ХЕРСОНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ
КАФЕДРА ПРОГРАМНИХ ЗАСОБІВ І ТЕХНОЛОГІЙ

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

і контрольні завдання до виконання лабораторних робіт з дисципліни
ТЕХНОЛОГІЇ КРОС-ПЛАТФОРМНОЇ РОЗРОБКИ

для студентів	4 курсу, 2 курсу (за скороченим терміном навчання)
підготовки	першого (бакалаврського) рівня вищої освіти
галузі знань	12 «Інформаційні технології»
спеціальності	121 «Інженерія програмного забезпечення»
освітньо-професійних програм	Програмна інженерія
факультету	Інформаційних технологій та дизайну

Хмельницький – 2024

Методичні рекомендації і контрольні завдання до виконання лабораторних робіт з дисципліни «Технології крос-платформної розробки» підготовки фахівців на першому (бакалаврському) рівні вищої освіти спеціальності 121 «Інженерія програмного забезпечення».

РОЗРОБНИКИ: Хохлов В.А., к.т.н.

РЕЦЕНЗЕНТ: Вороненко М.О., к.т.н., доцент

Затверджено на засіданні кафедри програмних засобів і технологій

Протокол № 1 від «04» вересня 2024 року

Завідувач кафедри

_____ доц., к.т.н. О.Є.Огнєва

УЗГОДЖЕНО:
З НАВЧАЛЬНО-МЕТОДИЧНИМ ВІДДІЛОМ

Реєстраційний номер № _____

Зміст

Інструктаж з техніки безпеки (охорони праці на робочих місцях)

1. Загальні вимоги

1.1. До роботи у комп’ютерному класі допускаються студенти, які пройшли інструктаж з техніки безпеки з відповідним записом у журналі з техніки безпеки і підписами.

1.2. Не можна заходити й перебувати у комп’ютерному класі без викладача.

1.3. Робота у комп’ютерному класі має проводитися тільки в суворій відповідності до розкладу занять і графіка самостійної роботи викладача та студентів.

1.4. Студентам заборонено відчиняти шафи живлення і комп’ютери як тоді, коли ЕОМ працюють, так і тоді, коли вони вимкнені.

2.Вимоги безпеки перед початком роботи

2.1. Заборонено заходити до класу у верхньому одязі чи приносити його з собою.

2.2. Заборонено приносити на робоче місце особисті речі, дискети і т.п., крім ручки і зошита.

2.3. На робочому місці слід сидіти так, щоб можна було, не нахиляючись користуватися клавіатурою, і водночас повністю бачити зображення на екрані дисплея.

2.4. Починати роботу можна лише за вказівкою викладача або інженера.

3.Вимоги безпеки під час роботи

3.1. Заборонено ходити по комп’ютерному класу, голосно розмовляти.

3.2. Виконувати слід тільки зазначене викладачем завдання. Категорично заборонено виконувати інші роботи.

3.3. На клавіші клавіатури потрібно натискати плавно, не припускати ударів.

3.4. Користуватися друкувальним пристроєм дозволяється тільки у присутності викладача або інженера.

3.5. Заборонено самостійно переміщувати апаратуру.

3.6. Заборонено запускати ігрові програми.

3.7. У випадку виникнення неполадок треба повідомити викладача або інженера.

3.8. Не намагатися самостійно відрегулювати апаратуру або усувати в ній неполадки.

4.Вимоги безпеки після закінчення роботи

4.1. Про хиби та неполадки, помічені під час роботи, слід зробити записи у відповідних журналах.

4.2. На робочому місці не потрібно залишати зайвих предметів.

5.Вимоги безпеки в аварійних ситуаціях

5.1. При появі незвичайного звуку або вимкнення апаратури потрібно негайно припинити роботу й довести це до відома викладача або інженера.

5.2. При появі запаху паленого слід припинити роботу, вимкнути апаратуру і повідомити про це викладача чи інженера. Коли це необхідно, допомогти гаситипожежу.

5.3. При потраплянні людини під напругу необхідно знеструмити відповідне робоче місце, надати першу долі карську допомогу і викликати "швидку".

5.4. При виникненні пожежі необхідно знеструмити клас, викликати пожежну команду і приступити до гасіння пожежі засобами, які є.

5.5. У разі недотримання студентами вимог із охорони праці та пожежної безпеки адміністрація вузу може притягти їх до дисциплінарної та адміністративної відповідальності.

Лабораторна робота №1. Налагодження середовища для Flutter-розробки

Тема роботи: встановлення та налагодження середовища для Flutter-розробки

Мета роботи: встановити необхідні компоненти для розробки

Теоретичні відомості

Основна сторінка з нформацією про установку Flutter на різних платформах: https://docs.flutter.dev/get-started/install

Розробку Flutter-програм можна вести в таких інстументах, як Vim (NeoVim), Android Studio, VS Code. Майже всі редактори мають плагіни для підтримки Flutter-розробки.

Розглянемо кроки, що необхідно виконати для розробки на flutter програм для Android.

Щоби створити новий проект наобхідно вибрати “New Flutter Project”. Після цього вказуються наступні параметри проекту:

- project name

- project location

- project type

- supported paltforms

та ряд інших.

