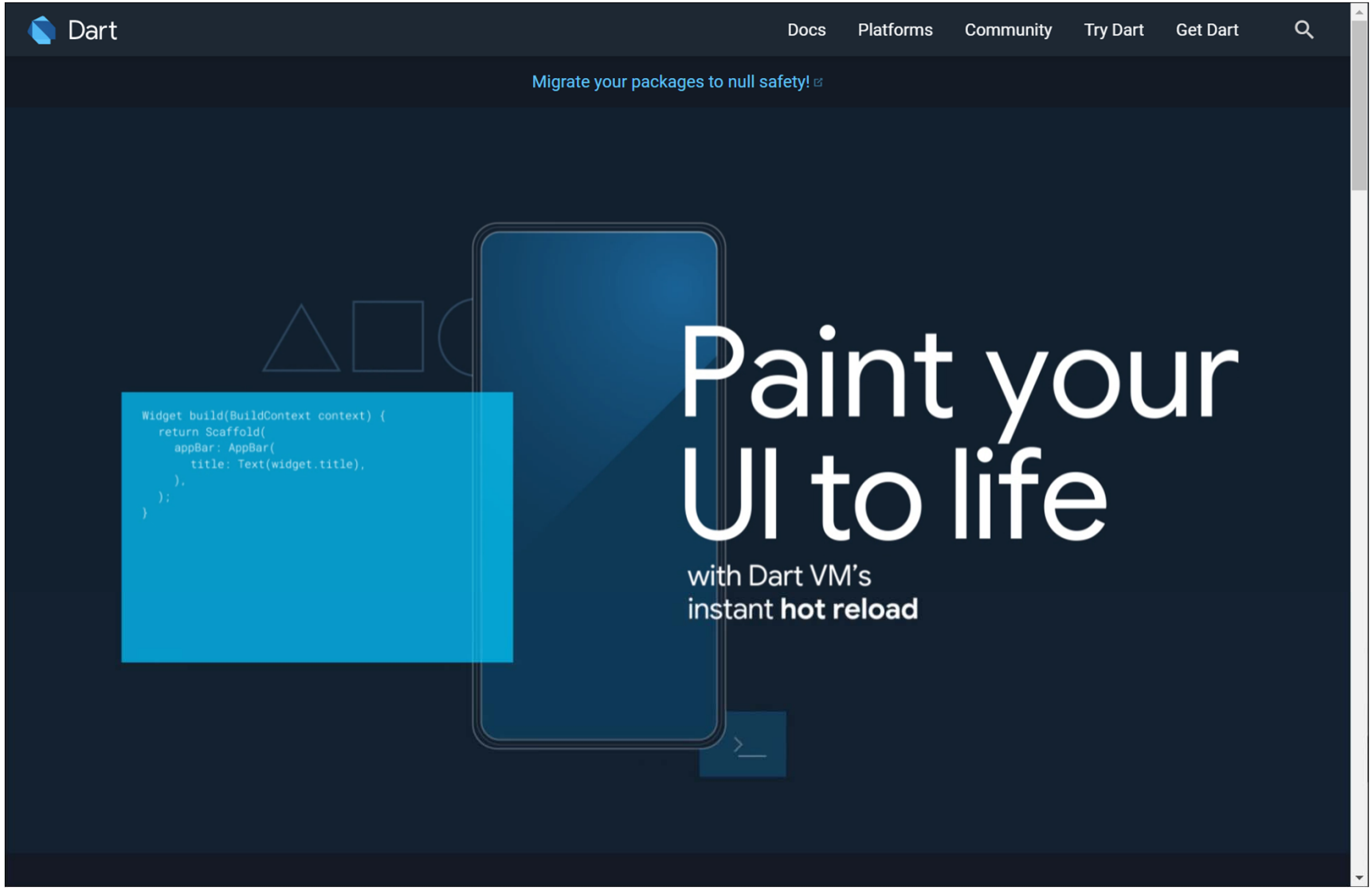
Flutter Fundamentals Module – Dart Primer

