed, and the previous screen will be shown if the user presses the BACK button. So, the screens will be stacked one-by-one.

Now, we modify our dart files to make them work together.

1. Open the **main.dart** file, and modify the build() method of the MyApp class as follows:

```
1   @override
2   Widget build(BuildContext context) {
3     return MaterialApp(
4       home: HomePage(),
5       routes: <String, WidgetBuilder>{
6         '/notificationList':
7             (BuildContext context) => NotificationListPage(),
8         '/notificationView':
9             (BuildContext context) => NotificationViewPage(),
10      },
11    );
12  }
```

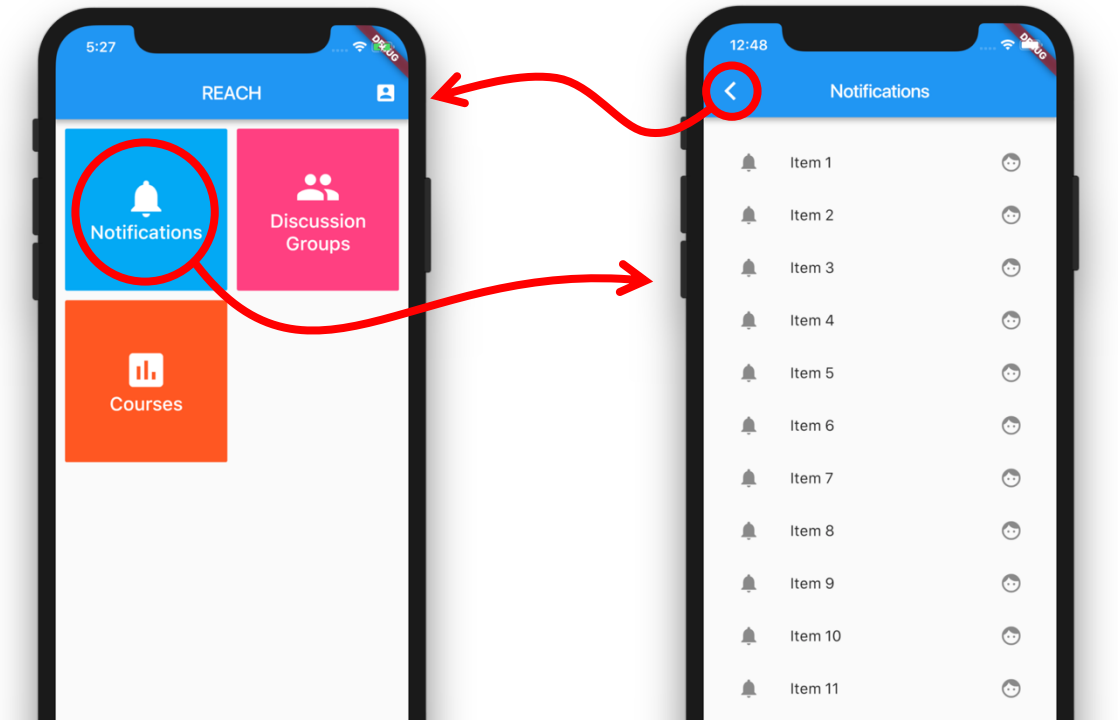
- We set the **routes** property as a map that associates to the functions for building different screens using the classes we declared in different dart files (line 5 ~ 10). A navigator of the material application then manages the user interfaces based on the route table – the assigned map.

2. Open the **home.dart** file, and locate the **menuScreen()** method. Assign a new callback function to the **onPressed** property of the first **FlatButton** widget as follows:

```
Widget menuScreen() {
  return Scaffold(
    appBar: AppBar(

      // lines are omitted…

    ),
    body: GridView.count(

      // lines are omitted…

      children: <Widget>[

        FlatButton(
          child: Column(

        // lines are omitted…

          ),
          color: Colors.lightBlue,
          onPressed:
           () => Navigator.pushNamed(context, '/notificationList'),
        ),

        // lines are omitted…

      ],
    ),
  );
}
```

- The callback function associated with the **onPressed** property creates a new **NotificationList** object and puts it to the navigator. When the user presses the *Notifications* button, a new notification list then shows up.

