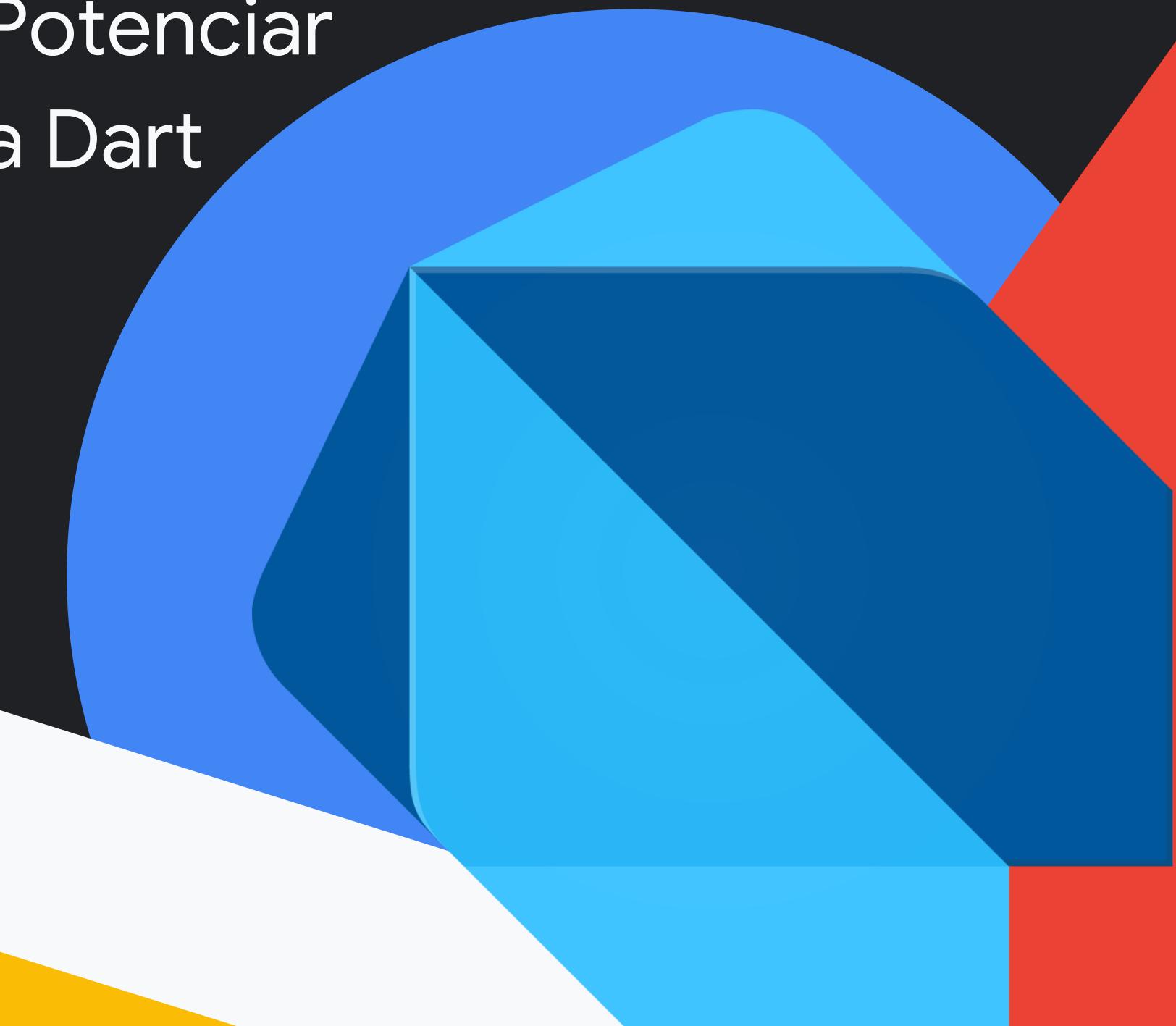


Otra vez instancias y clases Ymás



Yosafat Coronel
GDSC ESCOM IPN
GitHub: YosafatM

Potenciar
a Dart



Itinerario

- Potenciar a los constructores
 - Redirección
 - Como constantes
- Potenciar a las instancias
 - Sobrecarga de operadores
 - Sobreescritura
- Potenciar la herencia y las clases
 - Palabra abstract
 - Constructor factory



A 3D-style cartoon character with brown hair and a wide, open-mouthed smile. He is wearing a red long-sleeved shirt and blue pants. He is holding a brown paper bag that contains a single orange carrot with green tops. The character is positioned behind a large green circle on the left side of the slide.

Constructores

Redirección
Nota sobre heredar
constructores
Constantes

Redirección

Con los constructores, podemos llamar a otros constructores:

```
class Point {  
    double x, y;  
  
    // The main constructor for this class.  
    Point(this.x, this.y);  
  
    // Delegates to the main constructor.  
    Point.alongXAxis(double x) : this(x, 0);  
}
```

La clase Point en su constructor alongXAxis llama al constructor default

Nota

Los constructores no se heredan, pero sí pueden ser accedidos mediante la palabra reservada: **super**

```
class Employee extends Person {  
    Employee() : super.fromJson(fetchDefaultData());  
    // ...  
}
```

fetchDefaultData es un método de **Employee**, mientras que el constructor **fromJson** es de la clase **Person**, se usa redirección.

