# Lập trình đa nền tảng

Nguyễn Duy Nhật Viễn

Chương 3

# Lập trình hướng đối tượng trong Dart

# Lớp

# Khái niệm

- Lớp: lớp của các đối tượng, trong Dart, mọi thứ đều là đối tượng. Các lớp đều kế thừa từ lớp Object
- Trong lớp có thể có các thành viên:
  - Các phương thức khởi tạo - Hàm được gọi khi tạo ra một đối tượng mới từ class
  - Các biến lưu dữ liệu của đối tượng - gọi là các trường - các thuộc tính – các dữ liệu thành viên
  - Các hàm - gọi là các hàm thành viên - các phương thức
  - Các hàm đặc biệt gọi khi thực hiện gán thuộc tính / truy cập thuộc tính - hàm setter/getter

# Khai báo và sử dụng

```
class Product {
  //Khai báo các thuộc tính
  String manufacture = '';
  String name = '';
  var price;
  int quantity=0;

  //Khai báo hàm khởi tạo
  Product(var price, {int quantity:0}) {
    this.price = price;
    this.quantity = quantity;
  }

  //Khai báo các phương thức
  calulateTotal() {
    return this.price * this.quantity;
  }

  showTotal() {
    var tong = this.calulateTotal();
    print("Tổng số tiền là: $tong");
  }
}
```

Sử dụng: để truy cập vào thành viên của đối tượng, sử dụng toán tử ‘.’ theo dạng thức: tên_đối_tượng.tên_thành_viên.

```
void main() {
  var product = new Product(600, quantity: 1);
  product.showTotal();

  product.quantity = 2;
  product.showTotal();
}
```

Trong phương thức của lớp, để tham khảo đến đối tượng của lớp dùng từ khóa this, ví dụ trong hàm calulateTotal() có đoạn return this.price * this.quantity

# Hàm tạo

- Hàm tạo là hàm đặc biệt, có tên trùng với tên lớp và không có kiểu dữ liệu trả về
- Hàm tạo được gọi qua khai báo đối tượng
  - final myObject = new MyClass();
  - final myObject = MyClass(); //từ Dart 2, ta có thể bỏ từ khoá new

# Hàm tạo

- Nullable và late

```
class Fraction {
    int? _numerator;
    int? _denominator;

    Fraction(int numerator, int denominator) {
        _numerator = numerator;
        _denominator = denominator;
    }
}
```

Các dữ liệu thành viên _numerator và _denominator khởi tạo
Khắc phục: khai báo late để khởi tạo sau.
- late final: dữ liệu thành viên không thể thay đổi sau đó
- late (không có final): dữ liệu thành viên có thể thay đổi giá trị

```
class Fraction {
    late final int _numerator;
    late final int _denominator;

    Fraction(int numerator, int denominator) {
        _numerator = numerator;
        _denominator = denominator;
    }
}
```

```
class Fraction {
    late int _numerator;
    late int _denominator;

    Fraction(int numerator, int denominator) {
        _numerator = numerator;
        _denominator = denominator;
    }
}
```

# Hàm tạo

- Nullable và Khởi tạo chính tắc:
  - Dart team khuyến cáo nên dùng khởi tạo chính tắc

```dart
class Fraction {
    int _numerator;
    int _denominator;

    Fraction(this._numerator, this._denominator);
}
```

```dart
class Fraction {
    final int _numerator;
    final int _denominator;

    Fraction(this._numerator, this._denominator);
}
```

Khởi tạo chính tắc vẫn có thể truyền giá trị cho dữ liệu thành viên sau đó

```dart
class Fraction {
  final int _numerator;
  final int _denominator;
  late final double _rational;
  Fraction(this._numerator, this._denominator) {
    _rational = _numerator / _denominator;
    doSomethingElse();
  }
}
```

