Mobile Programming with Flutter

เริ่มพัฒนาแอปพลิเคชันบนมือถือด้วย Flutter

⭐ ทำความเข้าใจกับพื้นฐานการเขียน Flutter

“จริง ๆ แล้ว Flutter ไม่ใช่ภาษาโปรแกรม
แต่เป็น Framework ที่ไว้ใช้ทำการ
สร้าง User Interface”

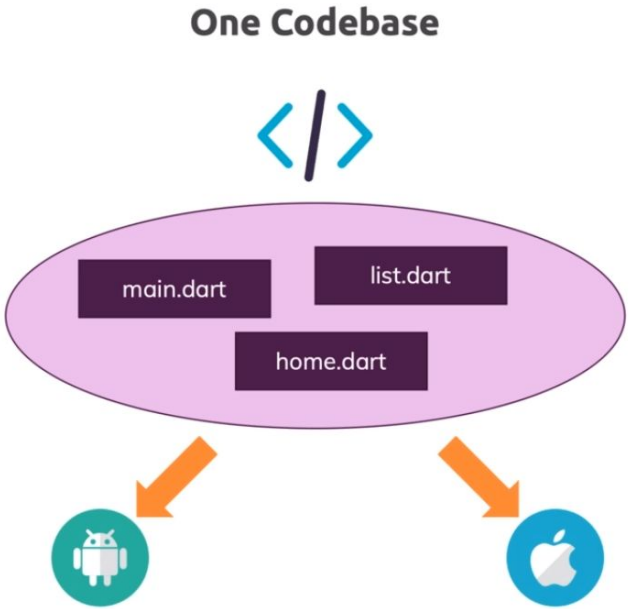
“จริง ๆ แล้ว Flutter ไม่ใช่ภาษาโปรแกรม
แต่เป็น Framework ที่ไว้ใช้ทำการ
สร้าง User Interface”

“ภายในจะประกอบไปด้วย Package และ
Function ที่จำเป็นต่าง ๆ ในการเขียน code”

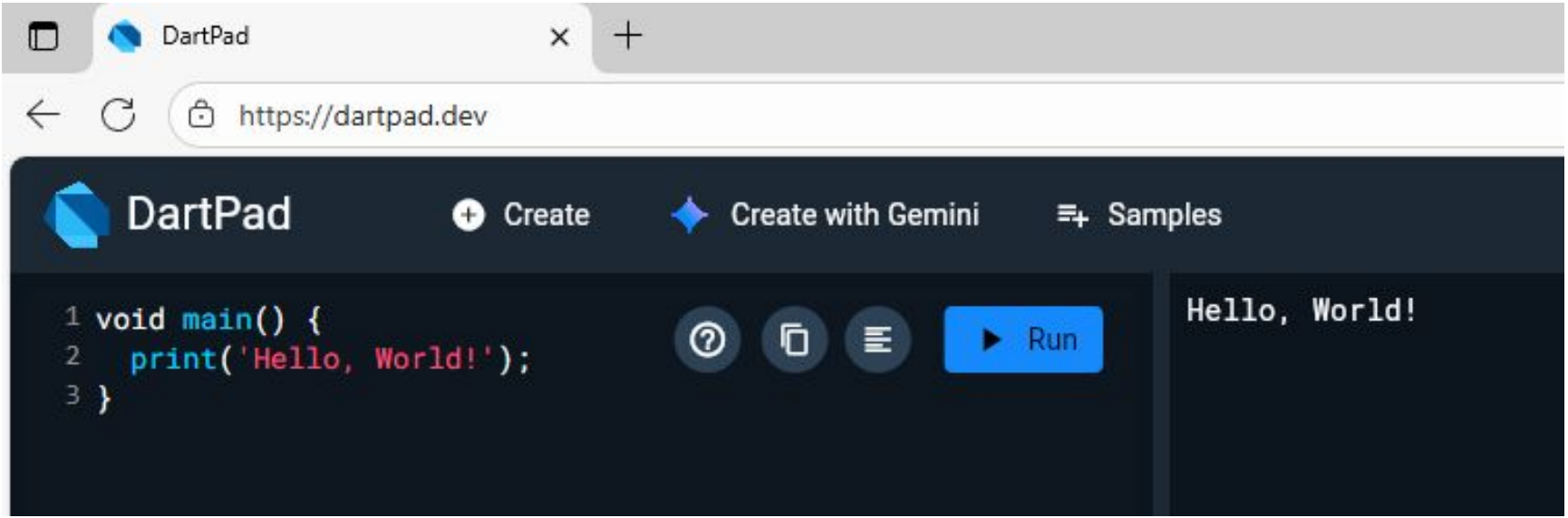
และ ถ้าถามว่า แล้วถ้าอยากเขียน App Flutter ต้องใช้ภาษาอะไรเขียน ?

Dart

Dart



⭐ มาเริ่มเขียน Dart ไปพร้อมกัน



main() = จุดเริ่มรันโปรแกรม

print() = สั่งแสดงข้อความใน console

Variables

การประกาศตัวแปร

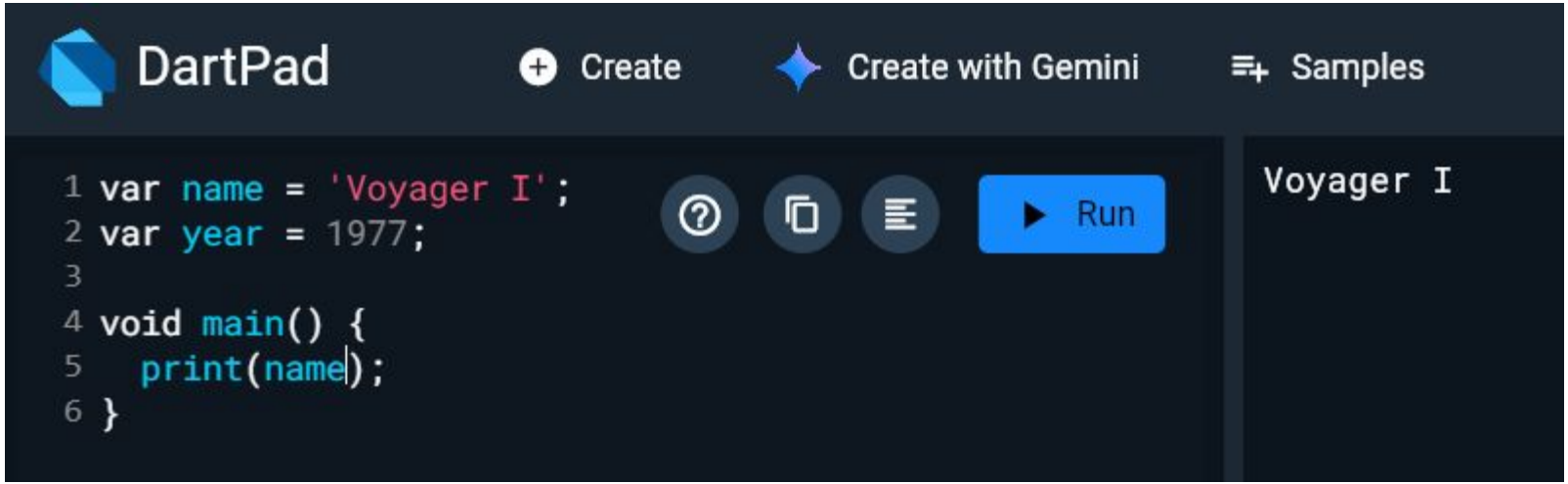
```
1 var name = 'Voyager I';
2 var year = 1977;
3 var flybyObjects = ['Jupiter', 'Saturn'];
4
```

Data Types & Built-in Functions

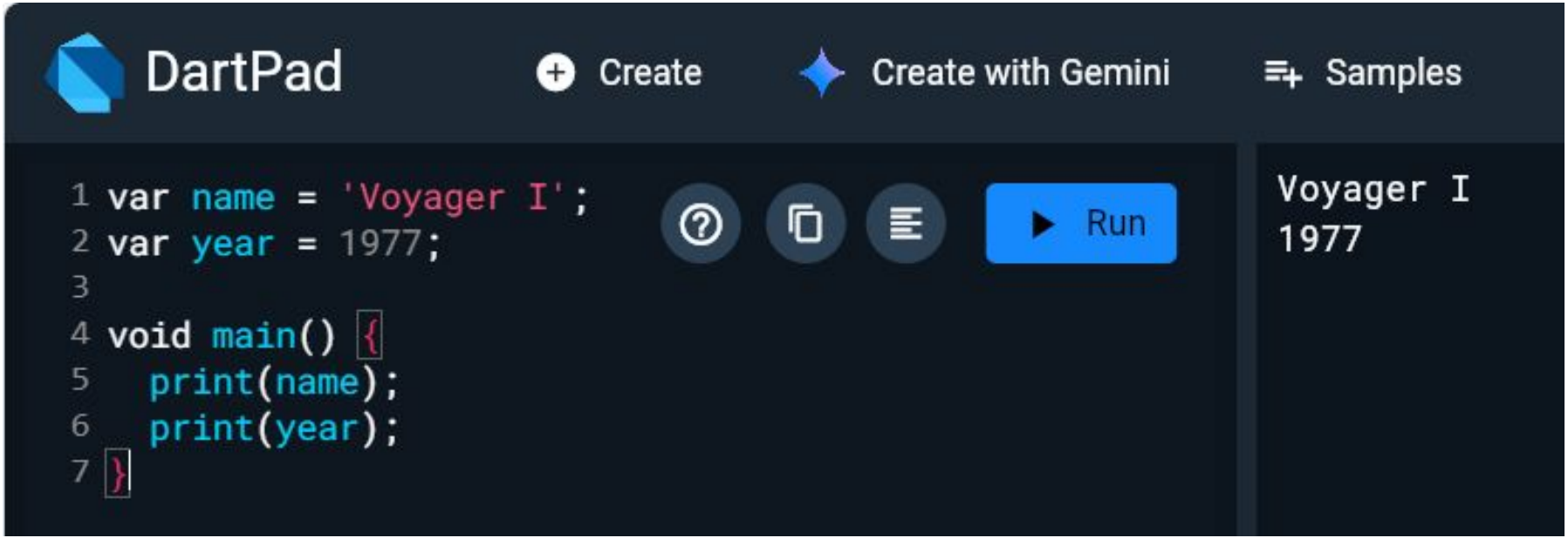
ใช้ var ให้ Dart เดา type ให้

Dart เป็น type-safe แต่ไม่ต้องประกาศ type ก็ได้

Print ตัวแปรพื้นฐาน (Beginner)