Peter Kassenaar –
info@kassenaar.com

Dart Primer

A – very – brief introduction to the Dart Programming language

A Dart Primer



https://dart.dev/

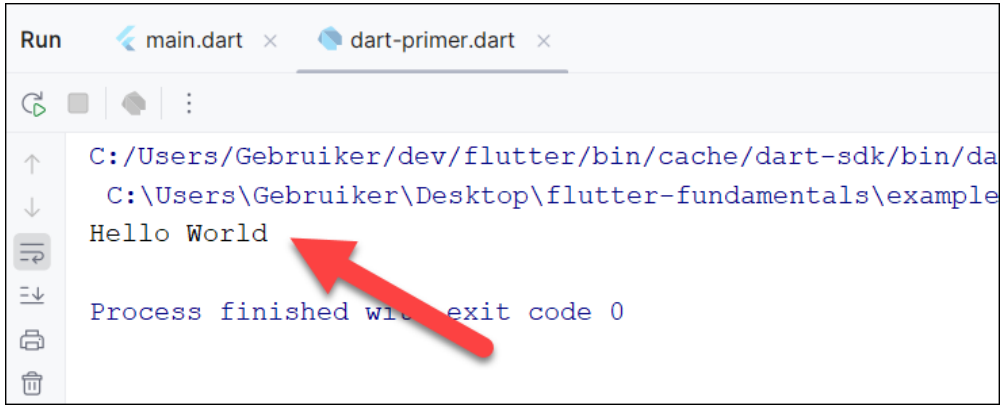
Dart Variables

"Even in type-safe Dart code, most variables don't need explicit types, thanks to type inference"

Top level function

- Every app requires the top-level `main()` function
- Functions that don't explicitly return a value, have the `void` type.
- To display text in the console, use the top-level `print()` function:

```
void main() {
  print ("Hello World");
}
```



Language basics: Type Inference

- You can declare most variables without explicitly specifying their type, using `var`.
- Thanks to type inference, these variables' types are determined by their *initial values*

Type inference for variables

```
var name = 'Voyager I';
var year = 1977;
var antennaDiameter = 3.7;
var flybyObjects = ['Jupiter', 'Saturn', 'Uranus', 'Neptune'];
var image = {
  'tags': ['saturn'],
  'url': '//path/to/saturn.jpg'
};
```

https://dart.dev/language

- You MUST end each statement with a semicolon ;
- You MAY use single and double quotes (‘…’ and “…”)

You MAY explicitly type variables:

```
String name = 'Voyager I';
int year = 1977;
double antennaDiameter = 3.7;
List<String> flybyObjects = ['Jupiter', 'Saturn', 'Uranus', 'Neptune'];
Map<String, Object> image = {
  'tags': ['saturn'],
  'url': '//path/to/saturn.jpg'
};
```

If you WANT variables to change type

- Use the dynamic keyword
  - `dynamic name = ‘Bob’;`
- Using default values:
  - If not initialized, a variabele is `null`:
  - `int myNumber;  // null`
- `final` and `const`
  - If you never intend to change the variabele
  - const variables are implicitly final
  - `Final name = ‘Bob’ // without a type annotation`
  - `Final String name = ‘Bob’;  //`

Built-in types

- Dart knows the following built-in types

  - Numbers (`int`, `double`)
  - Strings (`String`)
  - Booleans (`bool`)
  - Records (`(value1, value2)`)
  - Functions (`Function`)
  - Lists (`List`, also known as *arrays*)
  - Sets (`Set`)
  - Maps (`Map`)
  - Runes (`Runes`; often replaced by the `characters` API)
  - Symbols (`Symbol`)
  - The value `null` (`Null`)

https://dart.dev/language/built-in-types

Other types:

- `Object` – superclass of all Dart objects (except `null`)
- `Enum`
- `Future` and `Stream` – for async operations
- `Iterable`
- `Never`
- `dynamic`
- `void`

Control flow statements

```
if(…) {

    …

} else {

    …

}

for (var object in Objects) { … }

while(…) { … }
```

https://dart.dev/language#control-flow-statements

Functions

Recommended: specify function return type

```
void initState() {
  // initialize state;
}
```

```
int getLength() {
  return someList.length
}
```

```
String printMessage() {
  return “Hello World”;
}
```

https://dart.dev/language/functions

Classes

- No constructor overloading
- Implicitly initialize members in constructor

```
class Person{
  // class properties
  String firstName;
  int age;
  String email;

  // constructor
  Person(this.firstName, this.age, this.email); // No constructor body here

  // method
  String sayHi(){
    return 'Hi, I am $firstName';
  }
}
```

https://dart.dev/language/classes

Instances

The new keyword is optional in newer versions of Dart:

```
Person employee= new Person('Peter', 22, 'peter@test.com'); // valid
```

```
Person employee= Person('Peter', 22, 'peter@test.com'); // also valid
```

Constructors

- Dart knows multiple types of constructors:

Constructors

Constructors are special functions that create instances of classes.

Dart implements many types of constructors. Except for default constructors, these functions use the same name as their class.

- Generative constructors: Creates new instances and initializes instance variables.
- Default constructors: Used to create a new instance when a constructor hasn't been specified. It doesn't take arguments and isn't named.
- Named constructors: Clarifies the purpose of a constructor or allows the creation of multiple constructors for the same class.
- Constant constructors: Creates instances as compile-type constants.
- Factory constructors: Either creates a new instance of a subtype or returns an existing instance from cache.
- Redirecting constructor: Forwards calls to another constructor in the same class.

https://dart.dev/language/constructors

Sound null safety

- Dart enforces sound `null` safety
- This means all variables require a value. They are `non-nullable`
- If you *want* a variable to be (possibly) null, add a ? To the type annotation

```
// 6. Sound null safety
var i = 42; // Inferred to be an int.
i = null; // invalid!

String? name3 = 'Alice'; // name3 MIGHT be assigned the value of null
name3 = null; // valid
```

Null safety principles:

- Non-nullable by default: unless you explicitly tell Dart that a variable can be null, it's considered non-nullable.
- Fully sound: this enables compiler optimizations. If the type system determines that something isn't null, then that thing can never be null.
- More info: https://dart.dev/null-safety
- dart.dev/null-safety/understanding-null-safety

More Dart Language features

- Comments
  - Use `//` for single-line comments
  - Use `/* … */` for multiline comments
- Imports
  - To access APIs defined in other libraries, use `import`.

```
// Importing core libraries –
import 'dart:math';

// Importing libraries from external packages
import 'package:test/test.dart';

// Importing files
import 'path/to/my_other_file.dart';
```

Enums

- Enums are a way of enumerating a predefined set of values or instances in a way which ensures that there cannot be any other instances of that type.

```
// 8. Enums, Defining the Enum
enum LogLevel { info, warning, error }
void showWarning(){
  LogLevel level = LogLevel.warning;
  String message = 'The log level is set to: ${level.name}';
  print (message);
}
```

Inheritance

- Dart knows single inheritance

Inheritance

Dart has single inheritance.

```dart
class Orbiter extends Spacecraft {
  double altitude;

  Orbiter(super.name, DateTime super.launchDate, this.altitude);
}
```

Read more about extending classes, the optional `@override` annotation, and more.

https://dart.dev/language#inheritance

Collections

"A collection is an object that represents a group of objects, which are called elements. Iterables are a kind of collection."

https://dart.dev/language/collections

Common collections

- `List` – like a JavaScript array
- `Set` – like an array, but elements can occur only once
- `Map` – Hashmap, using key/value pairs – like a JavaScript object.