3. Save and re-run your app to see the result.



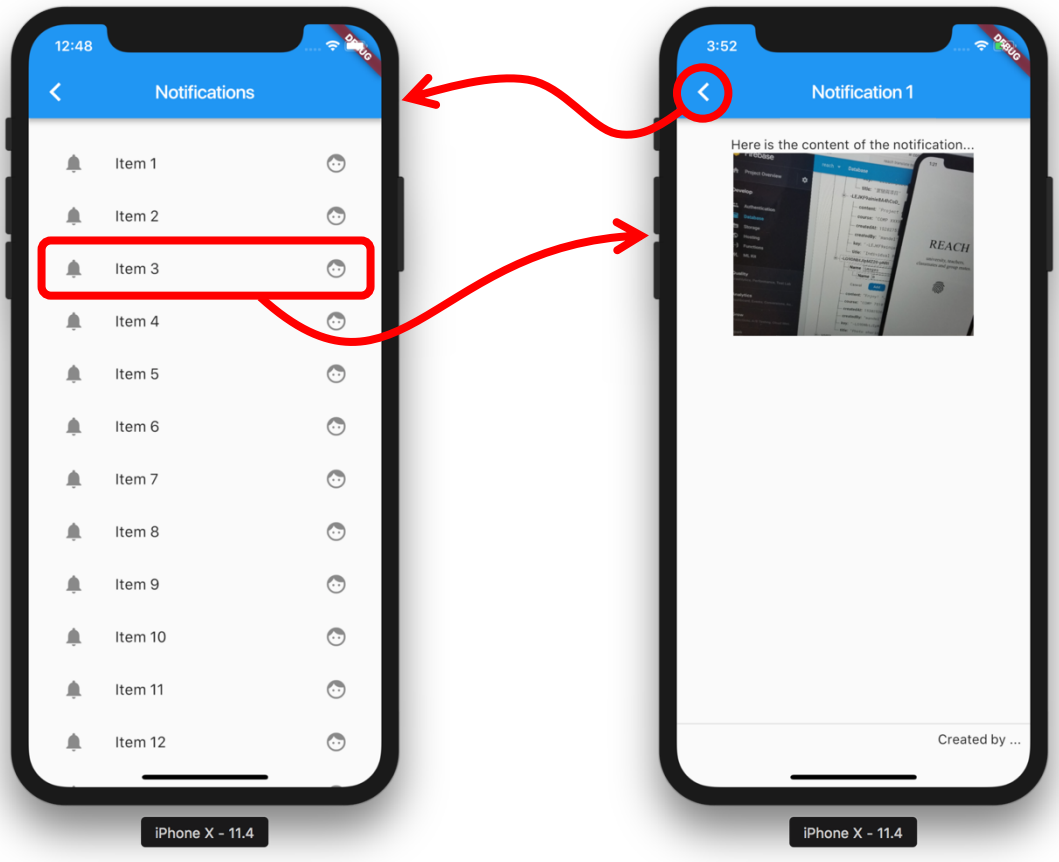
We then connect the notification list page to the notification view page.

4. Open the **notification_list.dart** file and set **On-Tap** event callback function to each **ListTile** widgets declared in the **build()** method of the **NotificationListState** class as follows:

```
    @override
    Widget build(BuildContext context) {

       // lines are omitted…

      for (var i =1; i <= 20; i++){
        widgetList.add(
            ListTile(
              leading: Icon(Icons.notifications),
              title: Text('Item $i'),
              trailing: Icon(Icons.face),

              onTap: () {
                Navigator.pushNamed(context, '/notificationView');
              }

            )
        );
      }

      // lines are omitted…

    }      // end of the build() method
```

- The callback function associated with the **onTap** property creates a **NotificationView** object and puts it to the navigator. When the user taps on one of the list items, the notification view page then shows up.

5. Run your app again to see the result.



Sharing Information

Currently, the notification view page shows the exactly same thing whichever item you tap. It is because there is no information passed by the list page to the view page.

The easier way to pass information between the pages is using global variables. We declare the global variables outside any class, so everyone can access these global variables. To make our program codes tidy, we are going to create a new dart file and then we declare all global variables inside it.

1. Create a new dart file named **global.dart**. And, add the following codes to it.
```
var notificationSelection = null;
```

2. Add the following line to the top of the **notification_list.dart** file.
```
import 'global.dart';
```

3. Go back to the **build()** method of the **NotificationListState** class. We make a small change in the for loop:

```
    for (var i =1; i <= 20; i++){
➔     var item = 'Notification $i';
      widgetList.add(
        ListTile(
          leading: Icon(Icons.notifications),
          title: Text('Item $i'),
          trailing: Icon(Icons.face),

➔         onTap: () {
➔           notificationSelection = item;
➔           Navigator.pushNamed(context, '/notificationView');
➔         }
        )
      );
    }
```

   - We declare variable **item** inside the *for* loop that will be used to store the value recently retrieved from the data list.