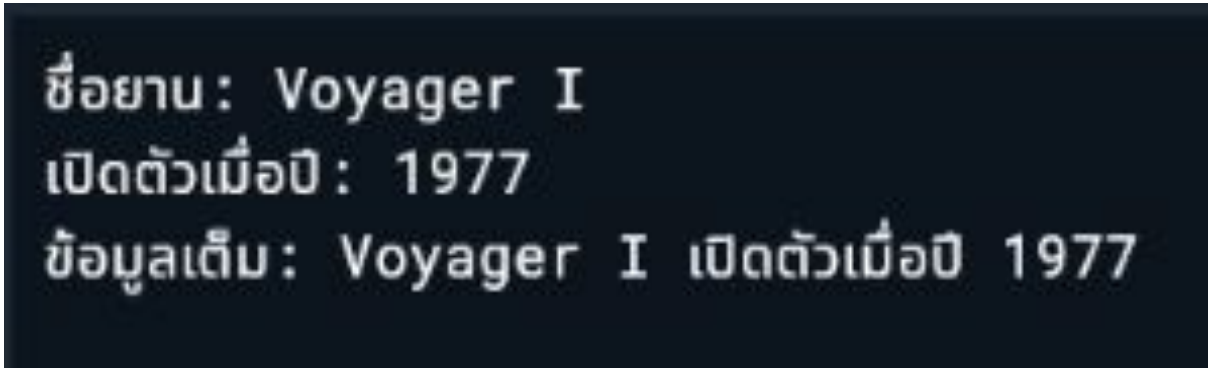


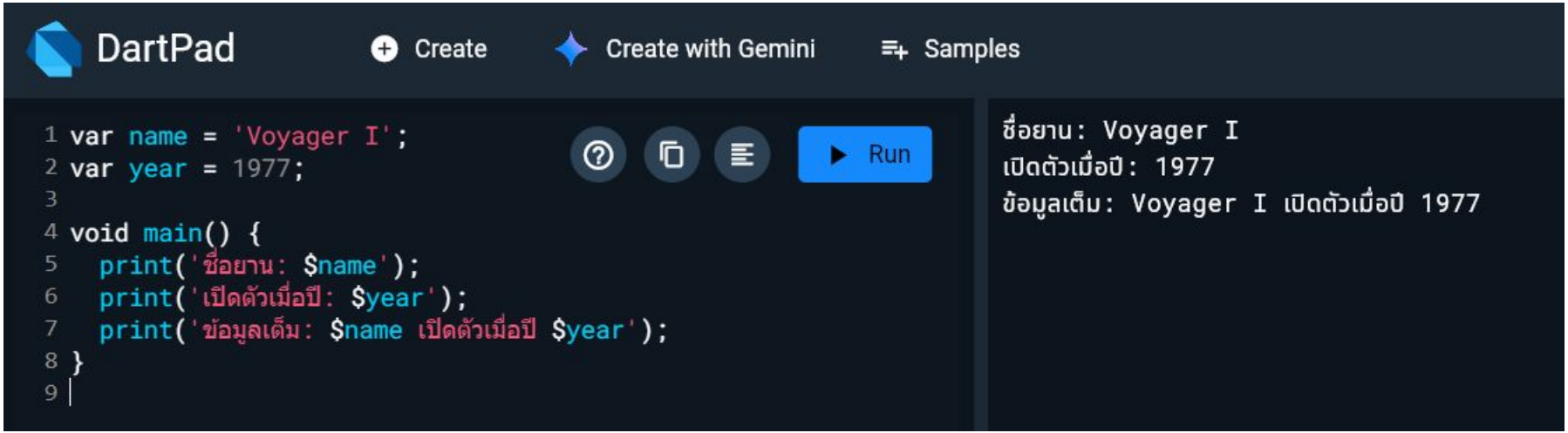
Quiz : แสดงค่าในตัวแปรที่เหลือ



Print แบบ String Interpolation

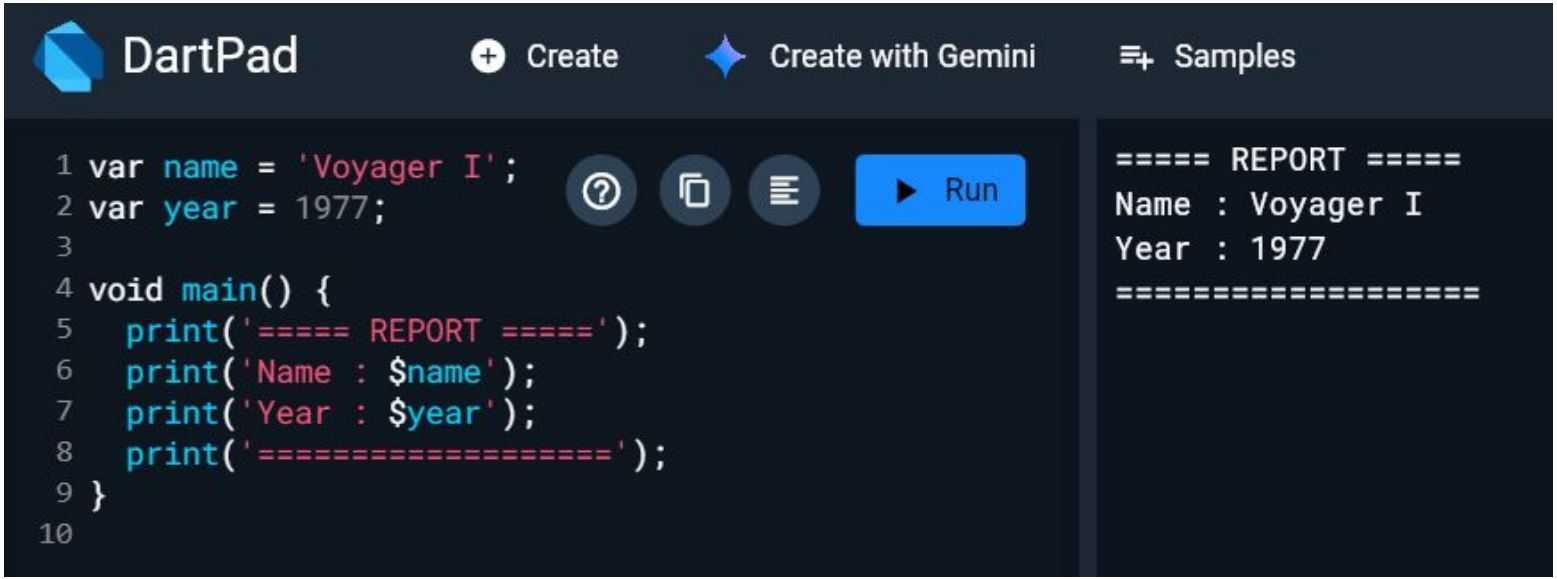
Quiz : แสดงข้อความที่มีตัวแปรซ่อนอยู่ใน String





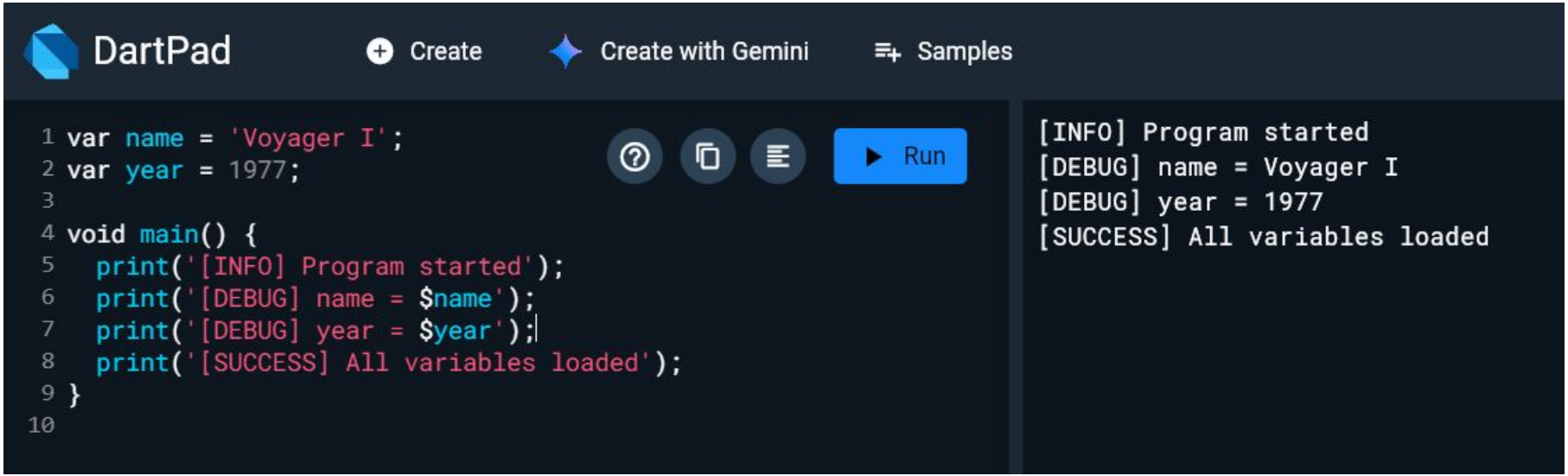
Quiz : Print แบบจัดรูปข้อความ (Formatting)

ลองทำให้มันแสดงออกมาแบบนี้หน่อย

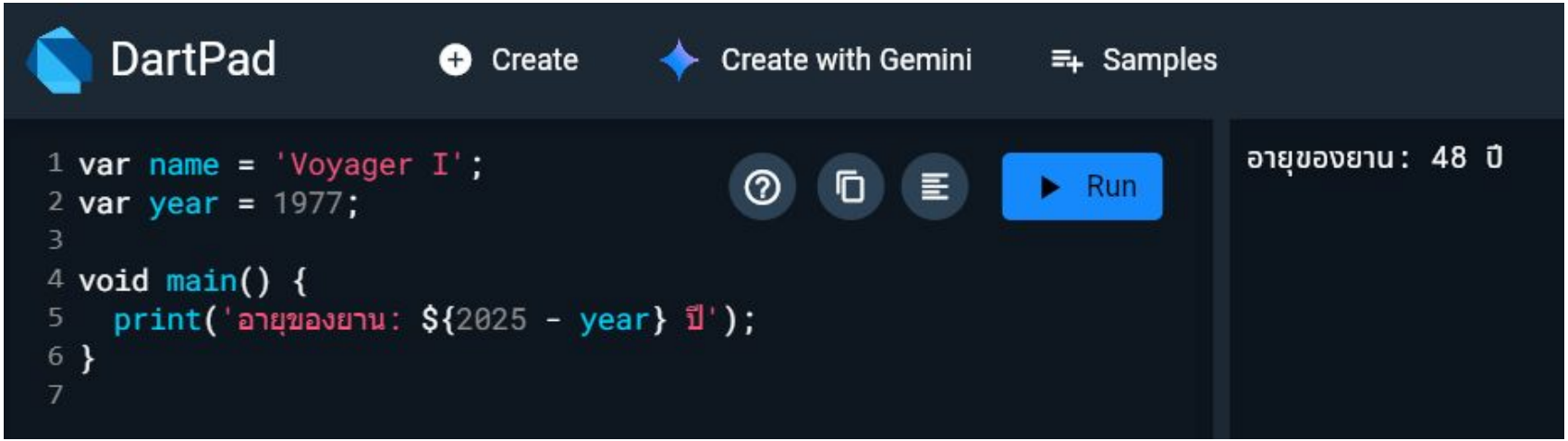


Quiz : ให้สร้างข้อความ debug

```
[INFO] Program started
[DEBUG] name = Voyager I
[DEBUG] year = 1977
[SUCCESS] All variables loaded
```



Print แบบ Expression



print การคำนวณทันที

การประกาศตัวแปรใน Dart

Here's an example of creating a variable and initializing it:

```
var name = 'Bob';
```

Variables store references. The variable called `name` contains a reference to a `String` object with a value of "Bob".

The type of the `name` variable is inferred to be `String`, but you can change that type by specifying it. If an object isn't restricted to a single type, specify the `Object` type (or `dynamic` if necessary).

```
Object name = 'Bob';
```

Another option is to explicitly declare the type that would be inferred:

```
String name = 'Bob';
```

Null safety

Null safety prevents an error that results from unintentional access of variables set to `null`. The error is called a null dereference error. A null dereference error occurs when you access a property or call a method on an expression that evaluates to `null`. An exception to this rule is when `null` supports the property or method, like `toString()` or `hashCode`. With null safety, the Dart compiler detects these potential errors at compile time.

For example, say you want to find the absolute value of an `int` variable `i`. If `i` is `null`, calling `i.abs()` causes a null dereference error. In other languages, trying this could lead to a runtime error, but Dart's compiler prohibits these actions. Therefore, Dart apps can't cause runtime errors.

Null safety introduces three key changes:

1. When you specify a type for a variable, parameter, or another relevant component, you can control whether the type allows `null`. To enable nullability, you add a `?` to the end of the type declaration.

```
String? name  // Nullable type. Can be `null` or string.

String name   // Non-nullable type. Cannot be `null` but can be string.
```

Null safety

2. You must initialize variables before using them. Nullable variables default to `null`, so they are initialized by default. Dart doesn't set initial values to non-nullable types. It forces you to set an initial value. Dart doesn't allow you to observe an uninitialized variable. This prevents you from accessing properties or calling methods where the receiver's type can be `null` but `null` doesn't support the method or property used.

3. You can't access properties or call methods on an expression with a nullable type. The same exception applies where it's a property or method that `null` supports like `hashCode` or `toString()`.

Sound null safety changes potential **runtime errors** into **edit-time** analysis errors. Null safety flags a non-null variable when it has been either:

- Not initialized with a non-null value.
- Assigned a `null` value.

This check allows you to fix these errors *before* deploying your app.