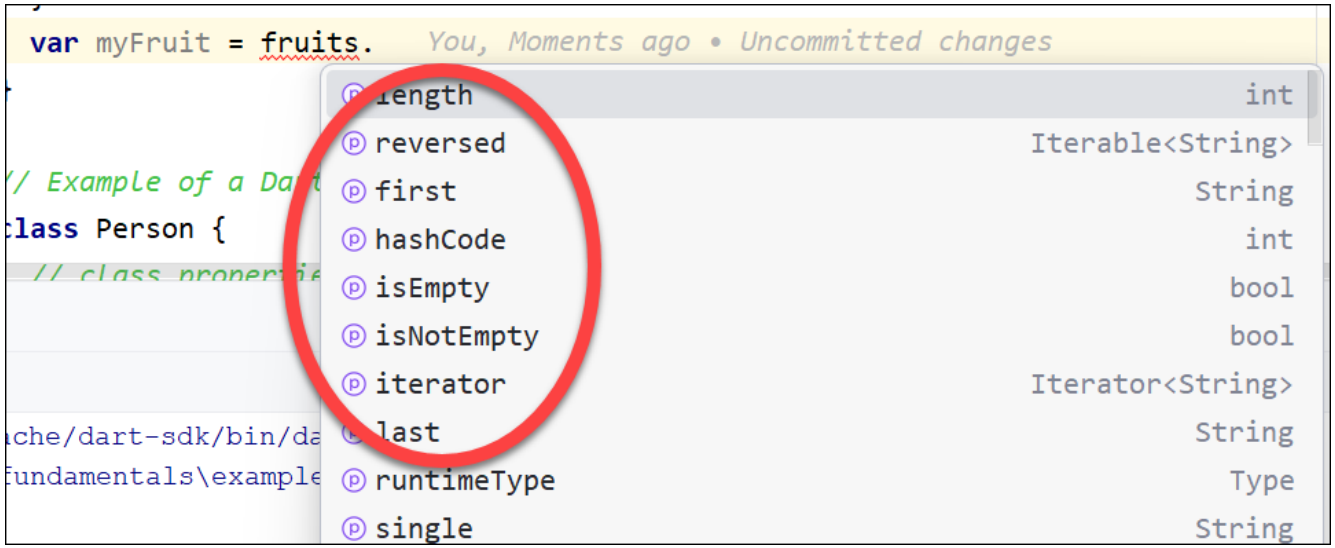
https://dart.dev/language/collections

Collections - List

```
List<String> fruits = ['Apple', 'Pear', 'Banana'];
for (var fruit in fruits) {
  print(fruit);
}
```

Lots of properties & methods available. See https://dart.dev/libraries/dart-core#lists

- .first, .last, .length;
- .forEach(), .firstWhere(), .indexOf()
- ...



Collections – Map

- A Map is an unordered collection of key-value pairs.
- Keys can be of any type, but must be unique.
  - So it *looks* like a JavaScript Object, but is more comparable to a JavaScript `Map()`
- You provide the types of a `Map` in its definition
  - Like in `Map<String, int> persons = { … }`
  - Official notation: `Map<K, V>`
- Keys MUST be unique, `values` can be anything