Після цього Android Studio створить мінімальний робочий проект на Flutter. Проект буде розміщего в каталозі наступної структури:

- android - Auto generated source code to create android application
- ios - Auto generated source code to create ios application
- lib - Main folder containing Dart code written using flutter framework
- lib/main.dart - Entry point of the Flutter application
- test - Folder containing Dart code to test the flutter application
- test/widget_test.dart - Sample code
- .gitignore - Git version control file
- .metadata - auto generated by the flutter tools
- .packages - auto generated to track the flutter packages
- .iml - project file used by Android studio
- pubspec.yaml - Used by Pub, Flutter package manager
- pubspec.lock - Auto generated by the Flutter package manager, Pub

- README.md - Project description file written in Markdown format

Сам проект має наступний код:

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

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: const MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}

class MyHomePage extends StatefulWidget {
   const MyHomePage({Key? key, required this.title}) : super(key:
key);

  final String title;

  @override
  State<MyHomePage> createState() => _MyHomePageState();
```

```
}

class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            const Text(
              'You have pushed the button this many times:',
            ),
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.headline4,
            ),
          ],
        ),
      ),
```

```
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

Завдання

1. Встановити Flutter SDK для своєї операційної системи
2. Створити мінімальний Flutter-проект, що підтримує Android та web
3. Протестувати можоивості hot-reload

Лабораторна робота №2. Агрегатні типи даних Dart

Тема роботи: агрегатні типи даних Dart

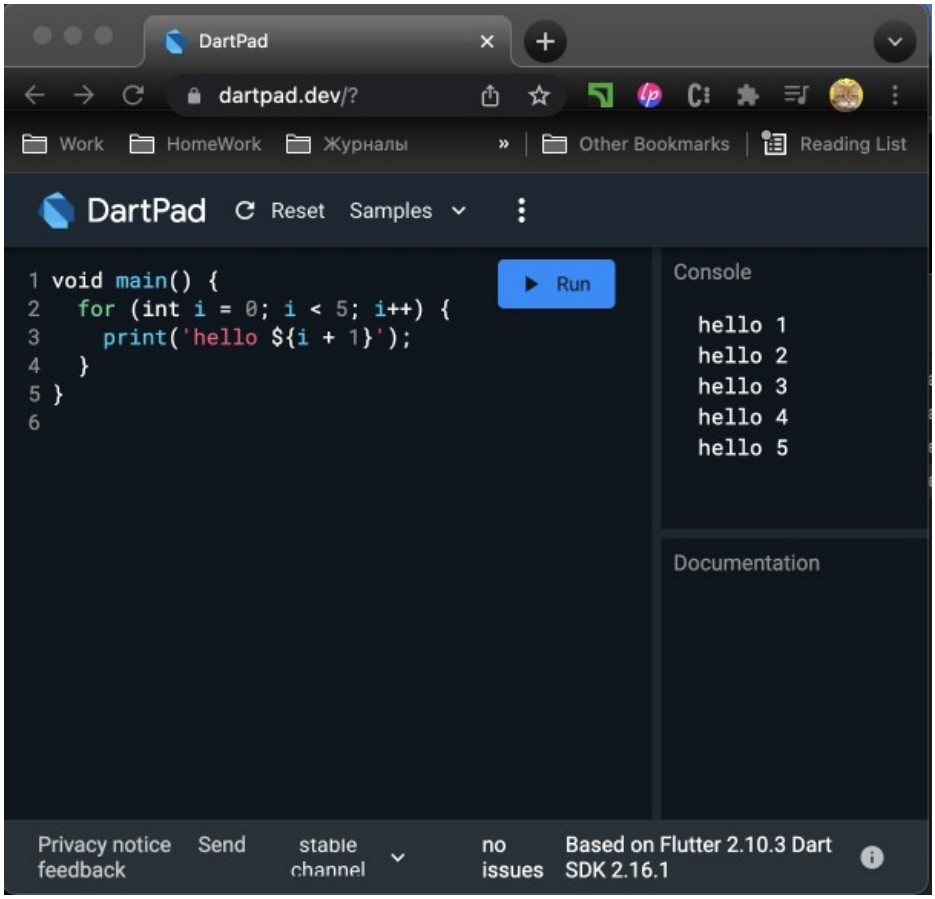
Мета роботи: ознайомитись с типами даних List

Теоретичні відомості

DartPad

Одним з зручних інструментів для вивчення Dart та Flutter є онлай-редактор коду DartPad: https://dartpad.dev/.

Приклад вікна DartPad з простою програмою:



Програм демонструє одну з модливостей рядків в Dart — string interpolation: якщо рядок містить фрагмент ${вираз}, то вираз буде обчислено перед використанням рядка.

Колекції в Dart

Мова Dart містить декілька классів для зберігання й обробки великої кількості об’єктів (агрегатні типи або колекції):

- масиви — List
- словники — Map
- множини — Set

List — це упорядкована група об’єктів, тобто кожен об’єкт має свій унікальний індекс. В списку може бути довільна кількість об’єктів з одним й тим самим значенням. Для створення списків використовуються або конструктори класу List, або літерали:

```
var list = [1, 2, 3];
```

Set — це неупорядкована група унікальних об’єктвів, тобто елементи не мають індексів і множтина на може містити кілька об’єктів з одним й тим самим значенням. Для створення множин використовуються або конструктори класу Set, або літерали:

```
var halogens = {'fluorine', 'chlorine', 'bromine', 'iodine', 'astatine'};
```

Словник — це об’єкт, що пов’язує ключі зі значеннями. Кожен ключ має бути унікальним,а значення можуть повторюватись. Для створення словників використовуються або конструктори класу Map, або літерали:

```
var gifts = {
  // Key:    Value
  'first': 'partridge',
  'second': 'turtledoves',
  'fifth': 'golden rings',
};
```

Колекції є параметризованими типами або генеріками, тобто можуть працювати з об’єктами будь-якого типу. Тип об’єктів вказується в <>:

```
List<int> emptyListOfInts = [];
```

Dart також має спеціальні оператори collection if та collection for для створення колецій в залежності від умови або циклу.

