Dart

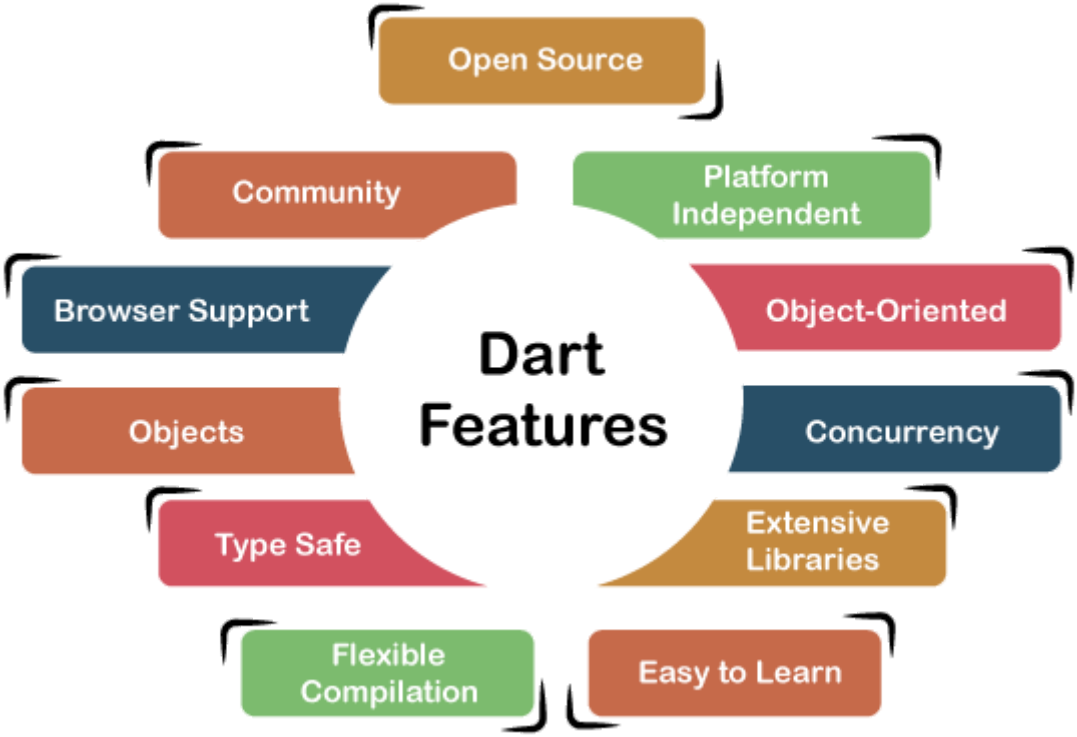
Introduction

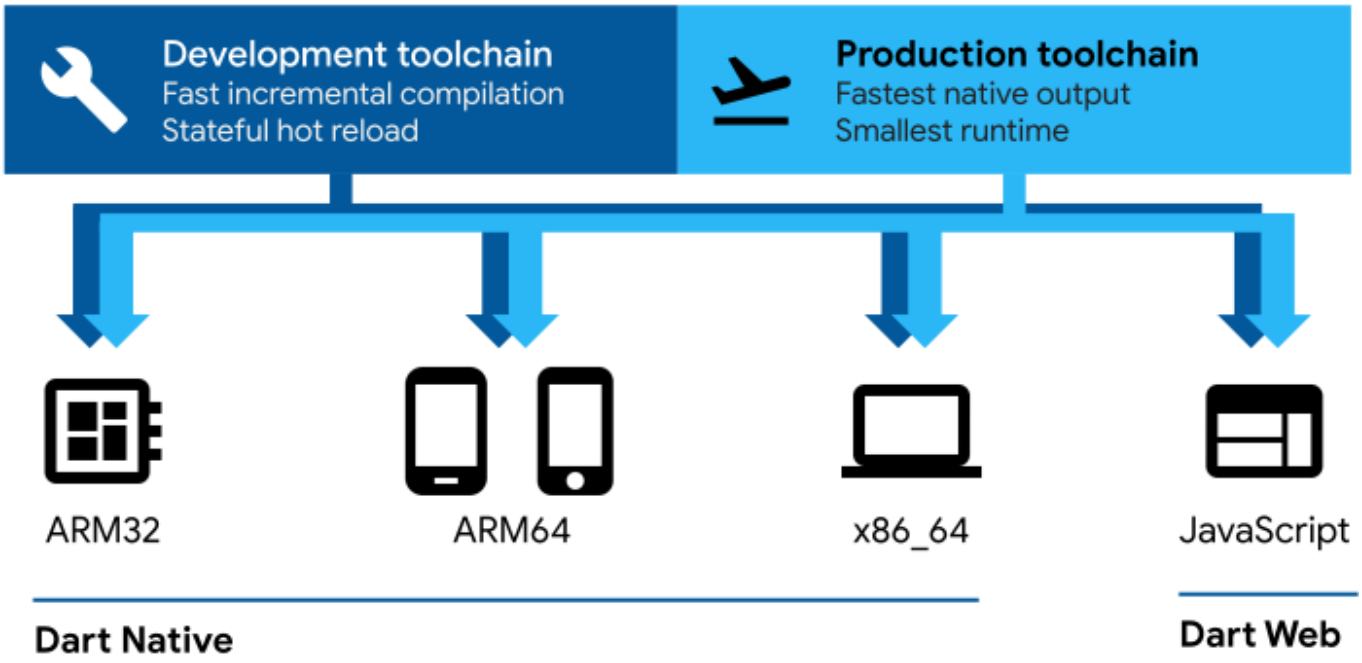
Aperçu

1. **quoi** : un langage de programmation :
    - multiplateforme (*Windows, Mac OS, Linux*)
    - compilable en code natif mobile (*Android/iOS*), bureau (*Windows, Mac OS, Linux*) et JavaScript.
    - open-source (*licence BSD*) et reconnu par le standard ECMA.
    - syntaxiquement proche à C, Java, et JavaScript avec portée lexicale (*lexical scoping*).
    - statiquement typé, mais offrant également un support pour le typage dynamique.
    - orienté-objet (*basé sur les classes*).
    - offrant un support pour la programmation fonctionnelle via les closures, les fonctions anonymes, etc.
    - offrant un support pour la programmation concurrente via les isolates.
2. **pourquoi** : remplacer JavaScript en surmontant ses limitations pour améliorer ses performances.
3. **quand** : première version sortie en 2011 ; première version stable en 2017.
4. **qui** : créé par la communauté Google et popularisé via Flutter.
5. **où** : écrire des scripts simples allant jusqu'à la création d'applications complètes (**bureau** (*Windows/Mac OS/Linux*), **web** (*frontend/backend*), **mobile** (*Android/iOS*)).

Dartpad

1. **lien** : dartpad.dev
2. **définition** : un éditeur entièrement en ligne à partir duquel les utilisateurs peuvent expérimenter les APIs Dart et exécuter du code Dart.
3. **features** : coloration syntaxique, analyse de code, auto-complétion, documentation et édition HTML/CSS.
4. **limitations** : on ne peut pas utiliser toutes les APIs du langage.





Dart Native

1. **pourquoi** : créer des applications bureau ou mobiles.
2. **comment** :
    - compilation **Just-in-Time** (**JIT**) avec la machine virtuelle Dart VM lors du développement qui convertit le code bytecode en code machine natif lors de l'exécution.
    - compilation **Ahead of Time** (**AOT**) pour produire du code machine natif exécutable lors du build de l'application.

Dart Web

1. **pourquoi** : créer des applications Web.
2. **comment** :
    - un compilateur `dartdev` pour exécuter et déboguer les applications web Dart sur le moteur Chrome lors du développement.
    - un compilateur `dart2js` pour transformer le code de l'application Web en code JavaScript optimisé portable sur tous les navigateurs lors de la production.
3. **applications** : `Flutter`, `AngularDart`, $\dots$

```
# compiler Dart en JavaScript en utilisant dart2js
dart2js --out=<output_file>.js <dart_input>.dart
```