Using Map<K, V>

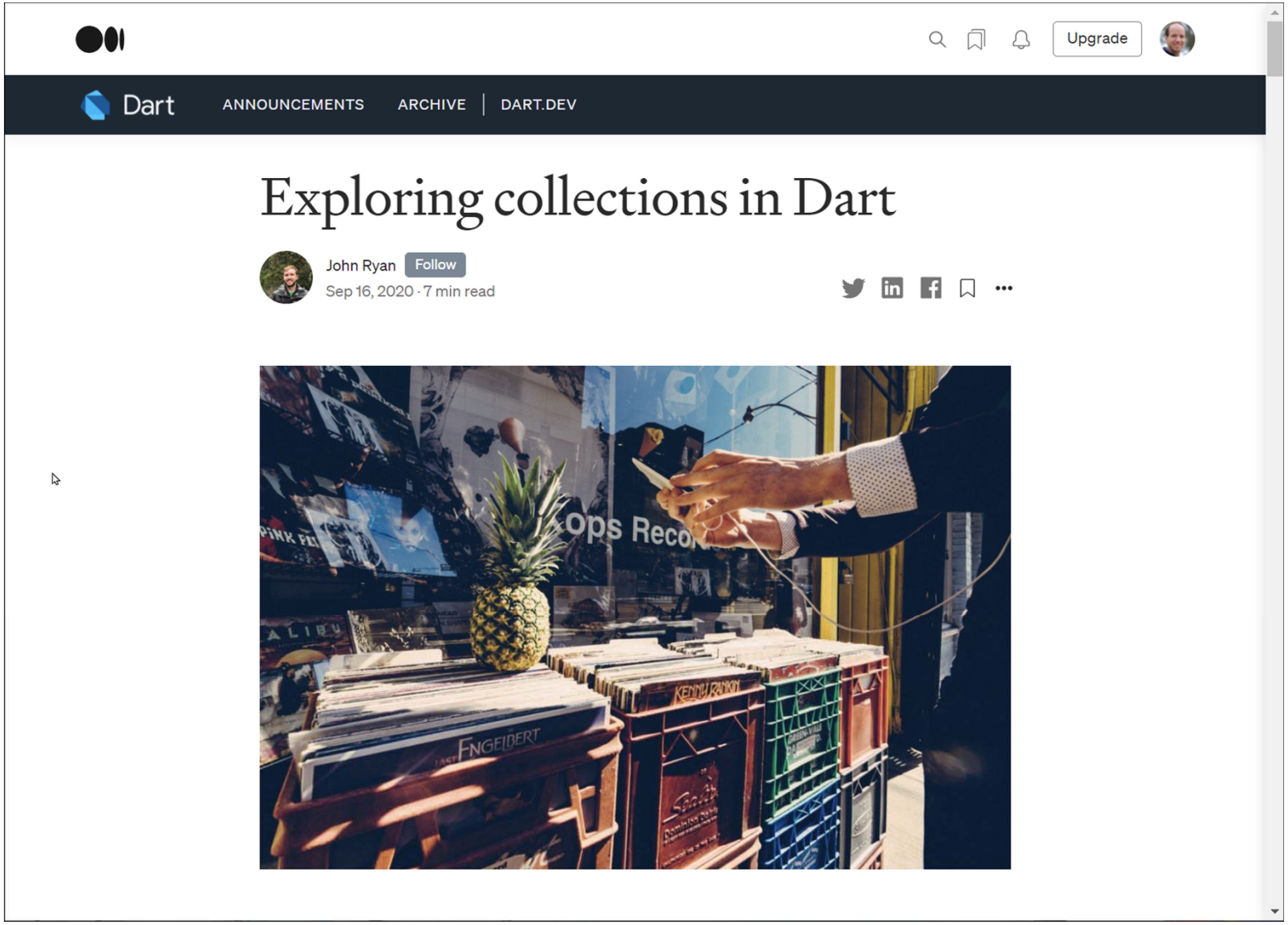
```
Map<String, int> ages = {
  'Alice': 30,
  'Bob': 25,
  'Charlie': 40,
};

// Access items in a map
print(ages['Alice']); // 30

// Add item to a map
ages['Dave'] = 28;

// Iterate over map key/values
ages.forEach((key, value) {
  print('$key is $value years old');
});
```

Dynamic maps

- If you want the values of a Map to be anything, use the keyword dynamic:

```
Map<String, dynamic> persons = {
  'Alice': 30,
  'Bob': 'Engineer',
  'Charlie': {'age': 40, 'hobby': 'golf'}
};
```