Default value

Uninitialized variables that have a nullable type have an initial value of `null`. Even variables with numeric types are initially null, because numbers—like everything else in Dart—are objects.

```
int? lineCount;
assert(lineCount == null);
```

Note: Production code ignores the `assert()` call. During development, on the other hand, `assert(condition)` throws an exception if *condition* is false. For details, check out Assert.

Default value

With null safety, you must initialize the values of non-nullable variables before you use them:

```
int lineCount = 0;
```

You don't have to initialize a local variable where it's declared, but you do need to assign it a value before it's used. For example, the following code is valid because Dart can detect that `lineCount` is non-null by the time it's passed to `print()`:

```
int lineCount;

if (weLikeToCount) {
  lineCount = countLines();
} else {
  lineCount = 0;
}

print(lineCount);
```

Top-level and class variables are lazily initialized; the initialization code runs the first time the variable is used.

Default value

```
int lineCount;

if (weLikeToCount) {
  lineCount = countLines();
} else {
  lineCount = 0;
}

print(lineCount);
```

Late variable

Late variables

The `late` modifier has two use cases:

- Declaring a non-nullable variable that's initialized after its declaration.
- Lazily initializing a variable.

Often Dart's control flow analysis can detect when a non-nullable variable is set to a non-null value before it's used, but sometimes analysis fails. Two common cases are top-level variables and instance variables: Dart often can't determine whether they're set, so it doesn't try.

If you're sure that a variable is set before it's used, but Dart disagrees, you can fix the error by marking the variable as `late`:

```
late String description;

void main() {
  description = 'Feijoada!';
  print(description);
}
```

Late variable

⚠ If you fail to initialize a `late` variable, a runtime error occurs when the variable is used.

When you mark a variable as `late` but initialize it at its declaration, then the initializer runs the first time the variable is used. This lazy initialization is handy in a couple of cases:

- The variable might not be needed, and initializing it is costly.
- You're initializing an instance variable, and its initializer needs access to `this`.

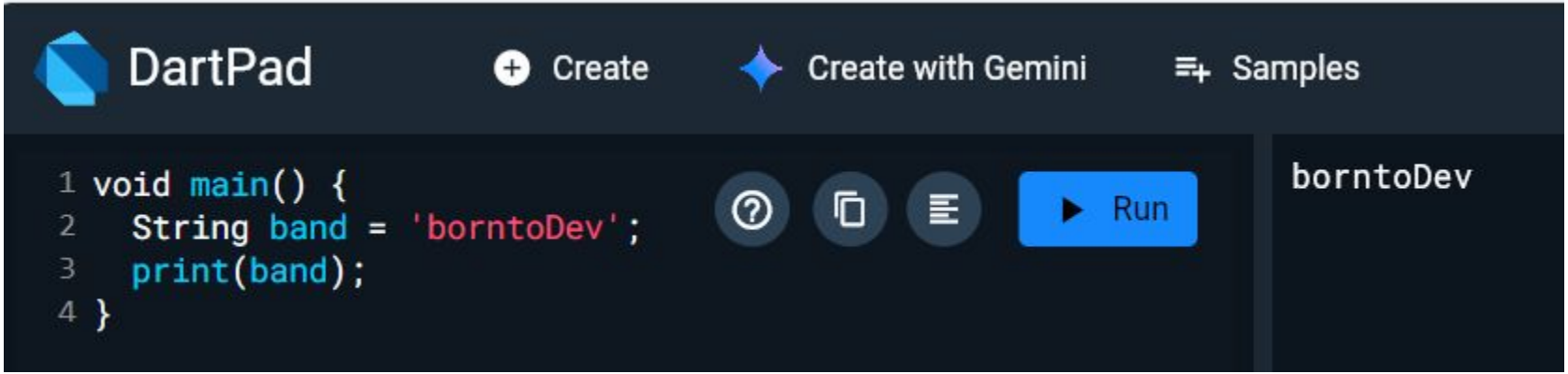
In the following example, if the `temperature` variable is never used, then the expensive `readThermometer()` function is never called:

```
// This is the program's only call to readThermometer().
late String temperature = readThermometer(); // Lazily initialized.
```

main.dart

```dart
void main() {
    //Integer value
    int num1 = 1;
    //Declare a double value
    double num2 = 1.5;
    print(num1);
    print(num2);
}
```

การใช้งานข้อมูลประเภทตัวเลข



การใช้งานข้อมูลประเภทข้อความ

การใช้งานข้อมูลประเภทตรรกะ

dart:core library

CLASSES
BigInt
bool
Comparable
DateTime
Deprecated
double
Duration
Enum
Expando
Finalizer
Function
Future
int
Invocation
Iterable
Iterator
List
Map
MapEntry
Match
Null
num
Object
Pattern
pragma
Record
RegExp
RegExpMatch
RuneIterator
Runes
Set
Sink
StackTrace
Stopwatch

List<E> class abstract

An indexable collection of objects with a length.

Subclasses of this class implement different kinds of lists. The most common kinds of lists are:

- **Fixed-length list**
  An error occurs when attempting to use operations that can change the length of the list.
- **Growable list**
  Full implementation of the API defined in this class.

The default growable list, as created by `[]`, keeps an internal buffer, and grows that buffer when necessary. This guarantees that a sequence of add operations will each execute in amortized constant time. Setting the length directly may take time proportional to the new length, and may change the internal capacity so that a following add operation will need to immediately increase the buffer capacity. Other list implementations may have different performance behavior.

Example of fixed-length list:

```
final fixedLengthList = List<int>.filled(5, 0); // Creates fixed-length list.
print(fixedLengthList); // [0, 0, 0, 0, 0]
fixedLengthList[0] = 87;
fixedLengthList.setAll(1, [1, 2, 3]);
print(fixedLengthList); // [87, 1, 2, 3, 0]
// Fixed length list length can't be changed or increased
fixedLengthList.length = 0;  // Throws
fixedLengthList.add(499);    // Throws
```

Example of growable list:

```
final growableList = <String>['A', 'B']; // Creates growable list.
```

To add data to the growable list, use operator[]=, add or addAll.

```
growableList[0] = 'G';
print(growableList); // [G, B]
growableList.add('X');
growableList.addAll({'C', 'B'});
print(growableList); // [G, B, X, C, B]
```

To check whether, and where, the element is in the list, use indexOf or lastIndexOf.

```
final indexA = growableList.indexOf('A'); // -1 (not in the list)
final firstIndexB = growableList.indexOf('B'); // 1
final lastIndexB = growableList.lastIndexOf('B'); // 4
```

To remove an element from the growable list, use remove, removeAt, removeLast, removeRange

Example of fixed-length list:

```dart
final fixedLengthList = List<int>.filled(5, 0); // Creates fixed-length list.
print(fixedLengthList); // [0, 0, 0, 0, 0]
fixedLengthList[0] = 87;
fixedLengthList.setAll(1, [1, 2, 3]);
print(fixedLengthList); // [87, 1, 2, 3, 0]
// Fixed length list length can't be changed or increased
fixedLengthList.length = 0;  // Throws
fixedLengthList.add(499);    // Throws
```

Example of growable list:

```dart
final growableList = <String>['A', 'B']; // Creates growable list.
```

To add data to the growable list, use operator[]=, add or addAll.

```dart
growableList[0] = 'G';
print(growableList); // [G, B]
growableList.add('X');
growableList.addAll({'C', 'B'});
print(growableList); // [G, B, X, C, B]
```

การใช้งานข้อมูล List

```
void main() {
  var scores = [1, 3, 4, 2];
  scores.remove(1);
  print(scores);
}
```

การใช้งานข้อมูล List

```dart
void main() {
  var scores = [1, 3, 4, 2, 5];
  print('Length: ${scores.length}');
}
```

การใช้งานข้อมูล List

```
void main() {
  var scores = [1, 3, 4, 2, 5];
  //
  print('First: ${scores.first}');
  print('Last: ${scores.last}');
}
```

การใช้งานข้อมูล List

```dart
void main() {
  var scores = [];
  print(scores.isEmpty); // true
  print(scores.isNotEmpty); // false
}
```

การใช้งานข้อมูล List

const = ตัวแปรที่กำหนดค่าได้ครั้งเดียว บังคับกำหนดค่าเริ่มต้น แก้ไขไม่ได้

Static = ใช้แชร์ตัวแปรภายใน Class เดียวกัน

Final = ใช้กำหนดค่าได้ครั้งเดียว ต้องมีค่าเริ่มต้นเสมอ และ มีการกำหนดค่าให้ไม่สามารถแก้ได้ แต่ถ้าเป็น Ref/Obj Data type จะสามารถแก้ไขค่าได้ แต่ไม่สามารถกำหนดค่าตัวแปรใหม่ได้

Final and const

If you never intend to change a variable, use `final` or `const`, either instead of `var` or in addition to a type. A final variable can be set only once; a const variable is a compile-time constant. (Const variables are implicitly final.)

> **Note:** Instance variables can be `final` but not `const`.

Here's an example of creating and setting a `final` variable:

```
final name = 'Bob'; // Without a type annotation
final String nickname = 'Bobby';
```

You can't change the value of a `final` variable:

```
✗ static analysis: error/warning
name = 'Alice'; // Error: a final variable can only be set once.
```

Final and const

Use `const` for variables that you want to be **compile-time constants**. If the const variable is at the class level, mark it `static const`. Where you declare the variable, set the value to a compile-time constant such as a number or string literal, a const variable, or the result of an arithmetic operation on constant numbers:

```
const bar = 1000000; // Unit of pressure (dynes/cm2)
const double atm = 1.01325 * bar; // Standard atmosphere
```

The `const` keyword isn't just for declaring constant variables. You can also use it to create constant *values*, as well as to declare constructors that *create* constant values. Any variable can have a constant value.

```
var foo = const [];
final bar = const [];
const baz = []; // Equivalent to `const []`
```

Final and const

You can omit `const` from the initializing expression of a `const` declaration, like for `baz` above. For details, see DON'T use const redundantly.

You can change the value of a non-final, non-const variable, even if it used to have a `const` value:

```
foo = [1, 2, 3]; // Was const []
```

You can't change the value of a `const` variable:

```
✗ static analysis: error/warning
baz = [42]; // Error: Constant variables can't be assigned a value.
```

You can define constants that use type checks and casts (`is` and `as`), collection `if`, and spread operators (`...` and `...?`):

```
const Object i = 3; // Where i is a const Object with an int value...
const list = [i as int]; // Use a typecast.
const map = {if (i is int) i: 'int'}; // Use is and collection if.
const set = {if (list is List<int>) ...list}; // ...and a spread.
```

ℹ **Note:** Although a `final` object cannot be modified, its fields can be changed. In comparison, a `const` object and its fields cannot be changed: they're *immutable*.

For more information on using `const` to create constant values, see Lists, Maps, and Classes.

To check whether, and where, the element is in the list, use indexOf or lastIndexOf.

```
final indexA = growableList.indexOf('A'); // -1 (not in the list)
final firstIndexB = growableList.indexOf('B'); // 1
final lastIndexB = growableList.lastIndexOf('B'); // 4
```

To remove an element from the growable list, use remove, removeAt, removeLast, removeRange or removeWhere.

```
growableList.remove('C');
growableList.removeLast();
print(growableList); // [G, B, X]
```

To insert an element at position in the list, use insert or insertAll.

```
growableList.insert(1, 'New');
print(growableList); // [G, New, B, X]
```

การใช้งานข้อมูล List

To replace a range of elements in the list, use fillRange, replaceRange or setRange.

```
growableList.replaceRange(0, 2, ['AB', 'A']);
print(growableList); // [AB, A, B, X]
growableList.fillRange(2, 4, 'F');
print(growableList); // [AB, A, F, F]
```

To sort the elements of the list, use sort.

```
growableList.sort((a, b) => a.compareTo(b));
print(growableList); // [A, AB, F, F]
```

To shuffle the elements of this list randomly, use shuffle.

```
growableList.shuffle();
print(growableList); // e.g. [AB, F, A, F]
```

To find the first element satisfying some predicate, or give a default value if none do, use firstWhere.

```
bool isVowel(String char) => char.length == 1 && "AEIOU".contains(char);
final firstVowel = growableList.firstWhere(isVowel, orElse: () => ''); // ''
```

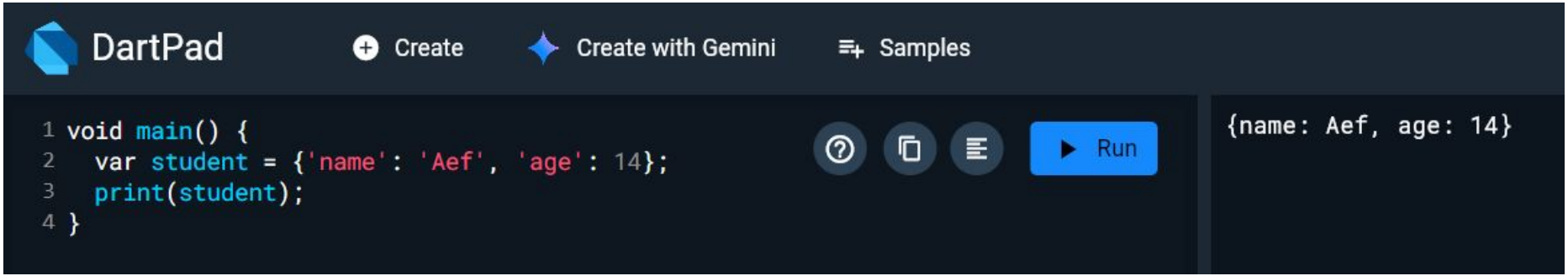
There are similar lastWhere and singleWhere methods.