# Danh sách khởi tạo

- Khi sử dụng cách khởi tạo chính tắc, tên của các dữ liệu thành viên phải khớp với tên được khai báo trong hàm tạo. Điều này làm lộ thông tin dữ liệu nội bộ của lớp
- Để đảm bảo bí mật, ta có thể sử dụng danh sách khởi tạo

```
class Test {
    int _secret;
    double _superSecret;

    Test(int age, double wallet)
        : _secret = age,
          _superSecret = wallet;
}
```

# Hàm tạo có tên

- Hàm tạo có tên để khởi tạo một số dữ liệu thành viên cụ thể

```
class Product {
  ///Khai báo các thuộc tính
  String manufacture = '';
  String name = '';
  var    price;
  int    quantity=0;
  //hàm tạo têm iphone để khởi tạo sản phẩm cụ thể
  Product.iphone(var price, {int quantity:0}) {
    this.price       = price;
    this.quantity    = quantity;
    this.manufacture = 'Apple';
  }
  // ...
}
//khởi tạo hàm tạo tên iphone theo cách như sau:
var product = Product.iphone(700, quantity: 2);
```

# Hàm tạo chuyển hướng

- Đôi khi hàm tạo thực hiện tương tự một hàm thành viên khác. Sử dụng hàm tạo chuyển hướng để tránh trùng lặp mã
- Ví dụ: Hàm oneHalf() gọi lại hàm tạo chính tắc để khỏi lặp code

```
Fraction(this._numerator, this._denominator);
// Represents '1/2'
Fraction.oneHalf() : this(1, 2);
// Represents integers, like '3' which is '3/1'
Fraction.whole(int val) : this(val, 1);
```

# Quyền truy cập

- Dart không có các quyền truy cập: public, protected, và private.
- Cùng một file.dart thì mọi thứ là **public**.
- Các tài nguyên thuộc về file.dart khác nếu tên biến có bắt đầu bởi kí tự _ thì được coi là **private** ở các mức độ packages/ files/ libraries.

```dart
class Person {
  // chỉ được truy xuất trong library này.
  final _name;

  Person(this._name);

  String greet(String who) => 'Hello, $who. I am $_name.';
}
```

# Hàm tạo factory

- Từ khóa factory trả về một phiên bản của lớp đã cho mà không phải là một lớp mới. Hữu ích khi:
  - muốn trả về một thể hiện của một lớp con thay vì chính lớp đó,
  - muốn triển khai một Singleton,
  - muốn trả về một thể hiện từ bộ nhớ cache.
- Hàm tạo factory không sử dụng this.

```
class Test {
    static final _objects = List<BigObject>();

    factory Test(BigObject obj) {
        if (!objects.contains(obj))
            objects.add(obj);

        return Test._default();
    }

    Test._default() {
        //do something...
    }
}
```

# Hàm tạo factory

```
class Logger {
  final String name;
  bool mute = false;

  static final Map<String, Logger> _cache = <String, Logger>{};

  factory Logger(String name) {
    return _cache.putIfAbsent(name, () => Logger._internal(name));
  }

  factory Logger.fromJson(Map<String, Object> json) {
    return Logger(json['name'].toString());
  }

  Logger._internal(this.name);

  void log(String msg) {
    if (!mute) print(msg);
  }
}
```

```
void main() {
  var logger = Logger('UI');
  logger.log('Button clicked');

  var logMap = {'name': 'UI'};
  var loggerJson = Logger.fromJson(logMap);
}
```

# Singleton

```dart
class Singleton {
  static final Singleton _singleton = new Singleton._internal();

  factory Singleton() {
    return _singleton;
  }

  Singleton._internal();
}

Run | Debug
main() {
  var s1 = new Singleton();
  var s2 = new Singleton();
  assert(identical(s1, s2));
  assert(s1 == s2);
}
```

# Static

- Thành viên "static" là thành viên của chính lớp đó thay vì trên những đối tượng của lớp.
- Ta có thể truy xuất thành viên static ngay cả khi không có đối tượng nào của lớp đó

```
class Example {
    static const name = "Flutter";
    static String test() => "Hello, I am $name!";
}

void main() {
    final name = Example.name;
    final text = Example.test();
}
```