   - We also change the callback function of the **On-Tap** event handling (line 23 ~ 26) – assign the value of variable **item** to the global variable **notificationSelection**. So, when the user taps one of the items on the list, the **item** representing the selected item will be recorded.

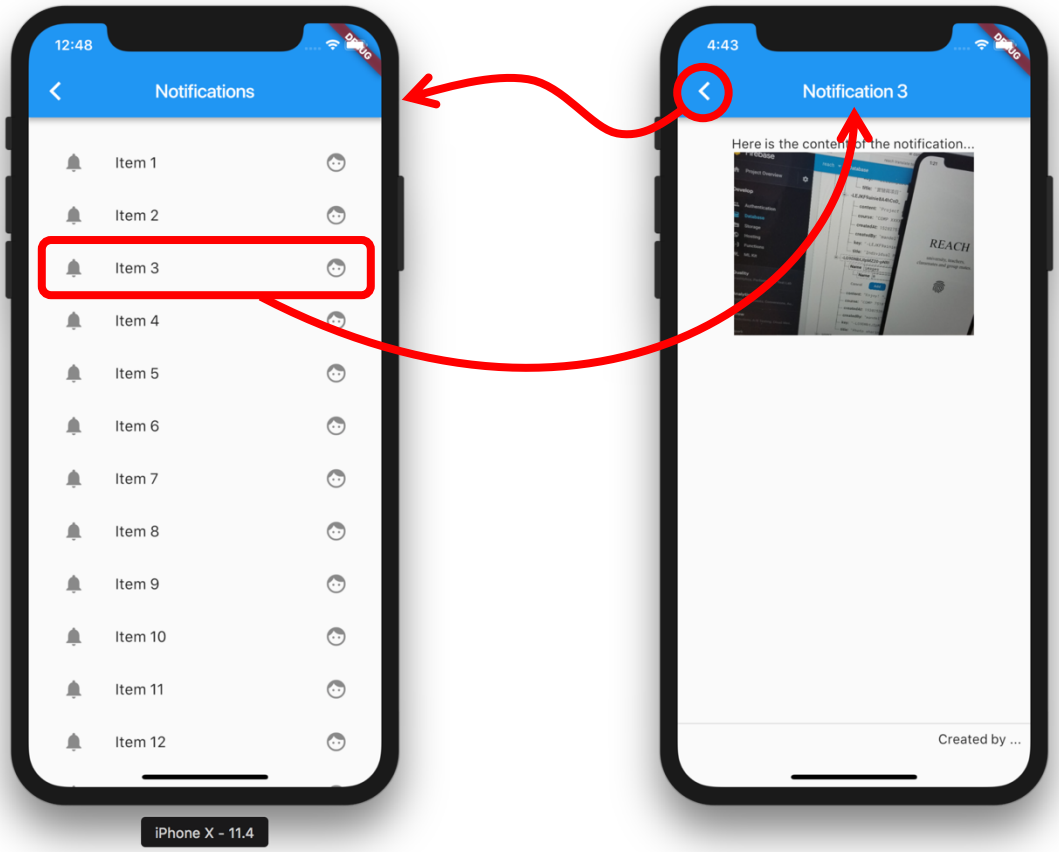
4. Next, we add the following line to the top of the **notification_view.dart** file.
```
import 'global.dart';
```

5. Then, modify the **build()** method of the **NotificationViewState** class to show the value of variable **notificationSelection** as follows:

```
    @override
    Widget build(BuildContext context) {
      return Scaffold(
        appBar: AppBar(
➔         title: Text('$notificationSelection'),
        ),

        // lines are omitted…

      );     // the end of the Scaffold widget
    }        // the end of the build() method
```

   - We only update the content of the title of the scaffold.

6. Run your app again to see the result.



Input Widgets

Currently, we have learned different widgets, but most of them are for showing the data or formatting the layouts. We now are going to learn the widgets for getting user inputs.

TextField

The **TextField** widget is used for handling user's text input. When the user tries something in the **TextField** widget, its *On-Changed* callback function will be invoked. The callback function can be used for updating internal variables, input validation or so on.

In addition, the **TextField** widget has many properties for customizing its input mode, such as:

- The **keyboardType** property specifies the keyboard type to multiple lines, email input, number input, etc.
- The **decoration** property shows the hint text when the text field is empty.
- The **maxLines** property specifies the maximum number of lines. The value equals null that indicates no limitation. Its default value is 1.