A list is an Iterable and supports all its methods, including where, map, whereType and toList.

Lists are Iterable. Iteration occurs over values in index order. Changing the values does not affect iteration, but changing the valid indices—that is, changing the list's length—between iteration steps causes a ConcurrentModificationError. This means that only growable lists can throw ConcurrentModificationError. If the length changes temporarily and is restored before continuing the iteration, the iterator might not detect it.

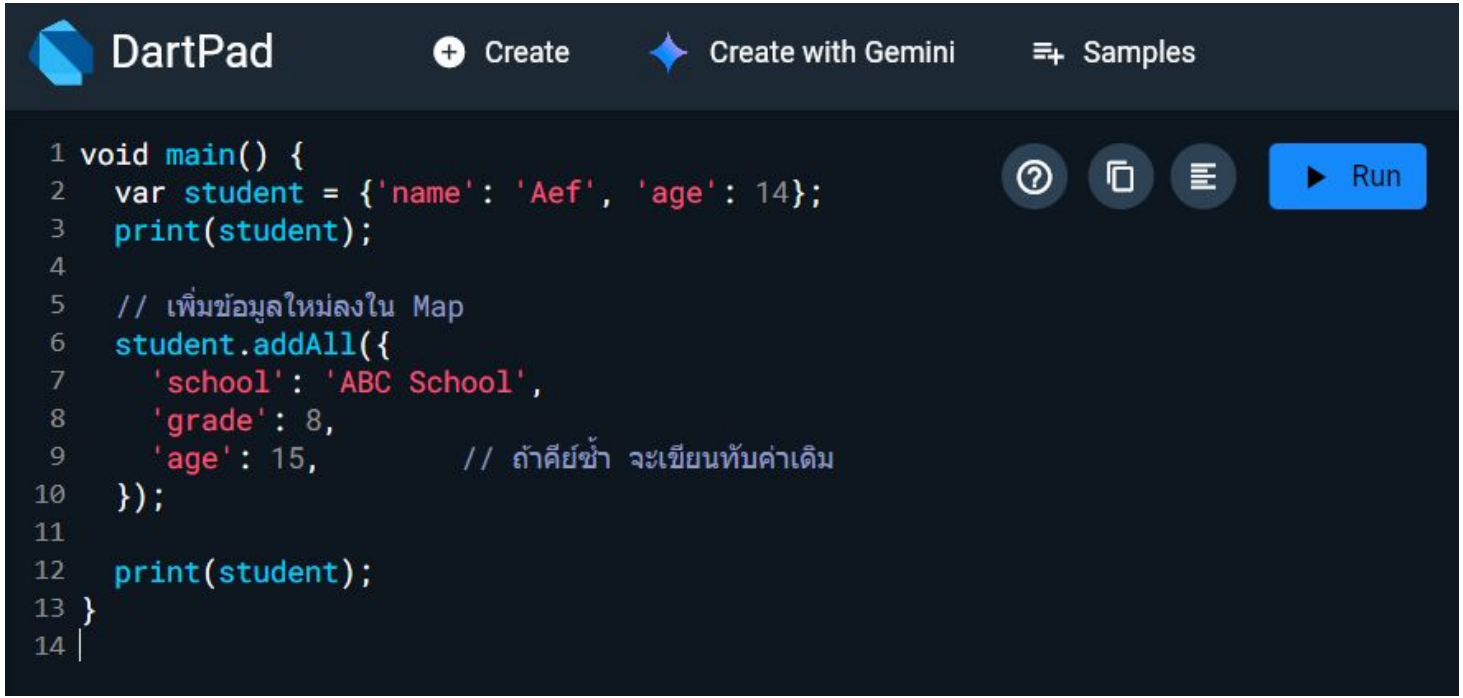
It is generally not allowed to modify the list's length (adding or removing elements) while an operation on the list is being performed, for example during a call to forEach or sort. Changing the list's length while it is being iterated, either by iterating it directly or through iterating an Iterable that is backed by the list, will break the iteration.

การใช้งานข้อมูล List

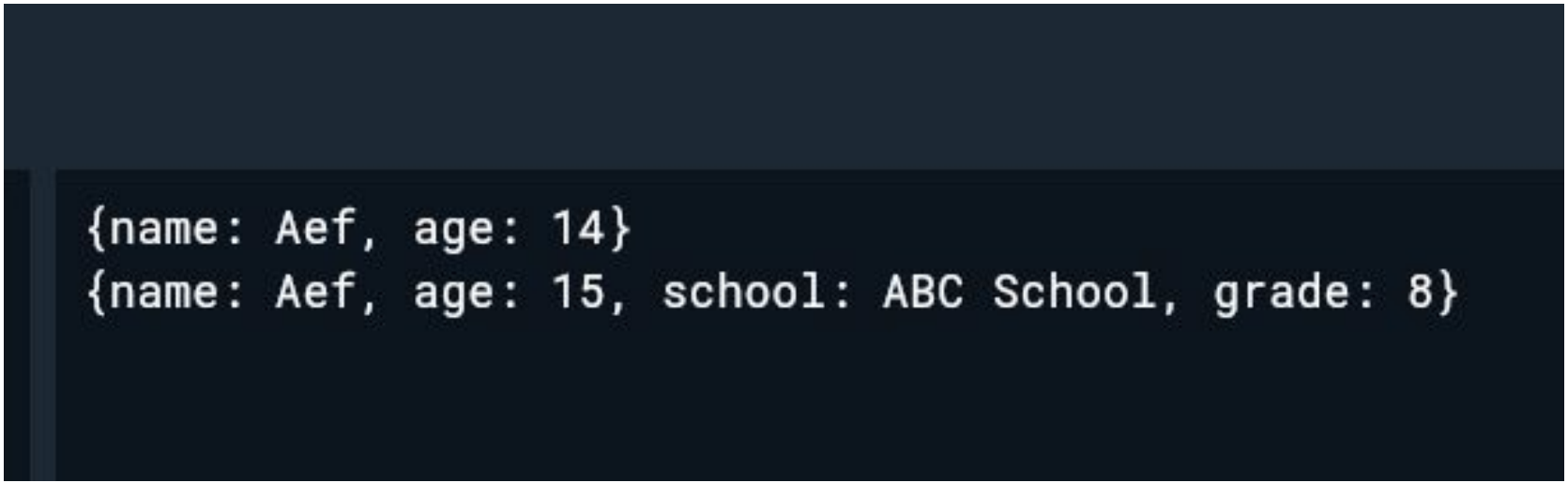


การใช้งานข้อมูล Map

ฟังก์ชันที่มักพบ และ ใช้งานได้บ่อย สำหรับ Map

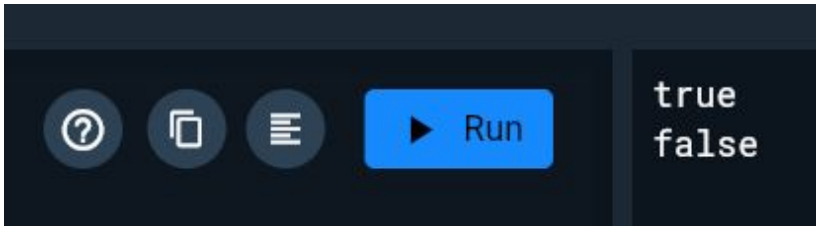


addAll()

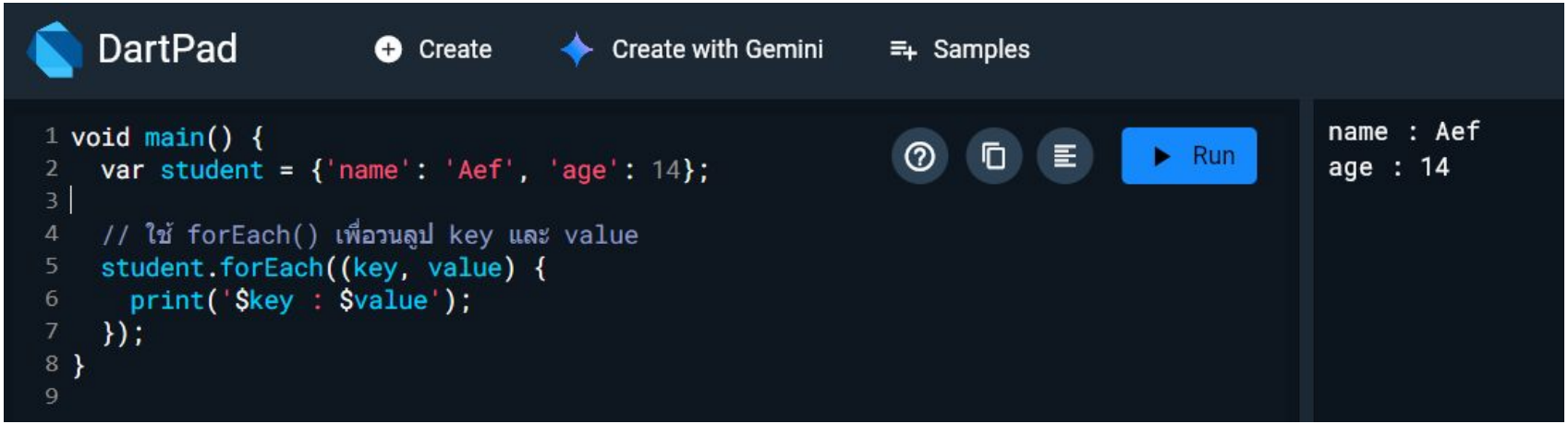


containsKey()

```dart
void main() {
  var student = {'name': 'Aef', 'age': 14};

  // ตรวจสอบว่ามี value นี้หรือไม่
  print(student.containsValue('Aef'));   // true
  print(student.containsValue(20));      // false
}

```



containsValue()



forEach()

ตัวอย่าง forEach() แบบ arrow function

forEach()

putIfAbsent()