# Static

- Thành viên static có phạm vi lớp, không phải phạm vi đối tượng
- Không sử dụng this trong các hàm thành viên static

```
class Example {
    int a = 0;
    static void test() {
        // Doesn't compile
        final version = this.a;
        print("$version");
    };
}
```

# Const

- Từ khoá const được dùng cho các giá trị không đổi, kiểu dữ liệu có thể giản lượt
- Từ khoá final được dùng cho nhưng giá trị không đổi nhưng chưa biết lúc biên dịch

```
// type of 'number' is int
const number = 5
// explicitly write the type
const String name = 'Alberto';

const sum = 5.6 + 7.34;
```

```
final contents = File('myFile.txt').readAsString();

// const contents = File('myFile.txt').readAsString();
// ^ does not compile!
```

# Const

- Các dữ liệu thành viên đều có thể là final, nhưng const chỉ được phép khi dữ liệu thành viên là static

```
class Example {
    // OK
    final double a = 0.0;
    // NO, instance variables can only be 'final'
    const double b = 0.0;

    // OK
    static const double PI = 3.14;
    // OK but without type annotation
    static const PI = 3.14;
}
```

# Hàm tạo hằng – const constructor

- Hàm tạo hằng chỉ được sử dụng khi muốn khởi tạo một danh sách các dữ liệu thành viên final

```
// Compiles
class Compiles {
    final int a;
    final int b;
    const Compiles(this.a, this.b);
}

// Does not compile because a is mutable (not final)
class DoesNot {
    int a;
    final int b;
    const DoesNot(this.a, this.b);
}
```

# Lớp bất biến - immutable class

- Lớp bất biến là lớp chỉ có các thành viên final
- Lớp bất biến phải sử dụng hàm tạo hằng

```
// Compiles
class Compiles {
    final int a;
    final int b;
    const Compiles(this.a, this.b);
}

final example1 = const Compiles(); // (1) constant object
final example2 = Compiles();       // (2) not a constant object!
```

# Phương thức - method

- Phương thức là các hàm mà cung cấp các hành vi cho đối tượng

```dart
import 'dart:math';

class Point {
  double x, y;

  Point(this.x, this.y);

  double distanceTo(Point other) {
    var dx = x - other.x;
    var dy = y - other.y;
    return sqrt(dx * dx + dy * dy);
  }
}
```

# Phương thức tĩnh

- Các phương thức (hàm) trong lớp chỉ truy cập được trên một đối tượng cụ thể
- Phương thức tĩnh không cần đối tượng triển khai từ lớp để hoạt động mà có thể gọi hàm đó thông qua tên lớp.

```
class Product {
  // ...
  static showListStore() {
    print('Store 1 ...');
    print('Store 2 ...');
  }
  // ...
}
```

```
Product.showListStore();
```

# Setter và getter

- Khi khai báo dữ liệu thành viên là public thì mọi nơi đều có thể truy xuất, điều này dẫn đến lớp mất đi tính an toàn

```
class Fraction {
    int numerator;
    int denominator;
    Fraction(this.numerator, this.denominator);
}

void main() {
    final frac = Fraction(1, 7);

    // Compilation error, numerator is read-only
    frac.numerator = 0;
    // No problems here, we can read its value
    final num = frac.numerator;
}
```

# Setter và getter

- Trong lớp này, ta chuyển _numerator và _denominator thành private.
- Getter chuyển dữ liệu thành read-only

```
class Fraction {
    int _numerator;
    int _denominator;
    Fraction(this._numerator, this._denominator);

    // Getters are read-only
    int get numerator => _numerator;
    int get denominator {
        return _denominator;
    }
}
```

```
void main() {
    final frac = Fraction(1, 7);

    // Compilation error, numerator is read-only
    frac.numerator = 0;
    // No problems here, we can read its value
    final num = frac.numerator;
}
```