The following example code is used to declare a **TextField** widget for multiple lines text input:

```
TextField(

  decoration: InputDecoration(
    hintText: 'Content',
  ),

  keyboardType: TextInputType.multiline,

  maxLines: null,

  onChanged: (text) => setState(() => content = text),

),
```

*Assume that the global variable **content** is declared for storing the user input.*

DropdownButton

The **DropdownButton** widget provides a dropdown list for selecting items (**DropdownMenuItem** widgets). When the user changes the selection of the **DropdownButton** widget, its *On-Changed* callback function will be invoked. And, its **value** property then stores the selection.

The following example code is used to create **DropdownMenuItem** widgets:

```
var keys = ['ALL', 'COMP 7510', 'COMP XXXX'];
var items = <DropdownMenuItem>[];

for (var k in keys) {
  items.add(DropdownMenuItem(child: Text(k), value: k,));
}
```

The following example code is used to create a **DropdownButton** widget:

```
DropdownButton(
  value: selectedCourse,
  items: items,
  onChanged: (course) => setState(() => selectedCourse = course),
),
```

Switch

The switch widget is an ON/OFF button. The user drags it to left or right for setting on or off.

The following example code is used to create a Switch widget:

```
Switch(
  value:onoff,
  onChanged: (value) => setState(()=> onoff = value),
),
```

Lab Exercises

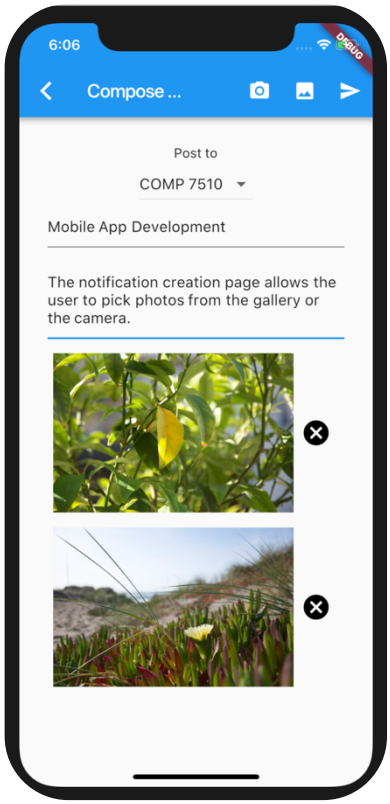
We learned how to make four different user interfaces including the sign-in page, main menu page, notification list page and notification view page. But, currently we do not have a user interface for notification creation.

Your task is to make use of the widgets you learned to create a new user interface. You may create a dart file named **notification_creation.dart**. And, declare a **NotificationCreationPage** class and a **NotificationCreationState** class in the dart file.

The notification creation page should include:

1. A **DropdownButton** widget for the target course selection
2. A **TextField** widget for title input
3. A **TextField** widget for content input
4. **IconButton** widget(s) for image selection (no callback function is required yet)
5. An **IconButton** widget for posting notification (no callback function is required yet)
6. **Image** widget(s) for showing images (you may copy the URLs from the google search)

Submit your **notification_creation.dart** file before the next lab.



Appendix – Reopening Flutter Project in Your Own Mac

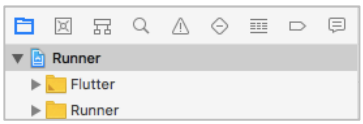
You may want to open your Flutter project in your own Mac. To do so, you must follow the steps shown in the **Appendix – Software Installation (macOS)**. Then, you need to follow the steps below:

1. Launch IntelliJ. In the Welcome window, click "Open".
2. In the Finder, find your Flutter project folder and click "Open".
3. Then, click IntelliJ's menu "IntelliJ IDEA" > "Preferences…".
4. Expand the "Languages & Frameworks" and select "Flutter" shown on the left-hand side of the Preferences window.
5. Select the correct **Flutter SDK path** using the "…" button and click "OK" to commit.
6. Go to the "Terminal" panel shown in the bottom, and type the following line and press ENTER to execute it.
   ```
   ~/development/flutter/bin/flutter clean
   ```
7. Open the "pubspec.yaml" file and click the "Get Packages" link.
8. Launch iOS simulator and run your program to check whether it runs normally or not.