Lists

Клас List має декілька іменованих конструторів, для створення спискив:

- empty({bool growable = false}) — створює пустий масив
- filled(int length, E fill, {bool growable = false}) — створює масив з заданою кількістю однакових елементів
- generate(int length, E generator(int index), {bool growable = true}) — створює масив за допомогою спеціальної функції-генератора елементів

Приклад collection if при створенні масиву:

var nav = ['Home', 'Furniture', 'Plants', if (promoActive) 'Outlet'];

Також клас List має багато властивостей, методів та операторів для роботи з його елементами:

- length — кількість елементів в списку

- first — перший елемент
- last — останій елемент
- isEmpty — повертає true, якщо масив пустий
- add(E value) - додати елемент
- clear() - видалити всі елемети з масиву
- indexOf(E element, [int start = 0]) — повертає індекс першого елементу зі вказаним значенням
- removeAt(int index) — видаляє з масиву елемент з вказаним індексом
- indexWhere(bool test(E element), [int start = 0]) — повертає індекс першого елементу, що задоволняє вказаній умові
- operator +(List<E> other) - додає один масив до іншого
- operator [](int index) — повертає елемент з вказаним індексом
- operator []=(int index, E value) — встановлює нове значення для елементу з вказаним індексом

Розглянемо приклад програми, що демонструє використання деяких методів та операторів класу List:

```
void main() {
  List<int> array = [ for(int i = 1; i< 6; i++) i,];
  array[1] = array[3] + 1;
  print("Array: $array");
  print("First item: ${array.first}");
  array.removeLast();
  print("Array: $array");
  array.add(7);
  array += [9, 11];
  print("Array: $array");
  final firstEvenIndex = array.indexWhere((e) => e % 2 == 0);
  print("Index of first even item: $firstEvenIndex");
  print("First even item: ${array[firstEvenIndex]}");
}
```

Завдання

Розробити програму, що виконує над заданим масивом вказані дії. Номер варіанту визначається наступним чином:

$$номер\_варианту = номер\_по\_списку \ \% \ 5 + 1$$

Варіант	Вид обробки	Тип елементів
1	Знайти мінімальний елемент	int
2	Сортувати в зростаючому порядку	double
3	Знайти індекс елементу з заданим значенням	String
4	Знайти суму додатних чисел	int
5	Знайти максимальне від’ємне число	double

Лабораторна робота №3. Підтримка об’єкто-орієнтованого програмування в Dart

Тема роботи: підтримка об’єкто-орієнтованого програмування в Dart

Мета роботи: ознайомитись з підтримкою ООП в Dart

Теоретичні відомості

Dart — це об’єктно-орієнтована мова з класами та успадкуванням на основі міксинів (mixin-based inheritance). Кожен об’єкт є екземпляром класу, і всі класи, крім Null, походять від Object. Спадкування на основі міксинів означає, що хоча кожен клас (крім верхнього класу, Object?) має рівно один суперклас, тіло класу можна повторно використовувати в кількох ієрархіях класів.

Декларація класу може містити:

- поля
- конструктори
- гетери та сетери
- методи

Приклад декларації класу:

```
class Employee {
      // поле
      String name;
      Employee(String name) {
           this.name = name;
      }
      //getter method
      String get emp_name {
            return name;
      }
      //setter method
      void set emp_name(String name) {
            this.name = name;
      }
      //method definition
      void printName() {
            print(name);
      }
}
```

Приклад використання:

```
void main() {
 //object creation
 Employee emp = new Employee(“FirstLastName”);
 emp.name="employee1";
 emp.printName ();
}
```

Для створення класу на основі базового класу використовується синтаксис:

```
class DerivedCalss extends BaseClass {
...
}
```

Якщо ім’я поля чи метода починається з ‘_’, воно є приватним, в іншому випадку — публичним.

Завдання

Розробити програму, що містить наступні класи:

1. Person з полями для призвіща та імені

2. Developer, що є дочірним класом для Person і містить додаткове поле type: front-end, back-end, mobile

3. ProductManager, що є дочірним класом для Person і містить поле для списку підлеглих Developer’s, а також методи для додавання, видалення розробників в список і друку списку.

Лабораторна робота №4. Базові відомості про віджети Flutter

Тема роботи: базові відомості про віджети Flutter

Мета роботи: ознайомитись з принципами побудови програм на базі віджетів

Теоретичні відомості

Основна ідея Flutter — все є віджет. Інтерфейс програми представляє собою композицію віджетів, які вкладаються один в інший в потрібному порядку.

У Flutter віджети можна згрупувати в кілька категорій на основі їхніх функцій, як наведено нижче:

- Спеціальні віджети платформи
- Віджети макета
- Віджети обслуговування стану
- Незалежні від платформи

Розглянемо деякі базові віджети.

Віджет Text використовується для відображення фрагмента рядка. Стиль рядка можна встановити за допомогою параметру style і класу TextStyle. Зразок коду для цієї мети виглядає так:

```
Text('Hello World!', style: TextStyle(fontWeight:
FontWeight.bold))
```

Віджет TextButton відображає кнопку (як правило, з текстом), що реагує на натискання.

Найважливіши властивості:

- child, Widget: віджет для візуального представлення кнопки
- style, ButtonStyle: задає стилі кнопки
- onPressed, VoidCallback?: функція, що буде викликана при натисканні на кнопку; якщо null, то кнопка буде неактивною

Приклад:

```
TextButton(
      style: TextButton.styleFrom(textStyle: const TextStyle(fontSize: 20)),
      onPressed: () {},
      child: const Text('Enabled'),
)
```

Віджет Image відображає на екрані вказане зображення. Віджет має декілька конструкторів, зокрема Image.asset, що завантажує зображення з ресурсів додатка (assets). Ці ресурси розповсюджуються разом з додатком. В кості ресурсів можуть будти будь-які файли:

зображення, звуки, шрифти, якісь спеціальні файли. Всі ресурси, що мають бути включені в додаток, мають буьти вказані в файлі pubspec.yaml, який мість інформацію, необхідну для побудови додатка. Ресурси задаються в секції assets. Щоб додати зображення в список ресурсів додатка необхідно (бажано) в дереві проекту створити окремий каталог, додати в цей каталог зображення і вказати його в pubspec.yaml:

```
flutter:
  assets:
    - assets/smiley.png
```

Після цього зображення можно використовувати:

```
Image.asset('assets/smiley.png')
```

При редагуванні файлу pubspec.yaml слід бути уважним, оскільки має значення кількісти пробілів на початку кожного рядка.