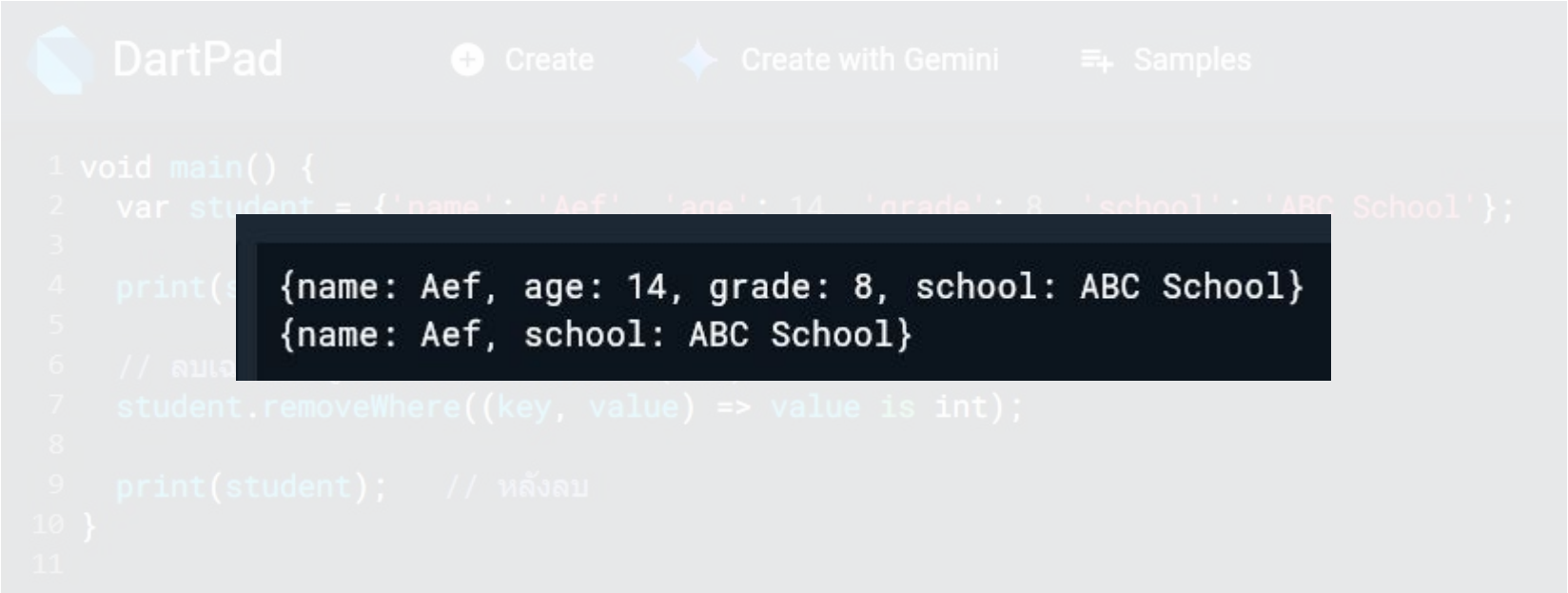
DartPad Create Create with Gemini Samples

```dart
void main() {
  var student = {'name': 'Aef', 'age': 14, 'school': 'ABC School'};

  print(student);   // ก่อนลบ

  // ลบข้อมูลตามชื่อ key
  student.remove('school');       // ลบคีย์ school ออก
  student.remove('grade');        // ไม่มี key นี้ จึงไม่มีผลใด ๆ

  print(student);   // หลังลบ
}
```

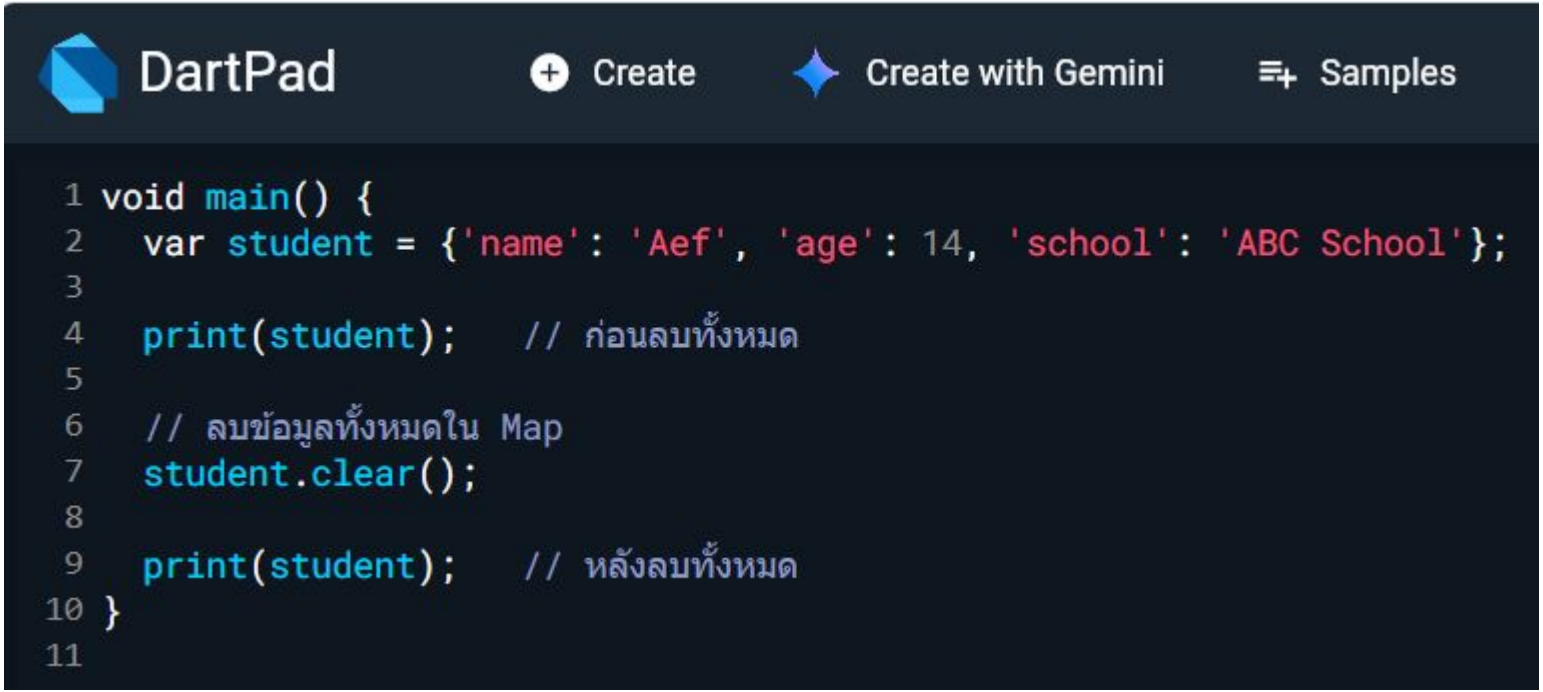
```
{name: Aef, age: 14, school: ABC School}
{name: Aef, age: 14}
```

remove()

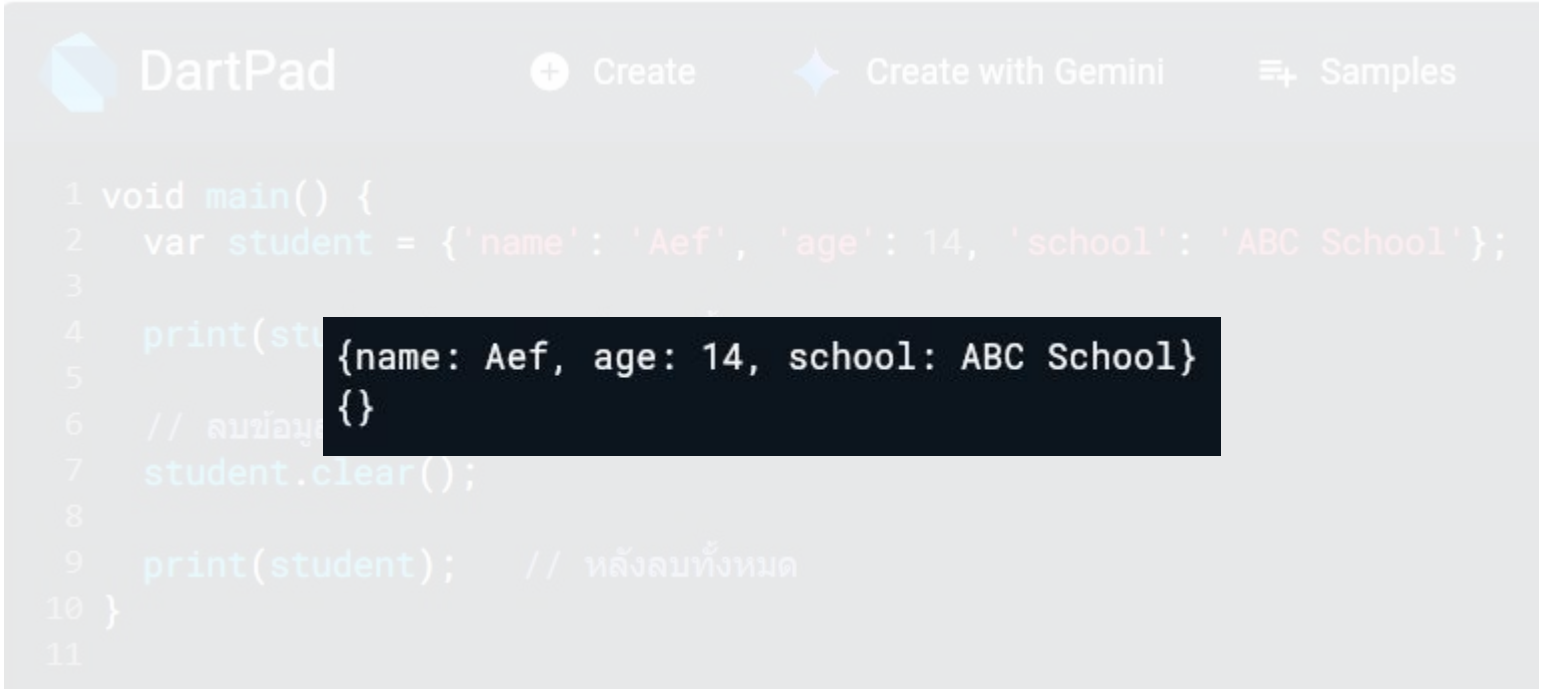
removeWhere()



removeWhere()



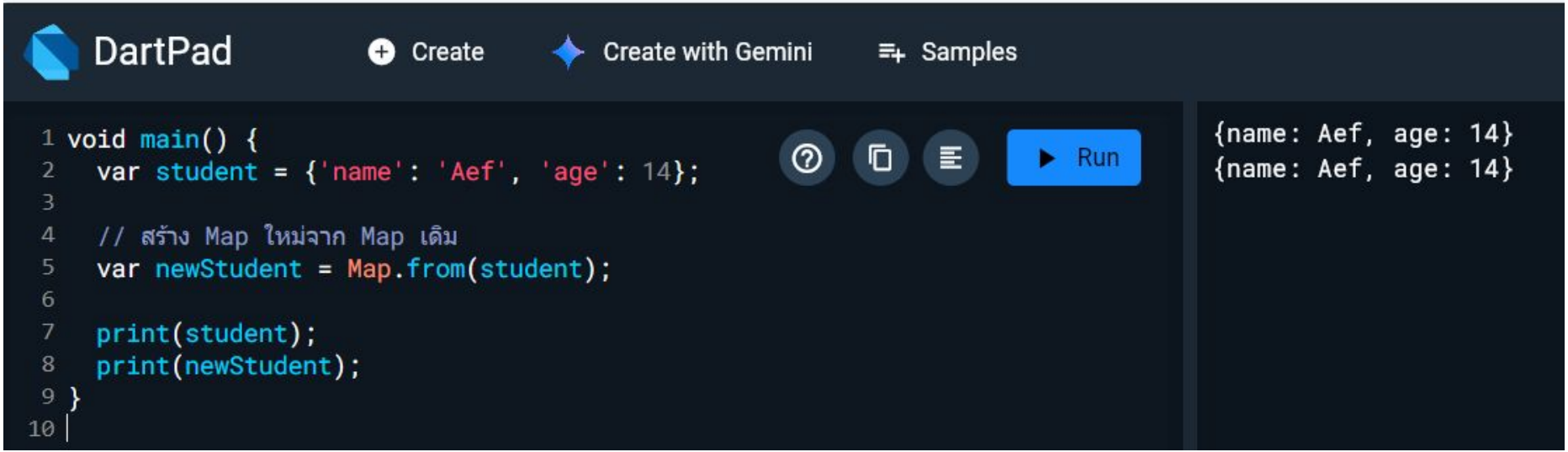
clear()



clear()

update()

update()