# Setter và getter

- Setter truy xuất thành viên private.
- Setter và getter trùng tên, nhưng thuộc tính khác nhau
- Setter để ghi giá trị an toàn

```
void main() {
    final frac = Fraction(1, 7);

    var den1 = frac.denominator; // den1 = 7
    frac.denominator = 0; // the setter changes it to 1
    den1 = frac.denominator      // den1 = 1
}
```

```
class Fraction {
    int _numerator;
    int _denominator;
    Fraction(this._numerator, this._denominator);

    // getters
    int get numerator => _numerator;
    int get denominator => _denominator;

    // setter
    set denominator(int value) {
        if (value == 0) {
            // Or better, throw an exception...
            _denominator = 1;
        } else {
            _denominator = value;
        }
    }
}
```

# Bad and Good

- Khi dữ liệu là read-only và public, không cần sử dụng getter

```
void main() {
    final frac = Fraction(1, 7);

    var den1 = frac.denominator; // den1 = 7
    frac.denominator = 0; // the setter changes it to 1
    den1 = frac.denominator      // den1 = 1
}
```

# Bad and Good

- Khi dữ liệu không cần kiểm tra hợp lệ, không cần dùng getter và setter

```
class Example {
    var _address = "https://fluttercompletereference.com";

    String get address => _address;
    set address(String value) => _address = value;
}
```

```
class Example {
    var address = "https://fluttercompletereference.com";
}
```

# Quá tải toán tử

```
class Fraction {
    Fraction operator+(Fraction other) =>
        Fraction(
          _numerator * other._denominator +
          _denominator * other._numerator,
          _denominator * other._denominator
        );

    Fraction operator-(Fraction other) => ...

    Fraction operator*(Fraction other) => ...

    Fraction operator/(Fraction other) => ...
}
```

```
// 2/5
final frac1 = Fraction(2, 5);
// 1/3
final frac2 = Fraction(1, 3);

// 2/5 + 1/3 = 11/15
final sum = frac1 + frac2
```

Lưu ý: không quá tải 1 toán tử nhiều hơn 1 lần

# Quá tải toán tử-Các lớp gọi được

- Phương thức call () đặc biệt liên quan đến quá tải toán tử vì nó cho phép các lớp có thể gọi được như là các hàm với toán tử ().
- Phương thức call() có thể có nhiều tham số, cơ thể có kiểu dữ liệu trả về hoặc void

```
class Example {
  double call(double a, double b) => a + b;
}
Run | Debug
void main() {
  final ex = Example(); // 1. Khai báo lớp theo các truyền thống
  final value = ex(1.3, -2.2); // 2. Đối tượng ex có thể như một hàm.
  //Phương thức call () cho phép một đối tượng được coi như một hàm.
  print("$value");
}
```

# Quá tải toán tử-Các lớp gọi được

- Các lớp có phương thức call() được xem là lớp gọi được
- Trong Dart, mọi thứ đều là đối tượng, hàm cũng là đối tượng

```dart
class _Test {
  const _Test();
  void call(String something) {
    print(something);
  }
}

const test = _Test();

Run | Debug
void main() {
  test("Hello");
}
```

# Nhân bản đối tượng

- anotherMe tham chiếu đến me, thay đổi trên me → thay đổi trên anotherMe và ngược lại

```dart
class Person {
  final String name;
  int age;
  Person({
    required this.name,
    required this.age,
  });
  void nhap(int age) {
    this.age = age;
  }
}

Run | Debug
void main() {
  var me = Person(name: "Alberto", age: 25);
  var anotherMe = me;
  print('Name: ${anotherMe.name}, Age: ${anotherMe.age.toString()}');
  anotherMe.age = 20;
  print('Name: ${me.name}, Age: ${me.age.toString()}');
}
```