Віджет Column дозволяє розташувати його дочірні елементи у вертикальному положенні. Основним аргументом його конструктора є children — це список дочірних віджетів для розміщення.

У Flutter всі віджети є похідними від StatelessWidget або StatefulWidget.

Віджет, похідний від StatelessWidget, не має жодної інформації про свій стан і використовується для побудови незмінних віджетів, наприклад для відображення назви додатка. Якщо ж вигляд віджету залежить від його внутрішнього стану (його змінних) слід використовувати StatefulWidget. З кожним класом, похідним від StatefulWidget пов’язаний клас, похідний від класу State, який й керує станом віджету. Основним методом StatelessWidget і State є метод build, що виконує побудову зовнішнтого вигляду віджету. Як правило, цей метод будує дерево з дочірних віджетів. Цей метод має буди перевизначеним в нових класах. Ще обним важливим методм в класі State є метод setState, виклик якого призводить до перебудови віджету.

Розглянемо програму, що створюється автоматчино при створенні нового flutter-проекту.

```
// імпорт  пакету fliutter/material. Цей пакет дозволяє створювати
//интерфейс відповідно до вказаних рекомендацій щодо дизайну на Android.

import 'package:flutter/material.dart';

//Це точка входу до програми Flutter. Викликає функцію runApp і передає
//їй об’єкт класу MyApp. Метою функції runApp є приєднання віджету до екрана.

void main() => runApp(const MyApp());

//Віджети використовуються для створення інтерфейсу  у фреймворку flutter.
//StatelessWidget є віджетом, який не підтримує жодного стану віджета.
```

```
//MyApp розширює StatelessWidget і перевизначає його метод build. Метою цього
//методу є створення частини інтерфейсу користувача програми.

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

//Метод build створює стандартний віджет   MaterialApp і передає йому
//в якості дочірнього новий віджет  MyHomePage

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
        title: 'Flutter Demo',
        home: MyHomePage(title: 'Flutter Demo Home Page'));
  }
}

//Вигляд віджету  MyHomePage залежить від значення внутрішнтої змінної, тому він
//розширює класс  StatefulWidget

class MyHomePage extends StatefulWidget {
  const MyHomePage({Key? key, required this.title}) : super(key: key);

  final String title;

  @override
  State<MyHomePage> createState() => _MyHomePageState();
}

//З кожним  віджетом StatefulWidget пов’язана окрема змінна классу  State

class _MyHomePageState extends State<MyHomePage> {

//Змінна, що впливає на вигляд віджету
  int _counter = 0;

  void _incrementCounter() {
```

```
//Щоб повідомити Flutter про необхідність перебудови віджету необхідно
//викликати метод setState
    setState(() { _counter++; });

  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text(widget.title)),
      body: Center(
          child: Column(
              mainAxisAlignment: MainAxisAlignment.center,
              children: <Widget>[
            const Text('You have pushed the button this many times:'),
            Text('$_counter')
          ])),

//Кнопка, що по натискані викликає вказанний метод

floatingActionButton: FloatingActionButton(
          onPressed: _incrementCounter,
          tooltip: 'Increment',
          child: const Icon(Icons.add)),
    );
  }
}
```

Завдання

Розробити програму, що складається з трьох віджетів, що розміщені в стовбчик один під одним:

- тексту
- зображення
- кнопки

При натисканні на кнопку віджет тексту відображає рядок “Even”, якщо поточна кількість натискань є парним числом, та “Odd” — якщо непраним. Віджет зобрадення послідовно відображає два зображення один за одним.

Лабораторна робота №5. Віджети макетів

Тема роботи: віджети макетів

Мета роботи: ознайомитись з принципами побудови програм на базі віджетів макетів

Теоретичні відомості

Оскільки основна концепція Flutter полягає в тому, що все є віджетом, Flutter включає функціональність макетування інтерфейсу в самих віджетах.

Віджети макета можна згрупувати у дві окремі категорії залежно від їх дочірнього елементу:

- віджет для підтримки одного дочірнього елементу (Single child widgets)
- віджет, який підтримує декілька дочірних елементів (Multiple child widgets)

Single child widgets мають лише один дочірній віджет, і кожен віджет має спеціальну функціональність. Наприклад, Center, Container, Padding та ін.

Multiple child widgets мають більше одного дочірнього віджета та макет кожного віджету унікальний.

Найчастіше використовуються наступні віджети:

- Row: дозволяє розташувати його дочірні елементи горизонтально.
- Column: дозволяє розташувати його дочірні елементи у вертикальному положенні.
- ListView: дозволяє впорядкувати його дочірні елементи у вигляді списку.
- GridView: дозволяє розташувати його дочірні елементи як галерею.
- Expanded: використовується для того, щоб дочірні віджети Row і Column займали максимально можливу площу.
- Table: віджет на основі таблиці.
- Flow: віджет на основі потоку.
- Stack: віджет на основі стека.

Кожен з віджетів має доволі багато параметрів, що дозволяють виконувати налаштування розмізення дочіних елементів.

Найбільш гнучким є Stack, який дозволяє розміщувати свої елементи в довільному порядку. Найчастіше разом зі Stack використовується віджет Align, що зозволяє задавати вирівнювання свого дочірного елементу.

Приклад побудови екрану з двома текстовими віджетами: один розміщується в верхній частині по центру, інший — в правому ніжньому кутку:

```
Center(
        child: Stack(
          children: const <Widget>[
```

```
        Align(
          alignment: Alignment.topCenter,
          child: Text('TopCenter'),
        ),
        Align(
          alignment: Alignment.bottomRight,
          child: Text('BottomRight'),
        ),
      ],
    ),
  ),
```

Завдання

Розробити програму, що демонструє використання віджетів макетів.