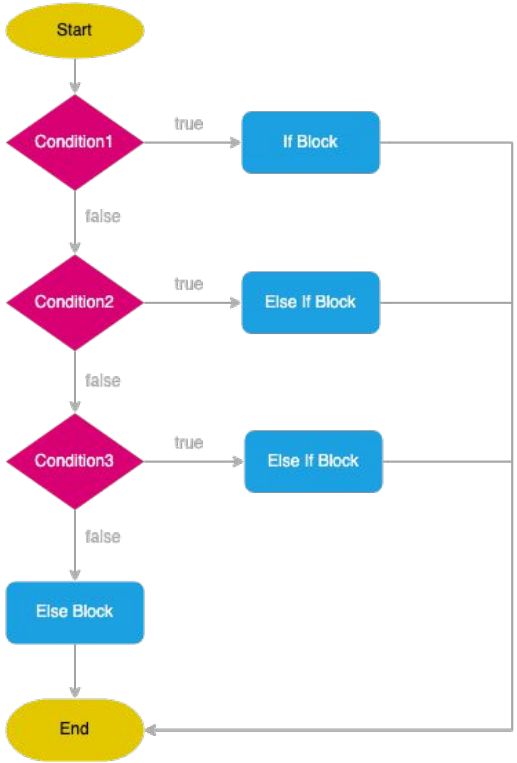


Map.from()

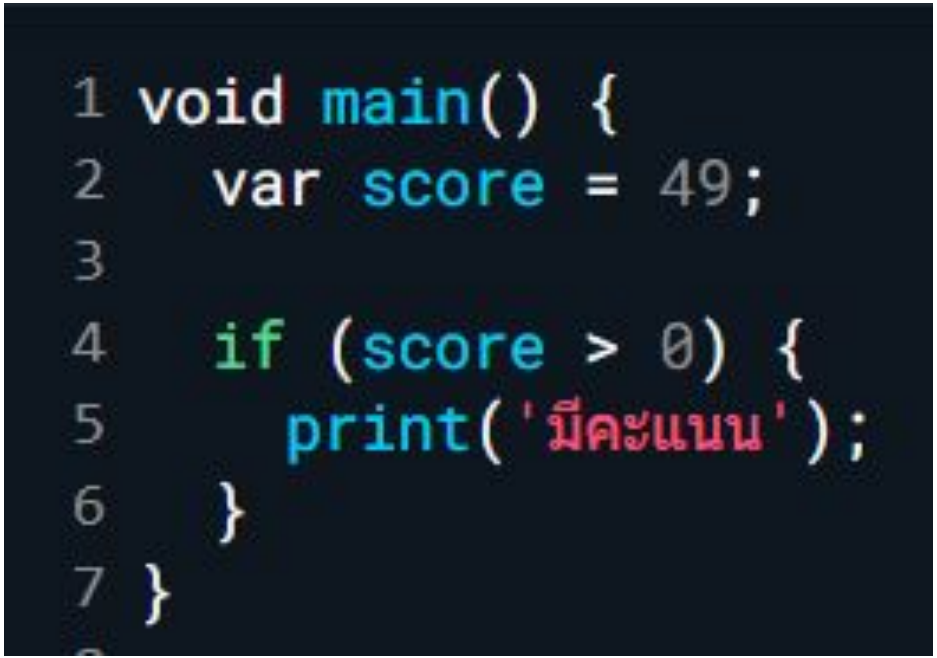
⭐ การใช้งาน conditional logic

```dart
if (condition1)
{
    // statement 1
}
else if (condition2)
{
    // statement 2
}
else if (condition3)
{
    // statement 3
}
else
{
    // else statment
}
```

การใช้เงื่อนไขใน Dart จะเป็น if, else if และ else



การใช้เงื่อนไขใน Dart จะเป็น if, else if และ else



Quiz 1 : ผลลัพธ์ที่แสดงผลคือ ?

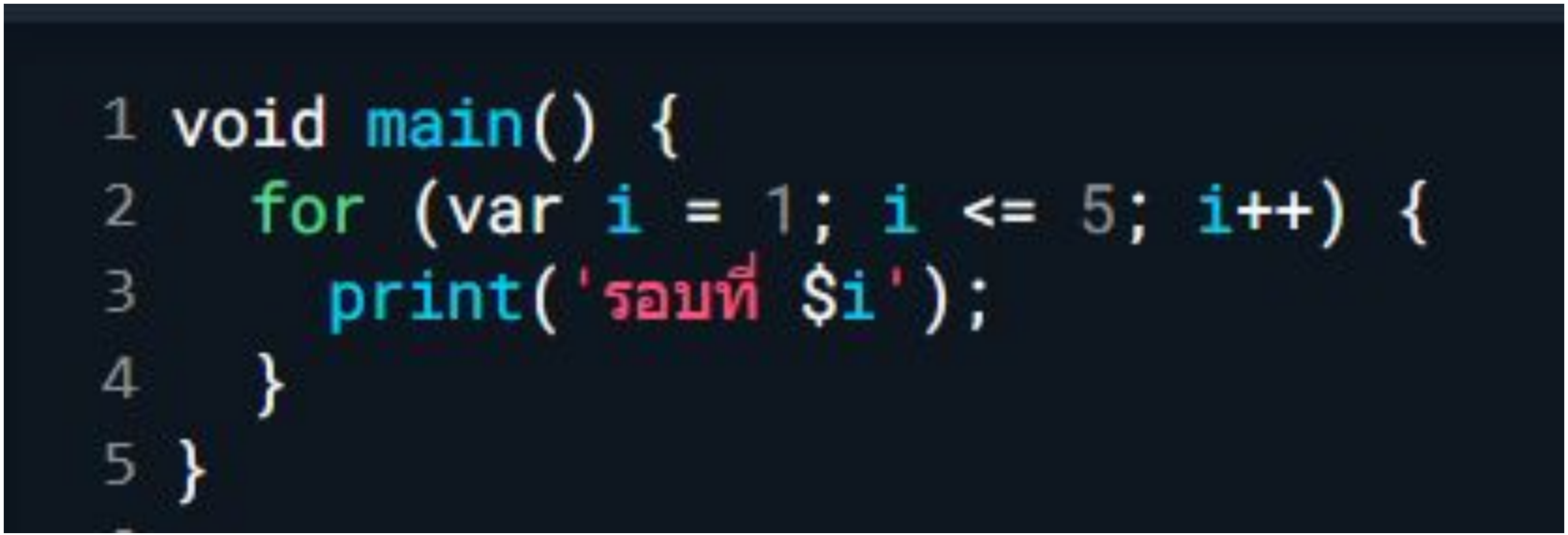
```
void main() {
  var score = 49;

  if (score >= 50) {
    print('เกือบตก');
  } else {
    print('เกือบผ่าน');
  }
}
```

Quiz 2 : ผลลัพธ์ที่แสดงผลคือ ?

```
void main() {
  var score = 49;

  if (score >= 90) {
    print('เอา A ไปเลย');
  } else if (score >= 50) {
    print('เกือบผ่าน');
  } else {
    print('เกือบตก');
  }
}
```

Quiz 3 : ผลลัพธ์ที่แสดงผลคือ ?

⭐ การใช้งาน Loop ใน Dart



การใช้งาน Loop for สำหรับ Dart

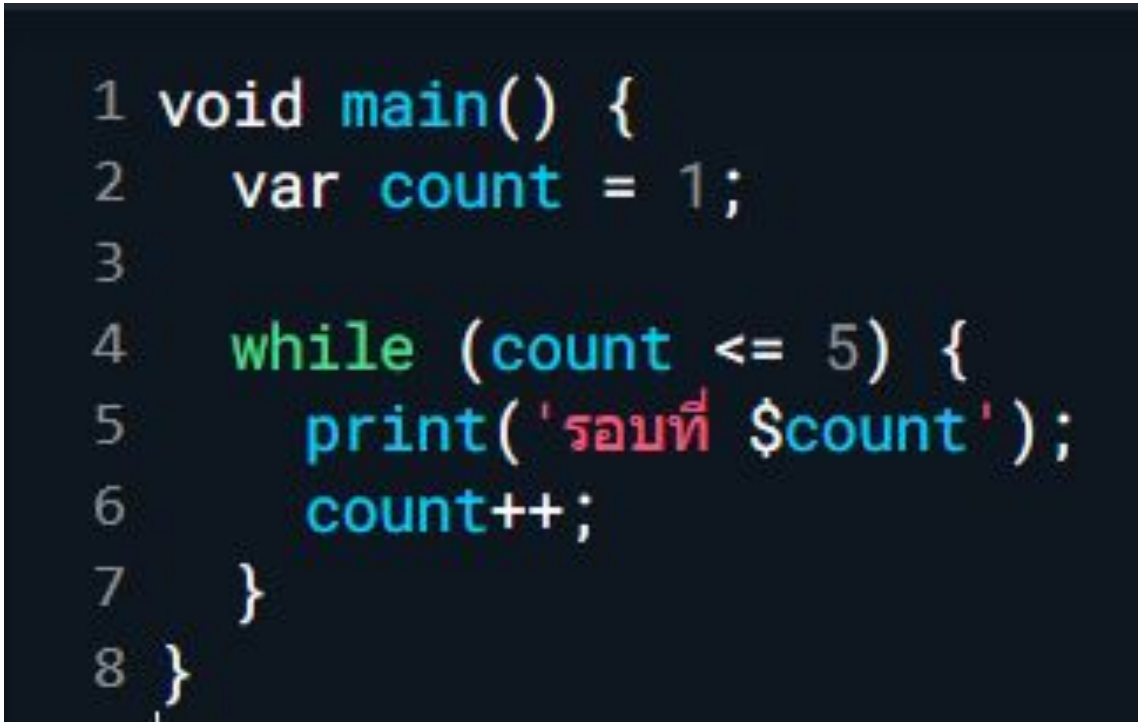
```dart
void main() {
  var numbers = [10, 20, 30, 40, 50];

  for (var i = 0; i < numbers.length; i++) {
    print('รอบที่ ${i + 1} = ${numbers[i]}');
  }
}
```

การใช้งานข้อมูล List

```dart
void main() {
  var numbers = [10, 20, 30, 40, 50];

  for (var i = 0; i < 3; i++) {
    print('รอบที่ ${i + 1} = ${numbers[i]}');
  }
}

```

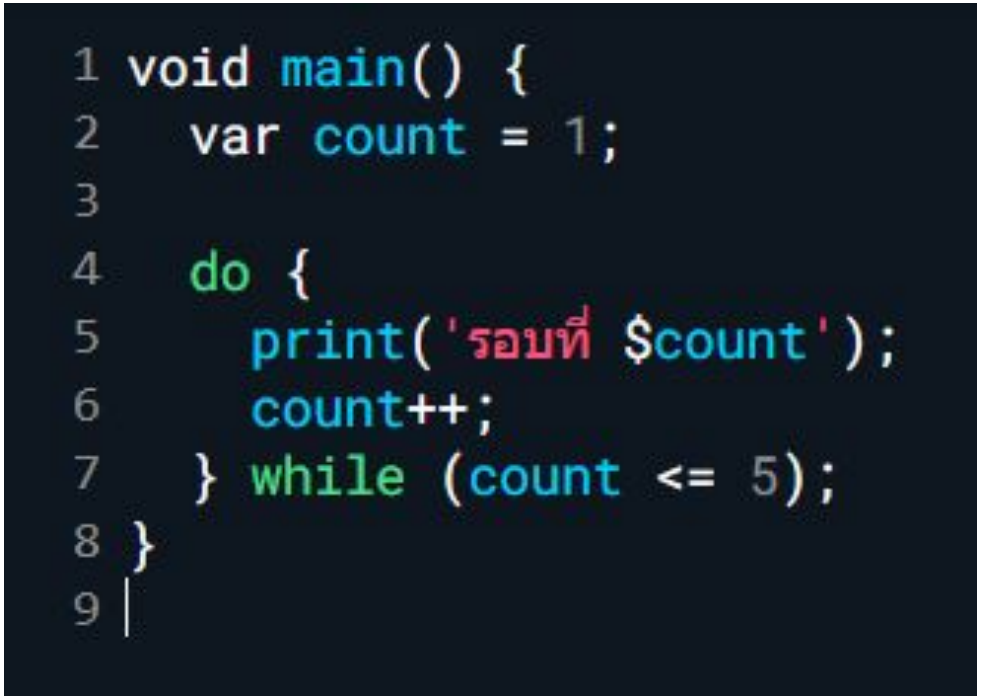
```
รอบที่ 1 = 10
รอบที่ 2 = 20
รอบที่ 3 = 30
```