More on collections



https://medium.com/dartlang/exploring-collections-in-dart-f66b6a02d0b1

More on Dart

- Dart has single inheritance
  - `Class Employee extends Person { …}`
- Dart has no keyword `interface`
- In Dart you can use mixins
  - https://dart.dev/guides/language/language-tour#adding-features-to-a-class-mixins
- Dart uses `async/await` together with `Future<T>` for async operations

Async operations

```
final oneSecond = Duration(seconds: 1);


…
Future<void> printAfterOneSecond(String msg) async {
  await Future.delayed(oneSecond);
  print(msg);
}
…
printAfterOneSecond('Hello World');
```

Generics

- Using Generics (like in Java, C# or TypeScript) to avoid code duplication
- Annotate between <…>
  - `List<String> fruits`
- https://dart.dev/language/generics

Generics

If you look at the API documentation for the basic array type, `List`, you'll see that the type is actually `List<E>`. The <…> notation marks List as a *generic* (or *parameterized*) type—a type that has formal type parameters. By convention, most type variables have single-letter names, such as E, T, S, K, and V.

Dartpad

DartPad | New Pad | Reset | Format | Install SDK | Java-to-Dart codelab: Basic bicycle example | Samples

RUN

```
class Bicycle {
  int cadence;
  int _speed = 40;
  int gear;

  Bicycle(this.cadence, this.gear);

  @override
  String toString() => 'Bicycle: $_speed mph';

  int get speed => _speed;

  void applyBrake(int decrement) {
  _speed -= decrement;
}

void speedUp(int increment) {
  _speed += increment;
}
}

void main() {
  var bike = new Bicycle(2, 1);
  print(bike);
  bike.applyBrake(1);
  print (bike);
  bike.applyBrake(10);
  print(bike);
  var myNumber = 1;
  myNumber = "Hello World";
  print(myNumber);
}
```