Якщо номер студента парний, програма має складатись з двох рядків, розміщених один під одним. Перший рядок мітстить два текстових віджети, нижній — два віджети з зображеннями.

Якщо номер студента непарний, програма має складатись з п’яти віджетів, що розміщені по кутках екрану і в центрі. У верхніх кутках — текстові віджети, у нижніх — віджети з зображенням, в центрі — віджет з іконкою.

Лабораторна робота №6. Керування станом Flutter-додатка

Тема роботи: керування станом Flutter-додатка

Мета роботи: ознайомитись з підходами до керування станом Flutter-додатка

Теоретичні відомості

Flutter — це фреймворк для побудови інтерфейсу користувача (UI). Але UI рідко коли буває сталим. Як правило, екран додатка має різний вигляд в залежності від тої чи іншої умови. Наприклад, фінансові додатки можуть відображати прибуток чи збиток різними кольорами, програма-календар може показувати поточний день іншим кольором, а сам календар відображати або списком днів, або таблицею, в залежності від налаштувань.

Найпростіший спосіб зберіати стан віджета — використовувати класи StatefulWidget і відповідний State. В класі State можна створювати відповідні поля для зберігання інформації про стан віджета, а оновлювати стан і викликати перебудову віджета за допомогою методу setState.

Такий підхід достатній для невиликих віджетів. Однак при роботі з додатком вцілому він стає складним для розуміння й супроводження.

Для Flutter розроблено достатеньо багато пакетів для керування станом додатка: ScopedModel, BloC (business logic component) та ін.

Розглянемо використання ще однієї бібліотеки — Provider. Для передачі даних з якогось рівня дерева віджетів на нижчі рівні у Flutter використовується спеціальний тип віджету — InherotedWidget. Provider — це обгортка над InheritedWidget, що спрошує роботи з ним.

Бібліотека Provider, як і інші додаткові бібліотеки для Flutter поставляється у вигляді спеціального плагіну (наявні плагіни можна переглянути на https://pub.dev). Щоб додати плагін до проекту, необхідно вказати його в файлі pubspec.yaml. Це спеціальний файл, який містить ріщноманітну інформацію про проект:

```
dependencies:
  flutter:
    sdk: flutter
  provider: ^6.0.4
```

Необхідно уважно ставитись до віступів у рядках цього файлу, оскільки формат суттєво залежить від них. Після оновлення файлу необхідно виконати команду:

```
flutter pub get
```

Тепер можна винести в окремий файл логіку роботи додатка. В прикладі з додатком-лічильником наша модель може мати вигляд:

```
import 'package:flutter/material.dart';
class CounterModel extends ChangeNotifier {
  int _counter = 0;
```

```
  int get counter => _counter;
  set counter(int value) {
    if (value != _counter) {
      _counter = value;
      notifyListeners();
    }
  }
}
```

ChangeNotifier — це спеціальний клас, що за допомогою метода notifyListeners() може повідомляти всім об’єктам, що підписані на нього (listeners) про зміни свого стану.

Плагін provider містить ряд різноманітних провайдерів, в нашому випадку необхідно використовувати ChangeNotifierProvider.

Загальний код з використанням такого провайдера має вигляд:

```
ChangeNotifierProvider<Model>(
      create: (_) => Model(),
      child: Consumer<Model>(
        builder: (context, model, child) {
          // Here you can use the model variable
          // to read and alter the state
        },
      ),
    )
```

Оскількі ми зберігаємо логіку нашого додатка не у віджеті, а в коремому класі, то можемо перейти від StatefulWidget до StatelessWidget:

```
class MyHomePage extends StatelessWidget {
  const MyHomePage({super.key, required this.title});
  final String title;
  @override
  Widget build(BuildContext context) {
    return ChangeNotifierProvider(
      create: (_) => CounterModel(),
      child: Consumer<CounterModel>(
        builder: (context, model, child) => Scaffold(
          appBar: AppBar(title: Text(title)),
          body: Center(
            child: Column(
              mainAxisAlignment: MainAxisAlignment.center,
```

```
              children: <Widget>[
                const Text('You have pushed the button this many times:'),
                Text('${model.counter}', style:
Theme.of(context).textTheme.headline4),
              ],
            ),
          ),
          floatingActionButton: FloatingActionButton(
            onPressed: () => model.counter++,
            tooltip: 'Increment',
            child: const Icon(Icons.add),
          ),
        ),
      ),
    );
  }
}
```

Таким чином, за допомогою Provider ми відокремили логічну частину програми від інтерфейсної. Така архітектура програми, коли є окремі частини програми виділяються в модель та вигляд, має назву Model-View-ViewModel — MVVM.

Пакет provider окрім віждету Consumer містить ряд віджетів, які дозволяють реагувати лише на зміну окремих полів моделі, зокрема віджет Select. Також Provider додає додаткові методи (select, read, тощо) до класу BuildContext, що спрощують використання виджетів Select.

З використанням ціх можливостей ми можемо створити окремий віджет, що буде відображати кількість натискань на кнопку. При чому, він буде змінюватись лише при змінні поля counter нашої моделі:

```
class CounterLabel extends StatelessWidget {
  const CounterLabel({super.key});


  @override
  Widget build(BuildContext context) {
    final value = context.select<CounterModel, int>((model) => model.counter);
    return Text('$value', style: Theme.of(context).textTheme.headline4);
  }
}
```

А фрегмент програми з FAB буде наступним:

```
floatingActionButton: FloatingActionButton(
     onPressed: () => context.read<CounterModel>().counter++,
```

```
      tooltip: 'Increment',
      child: const Icon(Icons.add),
),
```

Таким чином, при зміні стану буде перебудовуватись лише потрібна частина UI.

Завдання

Модифікувати програму лічильник наступним чином:

Додати два лічильники counter1 та counter2.

Замість floatingActionButton додати чотири кнопки: перша має збільшувати counter1, друга — зменшувати counter1, третя - має збільшувати counter2, четверта — зменшувати counter2.