Const

Si **todas** las propiedades de una clase son **final**, entonces, tenemos permitido llamar y crear constructores con **const**

```
void main() {  
  var origin = const Point(1, 1);  
  var coors = Point.cambiable(1, 5);  
}  
  
class Point {  
  final int x, y;  
  
  const Point(this.x, this.y);  
  Point.cambiable(this.x, this.y);  
  
  @override  
  String toString(){  
    return "($x, $y)";  
  }  
}
```

Privado

Para hacer un constructor privado, solo se usa el guión bajo como si fuera nombrado, con eso ya no se puede usar el constructor default

```
class ConexionPesada {  
    static ConexionPesada? _s;  
  
    ConexionPesada._();  
  
    factory ConexionPesada() {  
        _s ??= ConexionPesada._();  
        return _s!;  
    }  
}
```

Instancias



Sobrecarga de operadores
Nota sobre la sobrecarga
Sobreescritura de `toString`

Operadores

Para definir cómo se usarán los operadores con alguna clase se utiliza la palabra reservada **operator**, más que **sobrescritura**, en realidad sería **sobrecarga**

```
void main() {  
    var a = Point(1, 1);  
    var b = Point(1, 5);  
    //Imprime (2, 6)  
    print("${a + b}");  
}  
  
class Point {  
    final int x, y;  
  
    Point(this.x, this.y);  
  
    operator + (Point p) =>  
        Point(x+p.x, y+p.y);  
  
    @override  
    String toString(){  
        return "($x, $y)";  
    }  
}
```

Operadores

En los tipos no es necesario que sean los mismos que la clase donde se especifica:

<tipo de retorno>

operator <op>(<tipo> <id>)

[bloque/cuerpo]

Notas

Se pueden sobrecargar los de comparación, aritméticos, los de índice

Siempre lleva un argumento, el tipo se recomienda sea el mismo que la clase donde lo estamos especificando.

Sin embargo, en la salida, puede cambiar, por ejemplo, la raíz de un real negativo, ya no da un real. Pero sigue dando un número (que sería un supertipo).

toString()

Uno de los métodos que tiene la clase Object es `toString`, como todas nuestras clases heredan de él, podemos sobreescribir cómo se debe imprimir algo:

```
void main() {  
    var a = Point(1, 1);  
    var b = Point(1, 5);  
    //Imprime (2, 6)  
    print("${a + b}");  
}  
  
class Point {  
    final int x, y;  
  
    Point(this.x, this.y);  
  
    operator + (Point p) =>  
        Point(x+p.x, y+p.y);  
  
    @override  
    String toString(){  
        return "($x, $y)";  
    }  
}
```

noSuchMethod

Otro método heredado por Object es **noSuchMethod**, como el linter no puede saber si algo existe o no con **dynamic**, se llama a este método cuando no existe la propiedad o método.

A las llamadas, también se les llama invocaciones en programación

```
void main() {  
    dynamic c = Point(0, 0);  
    c.add(15);  
}  
  
class Point {  
    final int x, y;  
  
    Point(this.x, this.y);  
  
    @override  
    noSuchMethod(Invocation i) {  
        print("Contacte al administrador")  
    }  
}
```



Herencia

Palabra reservada abstract
Clases y métodos abstractos
Palabra reservada factory

abstract

La palabra `abstract` sirve para declarar las **cosas a medias**, e indicarle a Dart que los que hereden deberán implementarlo

```
// This class is declared abstract and thus
// can't be instantiated.
abstract class AbstractContainer {
    // Define constructors, fields, methods...
    void updateChildren(); // Abstract method.
}
```

Para indicarle a Dart que una clase es abstracta, se usa `abstract`, si un método no tiene cuerpo, queda implícito que es abstracto

abstract

Los métodos abstractos solo pueden estar en clases abstractas, solo se requiere su **signatura**

```
// This class is declared abstract and thus
// can't be instantiated.
abstract class AbstractContainer {
    // Define constructors, fields, methods...

    void updateChildren(); // Abstract method.
}
```

La signatura de un método se compone de: tipo de retorno, nombre y lista de parámetros

abstract

*Un cuerpo vacío
es ya una
implementación*

Las clases que hereden de las abstractas, o bien, implementan los métodos, o siguen siendo abstractas:

```
abstract class Doer {  
    // Define instance variables and methods...  
  
    void doSomething(); // Define an abstract method.  
}  
  
class EffectiveDoer extends Doer {  
    void doSomething() {  
        // Provide an implementation, so the method is not abstract here...  
    }  
}
```

factory

Aunque factory sirve para otra cosa, puede hacer que las clases abstractas se puedan instanciar

Se pueden crear las instancias con Exception()

```
abstract class Exception {  
    factory Exception([var message]) => _Exception(message);  
}  
  
/// Default implementation of [Exception] which carries a message.  
class _Exception implements Exception {  
    final dynamic message;  
  
    _Exception([this.message]);  
  
    String toString() {  
        Object? message = this.message;  
        if (message == null) return "Exception";  
        return "Exception: $message";  
    }  
}
```

factory

Factory es un constructor donde no necesariamente se crea un objeto, podemos pasar uno que ya exista

```
class ConexionPesada {  
    static ConexionPesada? _s;  
  
    ConexionPesada._();  
  
    factory ConexionPesada() {  
        _s ??= ConexionPesada._();  
        return _s!;  
    }  
}
```

¡Muchas gracias!