# Nhân bản đối tượng

- Nhân bản đối tượng với một số dữ liệu khác

```dart
class Person {
  final String name;
  final int age;
  const Person({
    required this.name,
    required this.age,
  });
  Person copyWith({
    String? name,
    int? age,
  }) =>
      Person(name: name ?? this.name, age: age ?? this.age);
  @override
  String toString() => "$name, $age";
}

Run | Debug
void main() {
  const me = const Person(name: "Alberto", age: 25);
  // Tạo một deep copy của 'me' với một name khác
  final anotherMe = me.copyWith(name: "Nguyễn");
  // Tạo một deep copy của 'me' với một age khác
  final futureMe = me.copyWith(age: 35);
  print("$me"); // Alberto, 25
  print("$anotherMe"); // Alberto, 25
  print("$futureMe"); // Alberto, 35
}
```

# Kế thừa

# Kế thừa

- Sử dung từ khóa extends để xây dựng một lớp mới kế thừa từ một lớp lớp đã có
- Lớp con có các thuộc tính và phương thức của lớp cha
- Từ khoá @override là từ chọn, nhưng nên sử dụng khi muốn nạp chồng thành viên lớp cha có cùng tên.

```
class A {
    double test(double a) => a * 0.5;
}

class B extends A {
    @override
    double test(double a) => a * 1.5;
}
```

# Kế thừa

- Khởi tạo lớp cha trước khi khởi tạo lớp con
- Từ khoá super để truy cập thành viên của lớp cha
- Hàm khởi tạo ở lớp con, phải gọi một hàm khởi tạo nào đó của lớp cha. Sau hàm khởi tạo của lớp con, dấu : chỉ rõ hàm tạo nào cả lớp cha sẽ được gọi

```
class Product {
   String manufacture = '';
  String name = '';
  var    price;
  late int    quantity;

  Product(var price, {int quantity:0}) {
    this.price    = price;
    this.quantity = quantity;
  }
}

class Table extends Product {
  double length = 0;
  double width = 0;
  Table(var price) : super(price, quantity:1) {
    this.name = "Bàn Ăn";
  }
}
```

# Super và hàm tạo

- Mọi lớp con trong Dart tự động cố gắng gọi hàm tạo mặc định của lớp cha.
- Nếu lớp cha không có hàm tạo mặc định, phải gọi phương thức khởi tạo siêu lớp theo cách thủ công trong danh sách khởi tạo

```
class Example { //lớp cha
    int a;
    Example(this.a);
}
class SubExample extends Example { //lớp con
    int b;
  // nếu không gọi 'super(b)',
  //lỗi biên dịch vì lớp cha không có hàm tạo mặc định
  SubExample(this.b) : super(b);
}
```

# Super và hàm tạo

- Lưu ý:
  - Nếu lớp cha có hàm tạo mặc định thì super là không cần thiết
  - Lời gọi super() phải để cuối danh sách khởi tạo thành viên

```
// Compiles
MyClass(int a) : _a = a, super(a*a);

// Doesn't compile
MyClass(int a) : super(a*a), _a = a;
```

# Ghi đè phương thức

- Khi lớp con định nghĩa lại phương thức của lớp cha thì tiến hành override ghi đè (nạp chồng) phương thức.
- Việc ghi đè được thực hiện để một lớp con có thể đưa ra triển khai riêng của nó cho phương thức đã được cung cấp bởi lớp cha.

```
class Product {
   String manufacture = '';
  String name = '';
  var    price;
  late int    quantity;
  Product(var price, {int quantity:0}) {
    this.price    = price;
    this.quantity = quantity;
  }
  calulateTotal() {
    return this.price * this.quantity;
  }
  showTotal() {
    var tong = this.calulateTotal();
    print("Tổng số tiền là: $tong");
  }
}
class Table extends Product {
  double length = 0;
  double width = 0;
  Table(var price) : super(price, quantity:1) {
    this.name = "Bàn Ăn";
  }
  @override
  showTotal() {
      print('Sản phẩm:' + this.name);
      super.showTotal();
    }
}
```