Variables

1. **définition** : des allocations mémoire pour stocker des valeurs (*i.e., des références d'objets en mémoire*).
2. **typage** :
    - chaque variable possède un type.
    - le type d'une variable peut être :
        1. **statiquement explicité** en utilisant le type désigné lors de la déclaration.
        2. **statiquement inféré** en utilisant `var` lors de la déclaration.
        3. **dynamiquement inféré** en utilisant `dynamic` lors de la déclaration.
    - par défaut, toutes les variables déclarées et non initialisées valent `null`.
3. **contraintes de nommage** :
    - caractères alphanumériques.
    - pas de caractères spéciaux ou blancs, sauf `_` et `$`.
    - premier caractère alphabétique ou `_` (*pour les membres privés d'un type*).
    - nom unique et sensitive à la casse.
4. **conventions de nommage** :
    - `PascalCase` : classes, interfaces, mixins, enums, typedefs, extensions, et paramètres de type pour les types génériques.
    - `snake_case` : bibliothèques, paquetages, dossiers, fichiers, importations.
    - `camelCase` : le reste.
5. **commentaires** : les commentaires peuvent être **monolignes** (*précédé par `//` sur une ligne*) ou **multilignes** (*circonscrits par `/* */`*).

```
// ../source/dart/snippets/commentaires.dart
// commentaire monoligne

/*
* commentaire
* multi-ligne
*/

/// commentaire
/// multi-ligne
```

```dart
// ../source/dart/snippets/variables.dart
void main() {
  /* déclaration, initialisation et affichage
    d'une variable de type statique (String)
  */
  String nom = "John";
  print(nom); // John

  /* erreur à la compilation, parce que
    nom est de type String et
    la valeur affectée est un entier
  */
  // nom = 123;

  // pas d'erreur, la valeur affectée est un String
  nom = "Alice";
  print(nom); // Alice

  /* déclaration, initialisation et affichage
    d'une variable de type dynamique
  */
  dynamic x = "Bob"; // x est un String maintenant
  print(x); // Bob

  // pas d'erreur, x est un number maintenant
  x = 123;
  print(x); // 123

  /* déclaration, initialisation et affichage
    d'une variable ayant un type statique inféré
    (bool)
  */
  var y = true;
  print(y); // true

  /* erreur à la compilation, parce que
    y est de type bool et
    la valeur affectée est un entier
  */
  // y = 123;
  y = false;
  print(y); // false

  /* déclaration d'une variable en utilisant var
    sans initialisation -> variable de type dynamique
  */
  var z;
  z = "Man"; // x est un String maintenant (initialisation)
  print(z); // Man

  // pas d'erreur, x est un number maintenant
  z = 123;
```

```
  print(z); // 123
}
```

Types

Data Type	Keyword	Description
Number	`num`, `int`, `double`	numbers (ints and doubles), integers, and floating points with double precision
String	`String`	sequences of characters
Boolean	`bool`	boolean values (`true` and `false`)
List	`List`	ordered lists of objects
Map	`Map`	maps of key-value pairs
Runes	`Runes`	integer Unicode code points of a String

Opérateurs

Opérateurs arithmétiques

symbol	description	utilisation	remarques
+	addition	a + b	-
-	soustraction	a - b	-
*	multiplication	a * b	-
/	division réelle	a / b	-
~/	division entière	a / b	-
%	modulo	a % b	uniquement avec les entiers
-	moins unaire	-a	-
++	incrémentation	++a ou a++	pré/post-incrémentation
--	décrémentation	--a ou a--	pré/post-décrémentation

```
// ../source/dart/snippets/operators/arithmetic_operators.dart
void main() {
  /* OPERATEURS ARITHMETIQUES */
  print("OPERATEURS ARITHMETIQUES");
  print("========================");
  int i1 = 5, i2 = 10;
  print("i1 = $i1"); // i1 = 5
  print("i2 = $i2"); // i2 = 10
```

```
  print("-i1 = ${-i1}"); // -i1 = -5
  print("i1 + i2 = ${i1 + i2}"); // i1 + i2 = 15
  print("i1 - i2 = ${i1 - i2}"); // i1 - i2 = -5
  print("i1 * i2 = ${i1 * i2}"); // i1 * i2 = 50
  print("i1 / i2 = ${i1 / i2}"); // i1 / i2 = 0.5
  print("i1 ~/ i2 = ${i1 ~/ i2}"); // i1 ~/ i2 = 0
  print("i1 % i2 = ${i1 % i2}"); // i1 % i2 = 5

  int i3 = i1++;
  int i4 = --i2;
  print("i3 = i1++ = $i3"); // i3 = i1++ = 5
  print("i4 = --i2 = $i4"); // i4 = --i2 = 9
  print("i1 = $i1"); // i1 = 6
  print("i2 = $i2"); // i2 = 9
}
```