Console

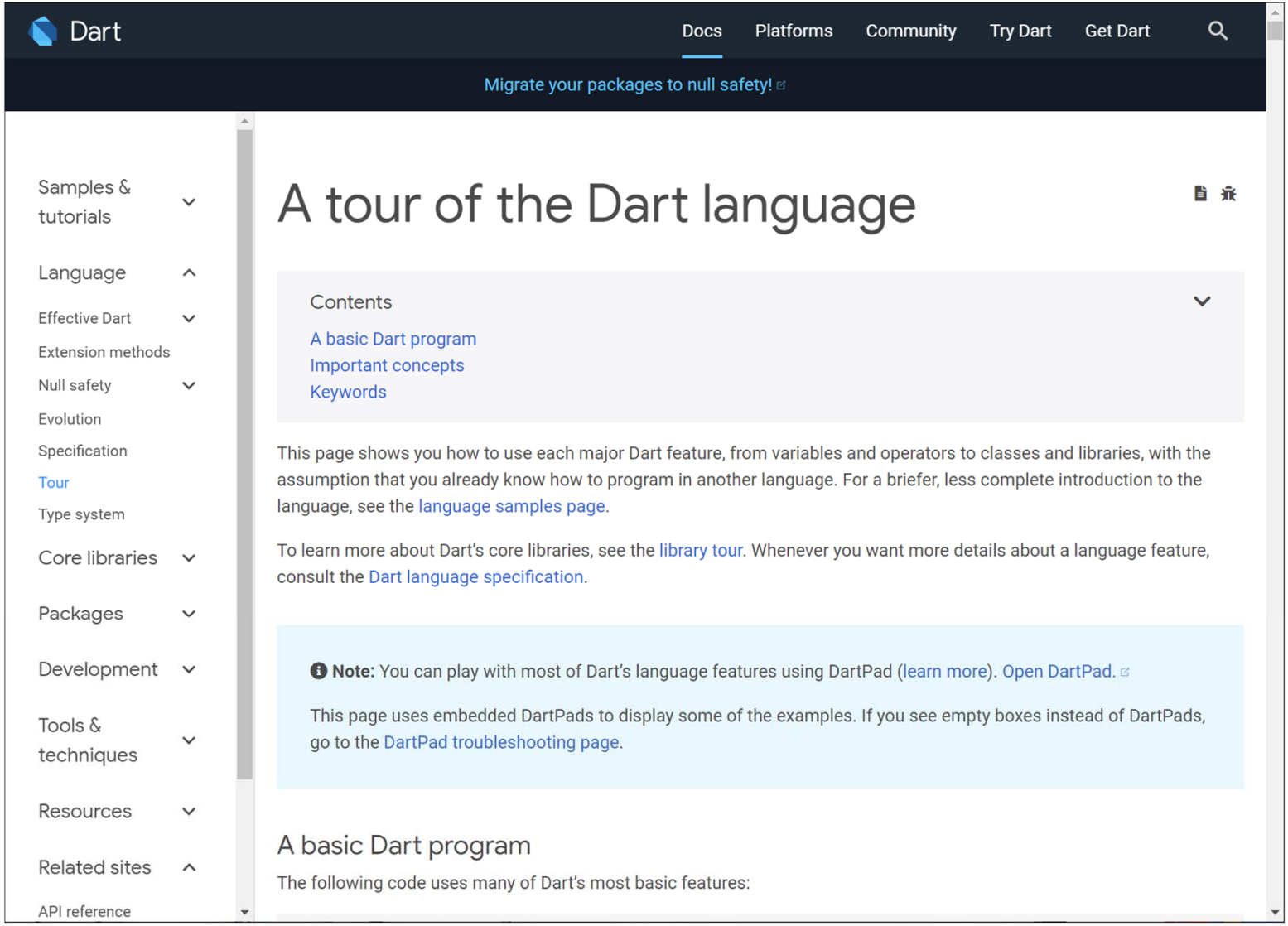
Documentation

error | A value of type 'String' can't be assigned to a variable of type 'int' - line 30

Privacy notice | Send feedback | Null Safety | 1 issue hide | Based on Flutter 1.25.0-8.3.pre Dart SDK 2.10.4

https://dartpad.dev

A Tour of Dart



https://dart.dev/guides/language/language-tour#a-basic-dart-program

Read the Dart blog

Medium Search Write Sign up Sign in

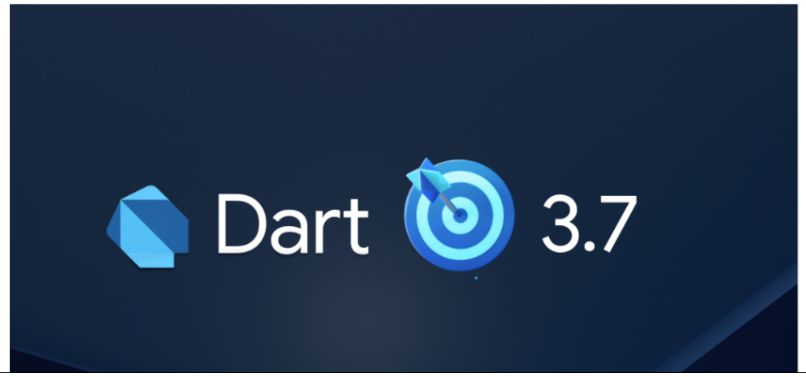
Announcing Dart 3.7

Marya Belanger · Follow
Published in Dart · 6 min read · 6 days ago

1.5K 7

It's a new year, and time for a new Dart stable release. Welcome to Dart 3.7!

The theme for this release is *developer productivity*. We've made the Dart language more consistent by tweaking how it handles wildcard variables. There's also a brand new style in the Dart formatter, new quick fixes and lints in the analyzer, and several cool new features on pub.dev. Let's get into the details!



https://medium.com/dartlang