# Ghi đè toán tử

- Cách thức thực hiện giống như ghi đè phương thức
- Tên phương thức được thay bang từ khoá operator ký_hiệu_toán tử
- Ví dụ:

```
class Table extends Product {
  double length = 0;
  double width = 0;
  Table(var price) : super(price, quantity: 1) {
    this.name = "Bàn Ăn";
  }
  @override
  showTotal() {
    print('Sản phẩm:' + this.name);
    super.showTotal();
  }

  Product operator + (Product p) => new Product(this.quantity + p.qua
}
```

# Lớp trừu tượng

- Từ khóa trừu tượng (abstract) định nghĩa một lớp cơ sở khởi tạo trực tiếp
- Các lớp dẫn xuất phải định nghĩa – khởi tạo nội dung này
- Một lớp trừu tượng có thể định nghĩa một (hoặc nhiều) hàm tạo
- Một lớp có ít nhất 1 phương thức không có thân hàm thì phải được khai báo là abstract

```
abstract class Example {
    void method();
}

class ExampleTwo extends Example {
    @override
    void method() {
        print("I'm not abstract!");
    }
}
```

# Lớp giao diện - Interface

- Interface là 1 tập các thành phần chỉ có khai báo mà không có phần định nghĩa
- Một interface được hiểu như là 1 khuôn mẫu mà mọi lớp thực thi nó đều phải tuân theo.
- Interface sẽ định nghĩa phần “làm gì” (khai báo) và những lớp thực thi interface này sẽ định nghĩa phần “làm như thế nào” (định nghĩa nội dung) tương ứng.
- Dart không có từ khoá interface, lớp trừu tượng muốn được triển khai bởi lớp dẫn xuất bằng từ khóa implements
- Khi một lớp được coi là giao diện thì lớp triển khai nó phải định nghĩa lại mọi phương thức, thuộc tính có trong giao diện
- Mục đích sử dụng giao diện là để đảm báo các lớp có cùng giao diện sẽ có các API giống nhau.

# Lớp giao diện - Interface

- Asbstract và không abstract

```
abstract class OneInterface {
    void one();
}

abstract class TwoInterface {
    void two();
}

class Example implements OneInterface, TwoInterface {
    @override
    void one() {}

    @override
    void two() {}
}
```

```
class OneInterface {
    void one() {}
}

class TwoInterface {
    void two() {}
}

class Example implements OneInterface, TwoInterface {
    @override
    void one() {}

    @override
    void two() {}
}
```

# extends và implements

- extension dành cho các lớp con và implement dành cho các lớp giao diện.
  - Khi dùng class B extends A {}, KHÔNG bị buộc phải ghi đè mọi phương thức của lớp A. Sự kế thừa diễn ra và có thể ghi đè bao nhiêu phương thức tùy thích.
  - Khi dùng class B implements A {}, buộc phải ghi đè mọi phương thức của lớp A. Việc kế thừa KHÔNG diễn ra bởi vì các phương thức chỉ cung cấp một API, một "khung" mà lớp con phải cụ thể hóa.
- Sử dụng thực tế:
  - Extend: khi bạn muốn thêm một số tính năng còn thiếu trong một lớp con.
  - Implement: khi không muốn cung cấp triển khai các chức năng mà chỉ cung cấp API.