Appendix – Solving Xcode 10 Build Problem

If you use Xcode 10, you will get Xcode build problem. The following steps show you how to solve the problem:

1. In the **Project** panel of IntelliJ (left-hand side), expand the **ios** folder.
2. Right-click the "Runner.xcworkspace" folder and click "Flutter" > "Open iOS module in Xcode".
3. Then, the Xcode launches and shows the **ios** module of your Flutter project.
4. In the Xcode, click the menu "Navigate" > "Reveal in Project Navigator".
5. Select the **Runner** project (the first Runner item) in the **Project Navigator** shown on the left-hand side.



6. Click "Build Phases" shown at the top of the main area. Then, expand the "Embed Frameworks" phase.

7. Click the "Flutter.framework" item and then click " – " to delete it.
8. Close the Xcode and go back to Intellij.
9. Run your program again. It should be built and run inside the simulator without any problem.

Appendix – Software Installation (macOS)

If you want to set up the development environment for your own Mac, you may follow the steps below:

1. Upgrade mac*OS* to the latest 64-bit version through the *App Store* (recommended).

2. Install *Xcode* through the App Store. Or, if you have *Xcode* installed in your Mac, upgrade the *Xcode* to the 9.4.1 version or later. After the installation, you have to launch it and install the required components.

3. Run the following command in the terminal after the Xcode installation.
```
sudo xcode-select -switch /Applications/Xcode.app/Contents/Developer
```

4. Install *Homebrew* by running the following command (single line) in the terminal.
```
/usr/bin/ruby -e "$(curl -fsSL
        https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

5. Install iOS required components by running the following commands in the terminal.
```
brew install cocoapods
pod setup
brew install -HEAD libimobiledevice
brew install ideviceinstaller
brew install ios-deploy
```

6. Launch *Xcode*, and then click the menu "Xcode" > "Open Developer Tool" > "Simulator". On the Simulator selection window, select "iPhone X". You will send the simulator prompts and running *iOS* as like as *iPhoneX*. At this moment, you may close *Xcode*, but keep the simulator up and running.

7. Download *IntelliJ IDEA* Community 2018.1.6 version (in the following called "*IntelliJ*") or later from the official website and install it.
   https://www.jetbrains.com/idea/download

8. Launch *IntelliJ*. If you first launch it, you simply click the "Next" button (several times) to skip the first-launch settings.

9. At the bottom right corner of the Welcome window of IntelliJ, click "Configure" and then "Plugins".

10. On the Plugins window, type "flutter" in the search box, and then click the link "Search in repositories".

11. On the Browse Repositories window, click "Install" of *Flutter* plug-in, and accept the privacy note and dependencies installation. Then, restart the *IntelliJ*.

12. Download *Flutter SDK* through the following URL.
    https://storage.googleapis.com/flutter_infra/releases/beta/macos/flutter_macos_v0.5.1-beta.zip

13. Unzip the downloaded zip file and copy the *flutter* directory to */Users/XXXX/development/* directory (where XXXX is your current username).

14. Run the following commands in the terminal.
```
cd ~
export PATH=`pwd`/flutter/bin:$PATH
flutter doctor
```

The output of the flutter doctor should be as follows:

```
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel beta, v0.5.1, on Mac OS X 10.13.6 17G65, locale en-HK)
[✗] Android toolchain - develop for Android devices
    ✗ Unable to locate Android SDK.
      Install Android Studio from: https://developer.android.com/studio/index.html
      On first launch it will assist you in installing the Android SDK components.
      (or visit https://flutter.io/setup/#android-setup for detailed instructions).
      If Android SDK has been installed to a custom location, set $ANDROID_HOME to that location.
[✓] iOS toolchain - develop for iOS devices (Xcode 9.4.1)
[✗] Android Studio (not installed)
[✓] IntelliJ IDEA Community Edition (version 2018.1.6)
[✓] Connected devices (1 available)
! Doctor found issues in 2 categories.
```

*** Note: You may get another locate is the doctor summary, but it is fine.*

*** Note: The categories with ✗ can be ignored because these settings are for Android platform only.*

References

[1] Display images from the internet. (n.d.). Retrieved from https://flutter.io/cookbook/images/network-image/

[2] Firebase for Flutter. (n.d.). Retrieved from https://codelabs.developers.google.com/codelabs/flutter-firebase/#0

[3] firebase_database | Flutter Package. (n.d.). Retrieved from https://pub.dartlang.org/packages/firebase_database

[4] firebase_storage | Flutter Package. (n.d.). Retrieved from https://pub.dartlang.org/packages/firebase_storage

[5] Flutter - Beautiful native apps in record time. (n.d.). Retrieved from https://flutter.io/

[6] A Tour of the Dart Language. (n.d.). Retrieved from https://www.dartlang.org/guides/language/language-tour