Додати два індікатори, які будуть показувати поточне значення лічильників counter1 та counter2 відповідно. Кожен з індикаторів має перебудовуватись лише при змінні відповідного лічильника.

Лабораторна робота №7. Навігація у Flutter-додатках

Тема роботи: навігація у Flutter-додатках

Мета роботи: ознайомитись навігації у Flutter-додатках

Теоретичні відомості

Навігація є основною концепцією розробки мобільних додатків. Це дозволяє користувачам переходити з одного екрана на інший. Добре реалізоавна навігація забезпечує організованість програми та розширює її можливості.

Flutter пропонує імперативний API (Navigator 1.0) і декларативний API (Navigator 2.0) механізми маршрутизації. У Navigator 1.0 ви можете додати сторінку лише на вершину стеку навігації та видалити сторінку з вершини стеку. З іншого боку, декларативний механізм дозволяє вам повністю контролювати навігаційний стек. Також Navigator 2.0 спрощує навігацію у web-додатках.

Керує навігацією у fluter віджет Navigator. Для додавання сторінки на вершину стеку навігації використовується метод push. Класовий варіант цього метода приймає два аргументы: BuildContext та сторінку, які в термінах flutter называються маршрутами (routes). Фрагмент коду, що виконує перехід на нову сторінку має вигляд:

```
Navigator.push(
      context,
      MaterialPageRoute<bool>(builder: (BuildContext context) => GamePage6()));
```

Для повернення на попередню сторінку використовується метод pop. Класовий варіант цього метода приймає один обов’язковий аргумент — BuildContext:

```
Navigator.pop(context)
```

Якщо сторінці необхідно повернути якесь значення на попередній маршрут, то його можна передати через другий аргумент метода pop. Це значення буде доступне як значення, що повернув метод push. Таким чином, загальний фрагмент коду для навігації може мати наступний вигляд:

```
   final bool? shouldShowAd = await Navigator.push(
      context,
      MaterialPageRoute<bool>(
        builder: (BuildContext context) => GamePage4())
    );
...
Navigator.pop(context, true);
```

Navigator 1.0 має також інший підхід для визначення маршрутів у додатку. Для цього віджету MaterialApp передати в якості одного з аругментів словник (map) марщрутів та їх ідентифікаторів. Ідентифікатором маршруту може бути довільний рядок символів. Також необхідно вказати початковий маршрут:

```
MaterialApp(
...
initialRoute: '/',
routes: {
    '/': (BuildContext context) => const MenuPage(),
    '/help': (BuildContext context) => const HelpPage(),
    '/settings': (BuildContext context) => const SettingsPage(),
   },
)
```

Для переходу на один з зазначених маршрутів використовується метод pushNamed:

```
Navigator.pushNamed(context, '/help')
```

Для декларативного стилю визначення маршрутів в додатку Navigator 2.0 додає додаткові аргументи для віджета Navigator.

Можливі сторінки додатка задаються за допомогою аргумента pages. Цей список може перебудовуватись в залежності від стану додатку. Перебудова списку призводить до переходів між сторінками. Для керування станом в найпростішому випадку можна використовувати свої змінні і метод setState.

При завданні параметра pages обов’язковим є також аргумент onPopPage — це функція, яка буде викликана прі поверненні з будь-якої сторінки назад по стеку навігації.

```
class MyApp extends StatefulWidget {
  const MyApp({super.key});

  @override
  State<MyApp> createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
  bool _isDetailedPageOpened = false;

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Navigation Demo',
      theme: ThemeData(primarySwatch: Colors.blue),
      home: Navigator(
        pages: [
          MaterialPage(
```

```
            key: const ValueKey('homePage'),
            child: MyHomePage(
              title: 'Navigation Demo Home Page',
              onButtonTap: () {
                setState(() {
                  _isDetailedPageOpened = true;
                });
              },
            ),
          ),
          if (_isDetailedPageOpened)
            const MaterialPage(
              key: ValueKey('detailsPage'),
              child: DetailsPage(),
            ),
        ],
        onPopPage: (route, result) {
          if (!route.didPop(result)) {
            return false;
          }
          if (_isDetailedPageOpened) {
            setState(() {
              _isDetailedPageOpened = false;
            });
          }
          return true;
        },
      ),
    );
  }
}
```

Для більш складної навігації та підтримки deep links Navigator 2.0 містить спеціальний віджет Router, а також додаткові класи: RouteInformationProvider, RouteInformationParser, RouterDelegate, BackButtonDispatcher.

Але конфігурування безпосередньо віджету Router виявляється доволі складним. Тому користувачі flutter розробили декілька плагінів, що спрощують цей процес. Найбільш досконалим та популярним є плагін go_router.

Завдання

Розробити додаток, що складається з трьох сторінок: HomePage, DetailsPage, SettingsPage. Сторінка HomePage містить дві кноки: одна виконує перехід на DetailsPage, інша — на SettingsPage. Розробити дві версії додатка: з використанням Naviagator 1.0 та Navigator 2.0.

Лабораторна робота №8. Анімація у Fluter-додатках

Тема роботи: анімація у Fluter-додатках

Мета роботи: ознайомитись основами анімації у Flutter-додатках

Теоретичні відомості

Анімація є невід’ємною частиною інтерфейсу сучасних мобільних додатків. Плавні переходи та інтерактивні елементи роблять програму приємною у використанні. З іншого боку, погано зроблена анімація може зробити програму незграбною або, що ще гірше, взагалі зламати програму.

Flutter містить велику вількість анімірованих віджетів. Всі вони міняють один або декілька параметрів віджета за вказаний проміжок часу. За початкове значення параметра береться його попереднє значення, на кінцеве — нове. Тривалість анімації (duration) є обов’язковим параметром таких віджетів.