# Mixin

- Mixin là một lớp, không được sử dụng trực tiếp để tạo ra đối tượng
- Mixin chứa các phương thức, thuộc tính dùng để gộp vào một lớp khác.
- Ví dụ: lớp C gộp mixin M vào trong nó

```
mixin M {
  var var1 = null;
  showSomething()
  {
    print('Print message ...');
  }
}

class C extends B with M {
  @override
  String name;

  @override
  void displayInfomation() {
   }
}
```

# Lớp Object

- Lớp Object là lớp gốc của mọi kiến trúc
- Mọi lớp đều có thể và nên ghi đè các phương thức được khai báo trong lớp Object.
- Các phương thức phổ biến trong lớp Object:
  - String toString().
  - bool operator ==(SomeClass other)

# Lớp Object

```dart
class Example {
  int a;
  Example(this.a);
  @override
  bool operator ==(Object other) {
// Hàm identical() được Dart code API cung cấp,
// kiểm tra 2 đối tượng có cùng tham chiếu
    if (identical(this, other)) return true;
// Nếu đối tượng so sánh cùng kiểu thì tiến hành so sánh
    if (other is Example) {
      final example = other;
// Nếu dùng runtimeType thì tiến hành so sánh
      return runtimeType == example.runtimeType && a == example.a;
    } else {
      return false;
    }
  }
// Ghi đè toán tử = phải ghi đè hashCode.
// hashCode hữu ích khi gọi identical() hoặc khi dùng hash maps (giới thiệu sau)
  @override
  int get hashCode => a.hashCode;
}
Run | Debug
void main() {
  final ex1 = Example(2);
  final ex2 = Example(2);
  print(ex1 == ex2); // true -> that's what we wanted!
}
```

# Ngoại lệ

# Khái niệm

- Dart có thể đưa ra các ngoại lệ để báo hiệu hành vi không mong muốn hoặc có lỗi đã xảy ra trong quá trình thực thi.
- Khi ném một ngoại lệ (throw exception), ta nên bắt nó, nếu không chương trình sẽ buộc phải kết thúc bằng mã lỗi.

```
class Fraction {
  int _numerator;
  int _denominator;
  Fraction(this._numerator, this._denominator) {
    if (_denominator == 0) {
      throw IntegerDivisionByZeroException();
    }
  }
}
```

# Try on và catch

- Nếu bắt ngoại lệ để chương trình không bị forcefully terminate
- Nếu bạn muốn lấy một phiên bản của ngoại lệ đã ném, chỉ cần thêm một câu lệnh catch:

```
void main() {
    try {
        final f = Fraction(1, 0);
    } on IntegerDivisionByZeroException {
        print("Ouch! Division by zero!");
    }
}
```

```
void main() {
    try {
        final f = Fraction(1, 0);
    } on IntegerDivisionByZeroException catch (exc) {
        // use the exc object
        doSomething(exc);

        print("Ouch! Division by zero!");
    }
}
```

# Try on và catch

- Tổng quát

```
void main() {
    try {
        final f = Fraction(1, 0);
    } on IntegerDivisionByZeroException {
        print("Division by zero!");
    } on FormatException {
        print("Invalid format!");
    } catch (e) {
        // You arrive here if the throum exception is neither
        // IntegerDivisionByZeroException or FormatException
        print("General error: $e");
    }
}
```

# Finally

- Để đảm bảo rằng một số mã luôn chạy cho dù ngoại lệ có được ném ra hay không
- Nếu không có mệnh đề bắt nào khớp với ngoại lệ, thì ngoại lệ được truyền sau khi mệnh đề finally sẽ chạy:

```
void main() {
    try {
        final f = Fraction(2, 3);
    } catch (e) {
        print("Error");
    } finally {
        print("Always here");
    }

    print("Finish");
}
```

# Generics and Collections

# Generics

# Giới thiệu

- Dart, Java, C# và Delphi nổi tiếng là những ngôn ngữ hỗ trợ lập trình chung – generic - tham số hoá
- Generic là một khái niệm để sử dụng các kiểu dữ liệu khác nhau với một định nghĩa duy nhất.
- Ví dụ: tạo các kiểu danh sách khác nhau trong Dart như List với List có thể là string, int, double...
- Trong generic dùng các chữ viết tắt E (element), T (data type), K (key), R (return type) và V (value).