การใช้งาน Loop for สำหรับ Dart

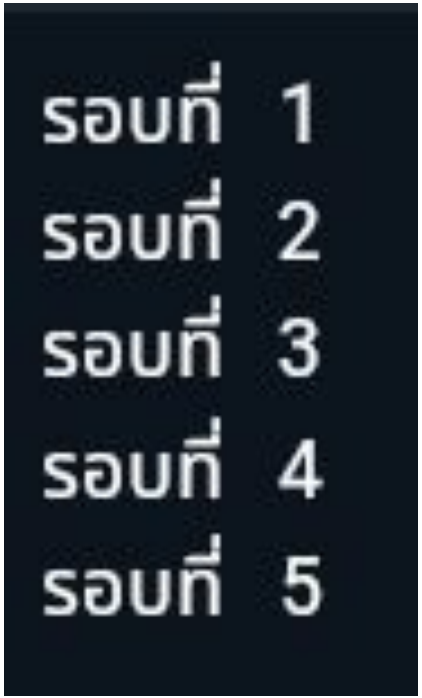


```dart
void main() {
  var count = 1;

  while (count <= 5) {
    print('รอบที่ $count');
    count++;
  }
}
```

การใช้งาน Loop while สำหรับ Dart



```dart
void main() {
  var count = 1;

  do {
    print('รอบที่ $count');
    count++;
  } while (count <= 5);
}
```



```
รอบที่ 1
รอบที่ 2
รอบที่ 3
รอบที่ 4
รอบที่ 5
```

การใช้งาน Loop do while สำหรับ Dart

```dart
void main() {
  List<String> fruits = ['apple', 'banana', 'cherry'];
  for (var fruit in fruits) {
    print(fruit);
  }
}
```

การใช้งาน Loop for-in สำหรับ Dart

```dart
void main() {
  List<int> numbers = [1, 2, 3, 4, 5, 13, 7, 8, 9, 10];
  for (var number in numbers) {
    if (number % 13 == 0) {
      print(number);
      break;
    }
  }
}
```

การใช้งาน break สำหรับ Dart

⭐ OOP with Dart

การใช้งาน class ใน Dart

```
1 class Car {
2     String brand;
3
4     Car(this.brand);
5 }
```

Quiz : จากความรู้ที่มี เราจะสร้าง Obj car ใน main ได้อย่างไร ?

```dart
class Car {
  String brand;

  Car(this.brand);
}

void main() {
 Car mycar = new Car("BMW");
 print(mycar.brand);
}
```

Quiz : จากความรู้ที่มี เราจะสร้าง Obj car ใน main ได้อย่างไร ?

เราสามารถใช้ var แทนได้เช่นกัน

```
class Car {
  String brand;
  Car(this.brand);
}

class ElectricCar extends Car {
 ElectricCar(String brand) : super(brand);
}
```

Quiz : จากความรู้ที่มี เราจะสร้าง Obj ElectricCar ใน main ได้อย่างไร ?

```
void main() {
  var myNewCar = new ElectricCar("Tesla");
  print(myNewCar.brand);
}
```

Quiz : จากความรู้ที่มี เราจะสร้าง Obj ElectricCar ใน main ได้อย่างไร ?

```
class Tesla implements ElectricCar {
  // Implement the interface here
}
```

การใช้งาน interface ใน Dart

```dart
// Interface
abstract class Shape {
  double getArea();
}

// Implementing the Shape interface
class Square implements Shape {
  Square(this.side);
  final double side;

  @override
  double getArea() => side * side;
}

class Circle implements Shape {
  Circle(this.radius);
  final double radius;

  @override
  double getArea() => 3.14 * radius * radius;
}
```

ตัวอย่าง 1 : การใช้งาน interface ใน Dart

```dart
// Interface
abstract class Shape {
  double getArea();
}

// Implementing the Shape interface
class Square implements Shape {
  Square(this.side);
  final double side;

  @override
  double getArea() => side * side;
}

class Circle implements Shape {
  Circle(this.radius);
  final double radius;

  @override
  double getArea() => 3.14 * radius * radius;
}
```

Quiz ตัวอย่าง 1 : เรียกใช้ยังไงหละ ?

```dart
// Interface
abstract class Shape {
  double getArea();
}

// Implementing the Shape interface
class Square implements Shape {
  Square(this.side);
  final double side;

  @override
  double getArea() => side * side;
}

class Circle implements Shape {
  Circle(this.radius);
  final double radius;

  @override
  double getArea() => 3.14 * radius * radius;
}
```

```dart
void main(){
  var myCircle = new Circle(10);
  print(myCircle.getArea());
}
```

```
314
```

Quiz ตัวอย่าง 1 : เรียกใช้ยังไงหละ ?

```dart
// Interface
abstract class Vehicle {
  void move();
  void stop();
}

// Implementing the Vehicle interfa
class Car implements Vehicle {
  @override
  void move() {
    print('Car is moving');
  }

  @override
  void stop() {
    print('Car has stopped');
  }
}
```

```dart
class Bike implements Vehicle {
  @override
  void move() {
    print('Bike is moving');
  }

  @override
  void stop() {
    print('Bike has stopped');
  }
}
```

ตัวอย่าง 2 : การใช้งาน interface ใน Dart

```dart
// Interface
abstract class Vehicle {
  void move();
  void stop();
}

// Implementing the Vehicle interfa
class Car implements Vehicle {
  @override
  void move() {
    print('Car is moving');
  }

  @override
  void stop() {
    print('Car has stopped');
  }
}
```

```dart
class Bike implements Vehicle {
  @override
  void move() {
    print('Bike is moving');
  }

  @override
  void stop() {
    print('Bike has stopped');
  }
}
```

Quiz ตัวอย่าง 2 : เรียกให้รถยนต์วิ่งยังไงหละ ?

```dart
// Interface
abstract class Vehicle {
  void move();
  void stop();
}

// Implementing the Vehicle interfa
class Car implements Vehicle {
  @override
  void move() {
    print('Car is moving');
  }

  @override
  void stop() {
    print('Car has stopped');
  }
}
```

```dart
class Bike implements Vehicle {
  @override
  void move() {
    print('Bike is moving');
  }

  @override
  void stop() {
    print('Bike has stopped');
  }
}
```

```dart
void main(){
  var myCar = new Car();
  myCar.move();
}
```

Quiz ตัวอย่าง 2 : เรียกให้รถยนต์วิ่งยังไงหละ ?

main.dart

```dart
// Interface
abstract class Vehicle {
  void move();
  void stop();
}

// Implementing the Vehicle interfa
class Car implements Vehicle {
  @override
  void move() {
    print('Car is moving');
  }

  @override
  void stop() {
    print('Car has stopped');
  }
}
```

```dart
class Bike implements Vehicle {
  @override
  void move() {
    print('Bike is moving');
  }

  @override
  void stop() {
    print('Bike has stopped');
  }
}
```

```
Car is moving
```

```dart
void main(){
  var myCar = new Car();
  myCar.move();
}
```

Quiz ตัวอย่าง 2 : เรียกให้รถยนต์วิ่งยังไงหละ ?

main.dart

```dart
// Interface
abstract class Printer {
  void printDocument();
  void scanDocument();
}

// Implementing the Printer interface
class LaserPrinter implements Printer {
  @override
  void printDocument() {
    print('Printing document with Laser Printer');
  }

  @override
  void scanDocument() {
    print('Scanning document with Laser Printer');
  }
}
```

```dart
class InkjetPrinter implements Printer {
  @override
  void printDocument() {
    print('Printing document with Inkjet Printer');
  }

  @override
  void scanDocument() {
    print('Scanning document with Inkjet Printer');
  }
}
```

ตัวอย่าง 3 : การใช้งาน interface ใน Dart

main.dart

```dart
// Interface
abstract class Printer {
  void printDocument();
  void scanDocument();
}

// Implementing the Printer interface
class LaserPrinter implements Printer {
  @override
  void printDocument() {
    print('Printing document with Laser Printer');
  }

  @override
  void scanDocument() {
    print('Scanning document with Laser Printer');
  }
}
```

```dart
class InkjetPrinter implements Printer {
  @override
  void printDocument() {
    print('Printing document with Inkjet Printer');
  }

  @override
  void scanDocument() {
    print('Scanning document with Inkjet Printer');
  }
}
```

Quiz ตัวอย่าง 3 : ใช้ InjetPrinter ยังไง ?

```dart
// Interface
abstract class Printer {
  void printDocument();
  void scanDocument();
}

// Implementing the Printer interface
class LaserPrinter implements Printer {
  @override
  void printDocument() {
    print('Printing document with Laser Printer');
  }

  @override
  void scanDocument() {
    print('Scanning document with Laser Printer');
  }
}
```

```dart
class InkjetPrinter implements Printer {
  @override
  void printDocument() {
    print('Printing document with Inkjet Printer');
  }

  @override
  void scanDocument() {
    print('Scanning document with Inkjet Printer');
  }
}
```

```dart
void main(){
  var myPrinter = new InkjetPrinter();
  myPrinter.printDocument();
}
```

Quiz ตัวอย่าง 3 : ใช้ InjetPrinter ยังไง ?

main.dart

```dart
// Interface
abstract class Printer {
  void printDocument();
  void scanDocument();
}

// Implementing the Printer interface
class LaserPrinter implements Printer {
  @override
  void printDocument() {
    print('Printing document with Laser Printer');
  }

  @override
  void scanDocument() {
    print('Scanning document with Laser Printer');
  }
}
```

```dart
class InkjetPrinter implements Printer {
  @override
  void printDocument() {
    print('Printing document with In
  }

  @override
  void scanDocument() {
    print('Scanning document with In
  }
}
```

Console | Shell

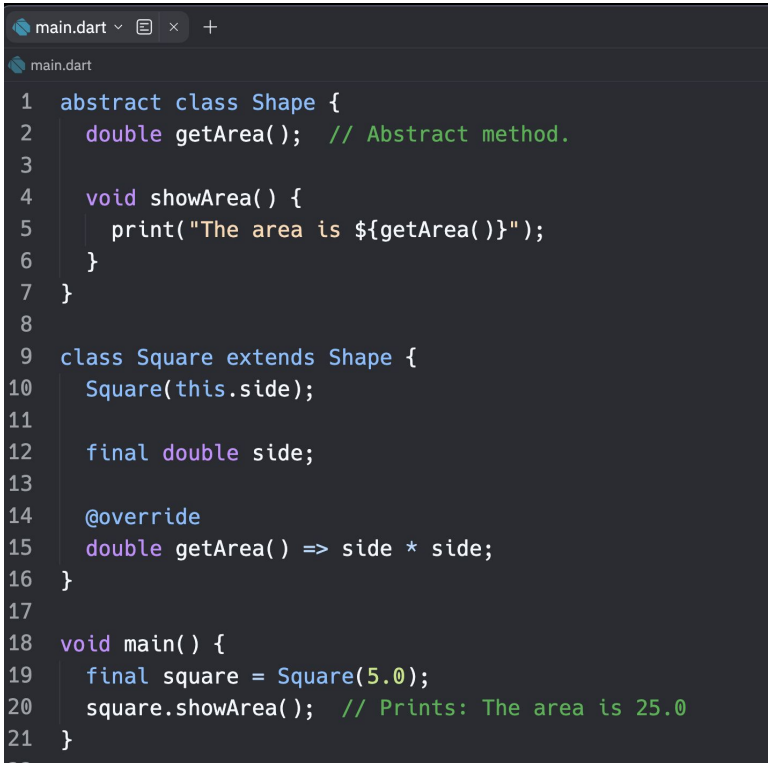
```
dart main.dart
Printing document with Inkjet Printer
```

```dart
void main(){
  var myPrinter = new InkjetPrinter();
  myPrinter.printDocument();
}
```

Quiz ตัวอย่าง 3 : ใช้ InjetPrinter ยังไง ?

```dart
abstract class AbstractCar {
  void honk(); // Abstract method
}
```