Розглянемо деякі з них:

- AnimatedAlign — міняє вирівнювання (ліворуч, праворуч, тощо...) віджета.
- AnimatedContainer — віджет Container, який змінює свої параметри.
- AnimatedOpacity — анімірована версія віджету Opacity, що задає прозорість свого дочірнього віджету.
- ScaleTransition — анімує масштаб віджету, що трасформується.
- AnimatedBuilder — загальний віджет для побудови анімації.

В якості прикладу додамо в стартовий проект (лічильник) віджет AnimatedContainer:

```
...
child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            AnimatedContainer(
              duration: const Duration(milliseconds: 300),
              color: Colors.green,
              height: 50,
              width: _counter * 20,
            ),
            const Text(
              'You have pushed the button this many times:',
            ),
            Text(
              '$_counter',
```

```
              style: Theme.of(context).textTheme.headlineMedium,
            ),
          ],
...
```

Такі зміни призведуть до появи над текстом зеленої полоски, яка буде отримувати нову ширину після кожного натискання на кнопку. Зміна буде відбуватися протягом 300 мілісекунд.

Розглянемо маханізми, які надає flutter для керування анімацією.

Однією з основних концепцій, пов’язаних з анімацією у flutter, є tweens. Ці об’єкти задають початкове та кінцеве значення якогось параметра (назва походить від слова beTWEEN). tween-об’єкти створюються або за допомогою generic-класу Tween<T> або за допомогою спеціалізованих класів, наприклад, ColorTween.

Для керування процесом анімації використовується Animation Controllers. Керування відбувається таким чином, що анімація змінює своє значення та повертає його на основі прогресу анімації.

Таким чином, кожна анімація Flutter потребує створення принаймні двох елементів:

- Tween, що генерує значення для анімації
- AnimationController, що керує прогресом анімації

AnimationController задає прогрес анімації від 0 до 1, тоді як Tween дає фактичне значення анімації, що очікується віджетом.

AnimationController також дає нам контроль над тим, як поводиться анімація за допомогою таких методів, як forward(), reverse() або repeat().

При створенні об’єкта AnimationController необхідно вказати як мінімум два переметри:

duration — час анімації

vsync — об’єкт, який надає так званні ticks — моменти часу, в які необхідно обчислити нові значення анімації. Для цього може застосовуватись, наприклад, міксін SingleTickerProviderStateMixin.

Завдання

Модифікувати додаток-лічильник таким чином, що розмір тексту індикатора лічильника збільшувався з використанням анімації. Розробити дві версії: з використанням вбудованого анімірованого віжету тексту і з використанням Tween та AnimationController.

Лабораторна робота №9. Робота з мережевими запитами у Flutter

Тема роботи: робота з мережевими запитами у Flutter

Мета роботи: Ознайомитись з flutter-підходом до роботи з мережевими запитами

Теоретичні відомості

Робота з Internet є важливою складовою сучасних мобільних додатків. Досить часто мобільний додаток працює в парі з серверною частиною, виконуючи різноманітні запити по мережеі Internet до сервера і відображаючи отриману інформацію в потрібному вигляді.

Одним з популярних flutter-пакетів для роботи з Internet є dio. Встановлюється пакет звичаним чином:

```
flutter pub add dio
```

або додається беспосередньо у файл pubspec.yaml в секцію dependencies:

```
dependencies:
  dio: ^x.y.z
```

Для Android-додатків потрібно також вказати в AndroidManifest.xml дозвіл на роботу з Internet:

```
<uses-permission android:name="android.permission.INTERNET" />
```

Об’єкт класу Dio зазвичай створюється наступним чином:

```
import 'package:dio/dio.dart';
...
final Dio _httpClient = Dio(BaseOptions(baseUrl:
'https://jsonplaceholder.typicode.com/'));
```

Окрім базової адреси конструктор приймає й інші параметри, що дозволяють виконувати потрібне налаштування об’єкту.

Отримання даних з серверу може виглядати наступним чином:

```
final url = 'posts';
final Response resp = await _httpClient.get<List<dynamic>>(url);
      posts = (resp.data as List<dynamic>)
          .map((e) => Post.fromJson(e as Map<String, dynamic>))
          .toList();
```

API Get https://jsonplaceholder.typicode.com/posts поверає список постів в форматі JSON:

```
[
  {
    "userId": 1,
    "id": 1,
```

```
    "title": "sunt aut facere repellat provident occaecati excepturi optio
reprehenderit",
    "body": "quia et suscipit\nsuscipit recusandae consequuntur expedita et
cum\nreprehenderit molestiae ut ut quas totam\nnostrum rerum est autem sunt rem
eveniet architecto"
  },
  {
    "userId": 1,
    "id": 2,
    "title": "qui est esse",
    "body": "est rerum tempore vitae\nsequi sint nihil reprehenderit dolor
beatae ea dolores neque\nfugiat blanditiis voluptate porro vel nihil molestiae
ut reiciendis\nqui aperiam non debitis possimus qui neque nisi nulla"
  },
...
]
```

Оскільки неможливо відразу визначити структуру даних, що повертається, метод get параметризується типом List<dynamic>. Методи класу Dio повертають як результат об’єкт класу Response. Цей клас містить ряд властивостей з інформацією про відповідть сервера: statusCode, headers, тощо. Відповідь сервера міститься у властивості data.

Для більш зручної обробки даних в програмах JSON-об’єкти після отримання перетворюються в dart-об’єкти. В даному випадку застосовується метод List.map, який перетворює об’єкти списку в інші об’єкти і повертає новий список. Іменований конструктор Post.fromJson створює об’єкт класу Post, використовуюся як аргумент JSON-словник з інформацією про пост:

```
class Post {
    Post({required this.id, required this.userId, required this.title, required
this.body});
    Post.fromJson(Map<String, dynamic> json)
      : id = json['id'] as int,
        userId = json['userId'] as int,
        title = json['title'] as String,
        body = json['body'] as String;

    final int id;
    final int userId;
    final String title;
    final String body;
}
```

Завдання

Розробити програму, що використовує сайт https://jsonplaceholder.typicode.com/ в якості сервера даних. Перша сторінка додатку має відображати список постів (posts): заголовок, тіло, ідентифікатор автора. Тап на будь-якому пості відкриває нову сторінку з інформацією про автора: ім’я, прізвище, email, адреса, номер телефону.