# Giới thiệu

- Generics thường được yêu cầu cho các kiểu đòi hỏi sự an toàn, tuy nhiên, generric hữu ích khi:
  - Chỉ định đúng các kiểu chung dẫn đến mã được tạo tốt hơn.
  - Có thể sử dụng generic để giảm trùng lặp mã.
- Nếu muốn cho một List chỉ chứa các String, có thể khai báo List<String>(đọc là List của Chuỗi). Cách này có thể bị lỗi nếu gán một object không phải kiểu String vào List,ví dụ:

```
✗ static analysis: error/warning
  var names = List<String>();
  names.addAll(['Seth', 'Kathy', 'Lars']);
  names.add(42); // Error
```

# Kiểu an toàn

- Có thể dùng dynamic để cung cấp 1 lớp có dữ liệu chung, nhưng vấn đề của dynamic là kém an toàn vì được thiết kế để làm việc với chuyển kiểu runtime
- Với generic, chuyển kiểu là không cần thiết vì trình biên dịch đảm bảo sự bảo vệ với việc sử dụng kiểu dữ liệu sai

```
class Cache {
  dynamic _obj;
  dynamic get value => _obj;
  Cache(dynamic value) {
    this._obj = value;
  }
}

Run | Debug
void main() {
  final cache = Cache(20);
  String value = cache.value;
  print("$value");
}
```

```
class Cache<T> {
  late T _obj;
  T get value => _obj;
  Cache(T value) {
    this._obj = value;
  }
}

Run | Debug
void main() {
  final cache = Cache(20);
  String value = cache.value;
  print("$value");
}
```

dynamic lúc dịch mới báo lỗi, generic báo lỗi trước khi dịch

# Tránh trùng lặp mã

- Ví dụ: tránh trùng lặp mã khi sử dụng

```
abstract class ObjectCache {
    Object getByKey(String key);
    void setByKey(String key, Object value);
}
// thiết lập lớp cho chuỗi
abstract class StringCache {
    String getByKey(String key);
    void setByKey(String key, String value);
}
// thiết lập lớp cho số ...
abstract class IntCache {
    String getByKey(String key);
    void setByKey(String key, int value);
}
```

```
//Tạo giao diện duy nhất có tham số kiểu
abstract class Cache<T> {
  T getByKey(String key);
  void setByKey(String key, T value);
}
```

# Hàm và lớp generic

- Có thể xây dựng các lớp, các hàm mà khi khai báo thì nó không làm việc trên một kiểu dữ liệu cụ thể mà là một kiểu dữ liệu chung, ký hiệu là kiểu dữ liệu A, B, E, T ...
- Khi triển khai lớp thành đối tượng mới chỉ định các kiểu dữ liệu ấy
- Khai báo ký hiệu đại diện trong cặp ngoặc nhọn <…>

```
class MyClass<E, T> {

}
fncGeneric<T>(T thamso) {
  print(thamso);
}
```

```
var a = new MyClass<int, String>();
//Ký hiệu E sẽ là kiểu int
//Kỹ hiệu T sẽ là kiểu String


var b = new MyClass<double, int>();
//Ký hiệu E sẽ là kiểu double
//Kỹ hiệu T sẽ là kiểu int
```

# Hàm và lớp generic

- Ví dụ

```
class MyClass<E,T>
{
  E thuoctinh1;
  T thuoctinh2;
  setThuoctinh(E t1, T t2) {
    this.thuoctinh1 = t1;
    this.thuoctinh2 = t2;
  }
  show() {
    print(this.thuoctinh1);
    print(this.thuoctinh2);
  }
}

fncGeneric<T>(T thamso) {
  print(thamso);
}
```

```
var a = new MyClass<int, String>();
a.setThuoctinh(11, 'Lop Generic');
a.show();
fncGeneric<int>(111);
```

# Collections

# List

- Dạng thức:

  List tên_list = new List(số phần tử);
  var/dynamic tên_list = new List(số phần tử);

- Kiểu dữ liệu hoặc số phần tử có thể chưa cần khai báo trước