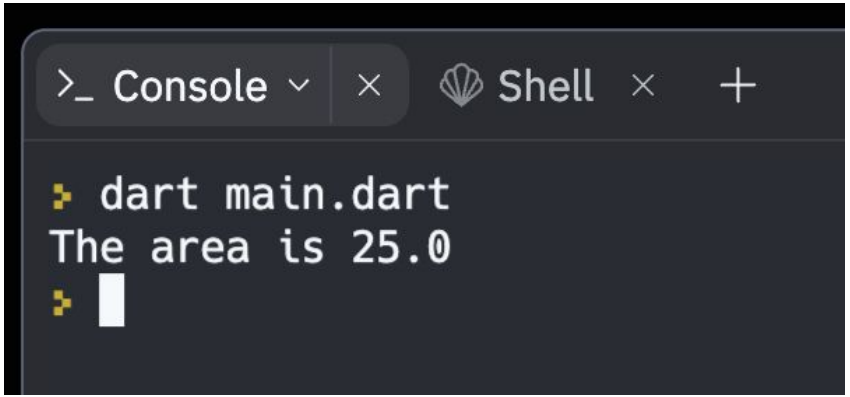
การใช้งาน Abstract ใน Dart



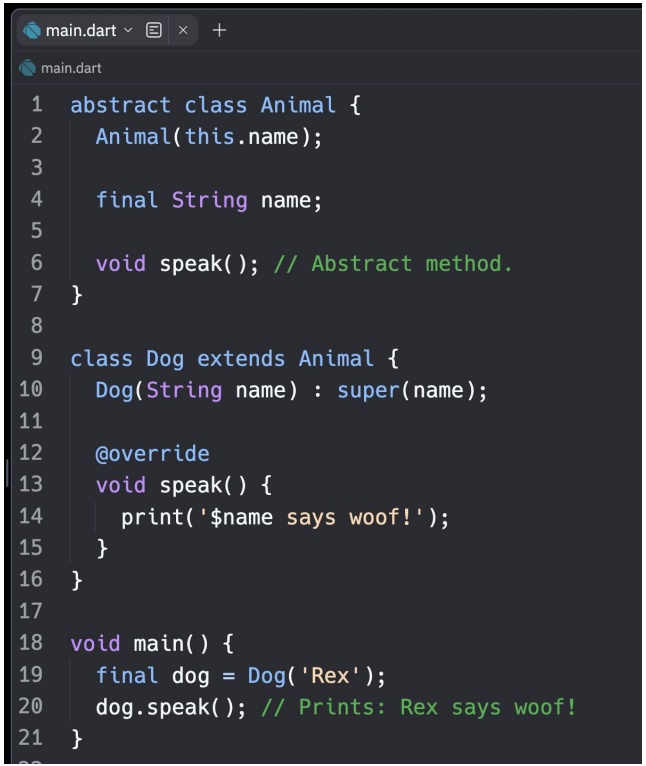
ตัวอย่าง 1 :การใช้งาน Abstract ใน Dart

main.dart

```dart
abstract class Shape {
  double getArea();  // Abstract method.

  void showArea() {
    print("The area is ${getArea()}");
  }
}

class Square extends Shape {
  Square(this.side);

  final double side;

  @override
  double getArea() => side * side;
}

void main() {
  final square = Square(5.0);
  square.showArea();  // Prints: The area is 25.0
}
```



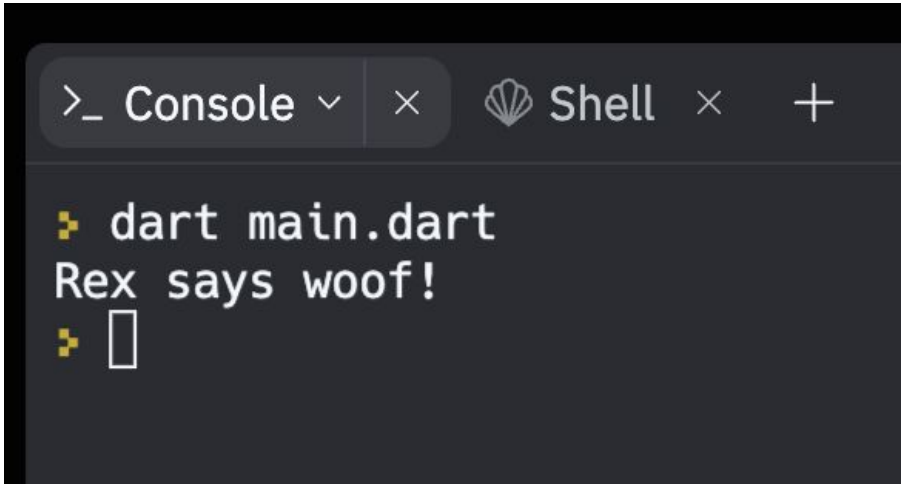
ตัวอย่าง 1 :การใช้งาน Abstract ใน Dart



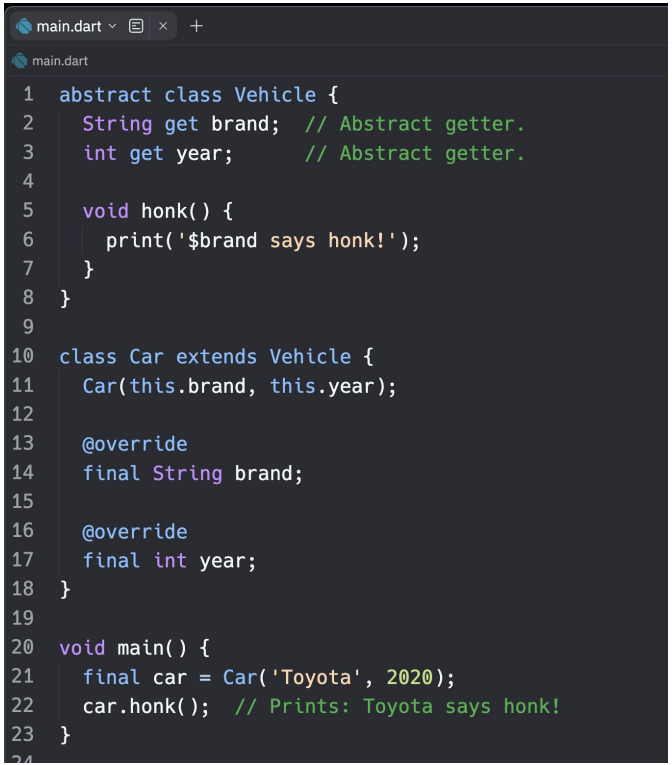
ตัวอย่าง 2 :การใช้งาน Abstract ใน Dart

main.dart

```dart
abstract class Animal {
  Animal(this.name);

  final String name;

  void speak(); // Abstract method.
}

class Dog extends Animal {
  Dog(String name) : super(name);

  @override
  void speak() {
    print('$name says woof!');
  }
}

void main() {
  final dog = Dog('Rex');
  dog.speak(); // Prints: Rex says woof!
}
```



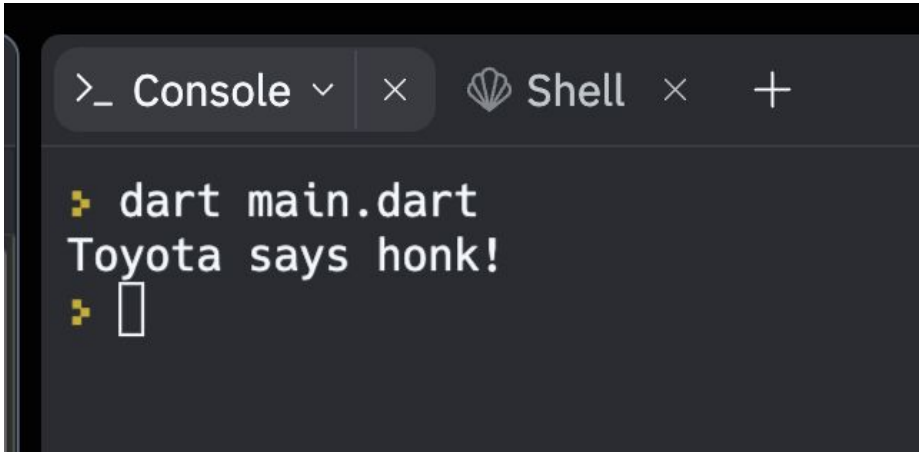
ตัวอย่าง 2 :การใช้งาน Abstract ใน Dart



ตัวอย่าง 3 :การใช้งาน Abstract ใน Dart

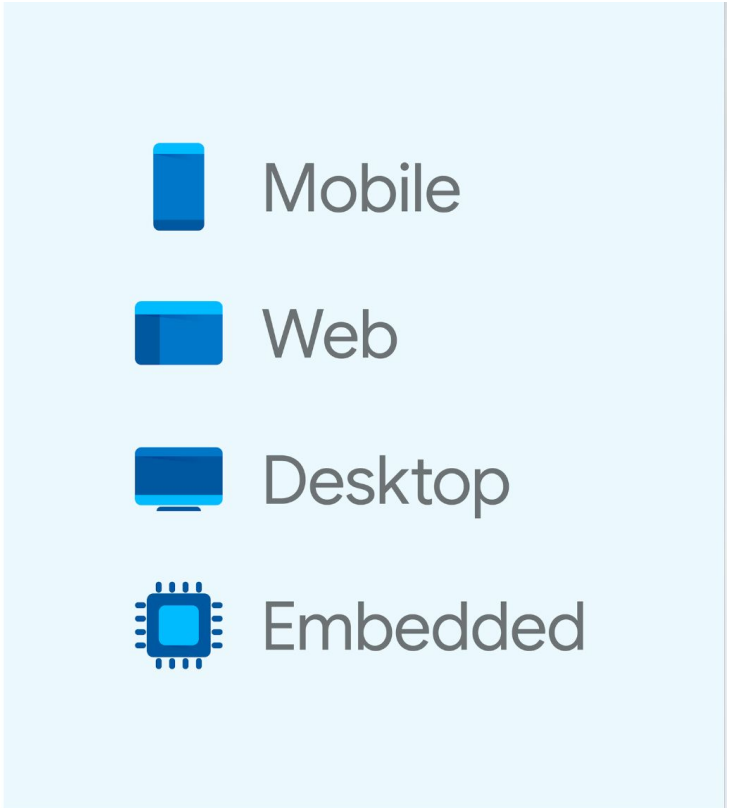
main.dart

```dart
abstract class Vehicle {
  String get brand;  // Abstract getter.
  int get year;      // Abstract getter.

  void honk() {
    print('$brand says honk!');
  }
}

class Car extends Vehicle {
  Car(this.brand, this.year);

  @override
  final String brand;

  @override
  final int year;
}

void main() {
  final car = Car('Toyota', 2020);
  car.honk();  // Prints: Toyota says honk!
}
```



ตัวอย่าง 3 :การใช้งาน Abstract ใน Dart

⭐ กลับมาดูที่ Flutter



“บน macOS เราสามารถสร้างแอปทีเดียว ให้ใช้ได้ทั้ง iOS, Android ได้เลย”

“แต่สำหรับ Windows และ Linux จะทำได้เฉพาะฝั่ง Android เท่านั้น”

📌 สิ่งที่เราต้องเตรียมสำหรับ Flutter

- **Flutter SDK**
  - Flutter SDK สำหรับจัดการ Project Flutter
  - Git สำหรับใช้งาน Flutter SDK
- **Platform Tools**
  - Android Studio สำหรับพัฒนา บน Android
  - XCode สำหรับใช้พัฒนาบน iOS
- **Virtual Device**
  - Android สำหรับ Preview App
  - iOS สำหรับ Preview App

⭐ เริ่มติดตั้ง Flutter

uu Windows

Windows install

System requirements

To install and run Flutter, your development environment must meet these minimum requirements:

- **Operating Systems**: Windows 10 or later (64-bit), x86-64 based.
- **Disk Space**: 1.64 GB (does not include disk space for IDE/tools).
- **Tools**: Flutter depends on these tools being available in your environment.
  - Windows PowerShell 5.0 or newer (this is pre-installed with Windows 10)
  - Git for Windows 2.x, with the **Use Git from the Windows Command Prompt** option.

    If Git for Windows is already installed, make sure you can run `git` commands from the command prompt or PowerShell.

Get the Flutter SDK

1. Download the following installation bundle to get the latest stable release of the Flutter SDK:

   flutter_windows_3.10.3-stable.zip

   For other release channels, and older builds, check out the SDK archive.

uu macOS