Лабораторна робота №10. Кодогенерація у Flutter

Тема роботи: кодогенерація у Flutter

Мета роботи: Ознайомитись з flutter-підходом до кодогенерації

Теоретичні відомості

Кодогенерація — це технологія автоматичного створення шаблоного коду, який постийно повторюється в різних проектах. Для Flutter розроблений ряд пакетів, що дозволяють по спеціальним анотациям генерувати такий код, як обробка json-даних, робота з мережею, тощо.

Для Flutter розроблено багато пакетів для кодогенерації. Пакет build_runner є особливим: він забезпечує функінування інших кодогенеруючих пакетів й керує їми. Цей пакет (як і інші для кодогенерації) використовується не під час роботи додатка, а під час його створення, тому він додається не в секцію dependencies, а в секцію dev_dependencies файла pubspec.yaml:

```
dev_dependencies:
  build_runner: ^2.4.12
```

Для запуску процесу генерації кода використовується команда:

```
dart run build_runner build -d
```

Параметр -d вказує на необхідність видалити при необхідності старі сгенеровані файли.

Анотації починаються з символу @, за яким йде назва анотації з можливими параметрами.

Пакет json_serializable використовується для генерації методів fromJson та toJson, які створюють Dart-об’єкти з json-словників та представляють Dart-об’єкти в json-форматі відповідно.

Для використання цього пакету його треба додати в секцію dev_dependencies та допоміжний пакет json_annotation:

```
dependencies:
  flutter:
    sdk: flutter
  json_annotation: ^4.9.0
dev_dependencies:
  build_runner: ^2.4.12
  json_serializable: ^6.8.0
```

Клас, для якого треба сгенерувати методи fromJson та toJson, необхідно відмітити анотацією @JsonSerializable():

```
import 'package:json_annotation/json_annotation.dart';
part 'post.g.dart';

@JsonSerializable()
```

```
class Post {
  Post({
    required this.id,
    required this.userId,
    required this.title,
    required this.body,
  });

  factory Post.fromJson(Map<String, dynamic> json) => _$PostFromJson(json);
  Map<String, dynamic> toJson() => _$PostToJson(this);

  final int id;
  final int userId;
  final String title;
  final String body;
}
```

fromJson — це фабричний конструктор, який викликає сгенерований метод _$PostFromJson, метод toJson — сгенерований метод _$PostToJson. Код цих методів знаходиться в файлі post.g.dart, який включається в файл post.dart директивою part.

Пакет retrofit використовується для генерації коду, що за допомогою пакету dio виконує мережеві запити й обробку відповідей. Для його використання потрібно додати відповідні залежності у файл pubspec.yaml:

```
dependencies:
  flutter:
    sdk: flutter
  dio: ^5.5.0+1
  json_annotation: ^4.9.0
  retrofit: ^4.1.0
dev_dependencies:
  build_runner: ^2.4.12
  json_serializable: ^6.8.0
  retrofit_generator: ^8.1.2
```

Приклад опису класу, що використовується для запитів до https://jsonplaceholder.typicode.com/, зокрема отримання посту за його ідентифікатором (id):

```
import 'package:dio/dio.dart';
import 'package:retrofit/retrofit.dart';

import '../domain/post.dart';
```

```
part 'retrofit_client.g.dart';

@RestApi(baseUrl: 'https://jsonplaceholder.typicode.com/')
abstract class RetrofitClient {
  factory RetrofitClient(Dio dio) = _RetrofitClient;

  @GET('/posts/{id}')
  Future<Post> getPost(@Path() int id);
}
```

Пакет містить багато анотацій, таких як @GET, @POST, @DELETE, @UPDATE, тощо для опису відповідних http-запитів, а також @Path, @Body для формування тіл запитів.

Сгенеровані класи можуть використовуватись наступним чином:

```
Future<Post?> _fetchPost() async {
    final dio = Dio();
    final client = RetrofitClient(dio);
    final post = await client.getPost(1);
    print('Title: ${post.title}');
    return post;
}
```

Завдання

Розробити програму, що використовує сайт https://jsonplaceholder.typicode.com/ в якості сервера даних. Перша сторінка додатку має відображати список постів (posts): заголовок, тіло, ідентифікатор автора. Тап на будь-якому пості відкриває нову сторінку з інформацією про автора: ім’я, прізвище, email, адреса, номер телефону.

Для обробки json-даних та запитів до Internet та відповідей треба використовувати кодогенерацію. Для отримання більше 70 балів треба використовувати пакет provider для керування станом додатка.

Рекомендована література

- Flutter Apprentice (Second Edition): Learn to Build Cross-Platform Apps / Mike Katz, Kevin David Moore, Vincent Ngo – Вид-во: “Razeware LLC”, 2021. – 615 с.
- Flutter TutorialPoints / Вид-во: “Tutorials Point (I) Pvt. Ltd.”, 2019. – 185 с.
- Dart Apprentice / Matt Galloway, Jonathan Sande – Вид-во: “Razeware LLC”, 2020. – 436 с.
- Gilad Bracha. The Dart Programming Language– Вид-во: “Addison-Wesley”, 2016р. – 201 с.
- Carmine Zaccagnino. Programming Flutter: Native, Cross-Platform Apps the Easy Way (The Pragmatic Programmers) 1st Edition – Вид-во: “Pragmatic Bookshelf”, 2020. – 370 с.
- Dart Portal – Режим доступу: https://dart.dev/
- Flutter Portal – Режим доступу: https://flutter.dev/