Opérateurs relationnels

symbol	description	utilisation	remarques
==	égalité physique	a == b	deux références sur le même objet
!=	différence physique	a != b	deux références sur deux objets différents
<	inférieur	a < b	-
<=	inférieur ou égal à	a <= b	-
>	supérieur	a > b	-
>=	supérieur ou égal à	a >= b	-

```
// ../source/dart/snippets/operators/relational_operators.dart
void main() {
  /* OPERATEURS RELATIONNELS */
  print("OPERATEURS RELATIONNELS");
  print("======================");
  int i1 = 5, i2 = 10;
  print("i1 = $i1"); // i1 = 5
  print("i2 = $i2"); // i2 = 10

  print("i1 == i2 : ${i1 == i2}"); // i1 == i2 : false
  print("i1 != i2 : ${i1 != i2}"); // i1 != i2 : true
  print("i1 < i2 : ${i1 < i2}"); // i1 < i2 : true
  print("i1 <= i2 : ${i1 <= i2}"); // i1 <= i2 : true
  print("i1 >= i2 : ${i1 > i2}"); // i1 >= i2 : false
  print("i1 > i2 : ${i1 >= i2}"); // i1 > i2 : false

  i1 = 10;
  print("\ni1 = $i1"); // i1 = 10
  print("i2 = $i2"); // i2 = 10
  print("i1 == i2 : ${i1 == i2}"); // i1 == i2 : true
```

```
  print("i1 != i2 : ${i1 != i2}"); // i1 != i2 : false
  print("i1 < i2 : ${i1 < i2}"); // i1 < i2 : false
  print("i1 <= i2 : ${i1 <= i2}"); // i1 <= i2 : true
  print("i1 >= i2 : ${i1 > i2}"); // i1 >= i2 : false
  print("i1 > i2 : ${i1 >= i2}"); // i1 > i2 : true
}
```

Opérateurs logiques

symbol	description	utilisation	remarques
&&	et logique	a && b	-
||	ou logique	a || b	-
!	non logique	!a	-

```
// ../source/dart/snippets/operators/logical_operators.dart
void main() {
  int a = 5, b = 7;

  // L'opérateur &&
  print("$a > 10 && $b < 10 = ${a > 10 && b <10}"); // 5 > 10 && 7 < 10 =
false

  // L'opérateur ||
  print("$a > 10 || $b < 10 = ${a > 10 || b <10}"); // 5 > 10 || 7 < 10 =
true

  // L'opérateur !
  print("!($a > 10) = ${!(a > 10)}"); // !(5 > 10) = true
}
```

Opérateurs bit à bit (*bitwise*)

symbol	description	utilisation	remarques
&	et binaire	a & b	-
|	ou binaire	a | b	-
~	non binaire (xor)	~a	-
^	ou exclusif binaire (xor)	a ^ b	-
<<	décalage de bit à gauche	a << b	décalage de b bits à gauche
>>	décalage de bit à droite	a >> b	décalage de b bits à droite

```

```

```dart
// ../source/dart/snippets/operators/bitwise_operators.dart
void main() {
  int a = 5, b = 7, c = 14, d = 1;

  // L'opérateur &
  print("$a & $b = ${a & b}"); // 5 & 7 = 1001 & 1011 = 1001 = 5

  // L'opérateur |
  print("$a | $b = ${a | b}"); // 5 | 7 = 1001 | 1011 = 1011 = 7

  // L'opérateur ^
  print("$a ^ $b = ${a ^ b}"); // 5 ^ 7 = 1001 ^ 1011 = 0010 = 2

  // L'opérateur ~
  print("~$a = ${~a}"); // ~5 = ~ 000...1001 = 111...0110 = 4294967290

  // L'opérateur >>
  print("$c >> $d = ${c >> d}"); // 14 >> 1 = 14 / 2^1 = 7

  // L'opérateur <<
  print("$c << $a = ${c << a}"); // 14 << 5 = 14 * 2^5 = 14 * 32 = 448
}
```

Opérateurs d'affectation

symbol	description	utilisation	remarques
=	affectation	a = b	-
+=	addition et affectation raccourcie	a += b	équivaut a = a + b
-=	soustraction et affectation raccourcie	a -= b	équivaut a = a - b
*=	multiplication et affectation raccourcie	a *= b	équivaut a = a * b
/=	division et affectation raccourcie	a /= b	équivaut a = a / b
%=	modulo et affectation raccourcie	a %= b	équivaut a = a % b
&=	et binaire bitwise et affectation raccourcie	a &= b	équivaut a = a & b
|=	ou binaire bitwise et affectation raccourcie	a |= b	équivaut a = a | b
^=	ou exclusif binaire bitwise et affectation raccourcie	a ^= b	équivaut a = a ^ b
<<=	décalage de bit à gauche et affectation raccourcie	a <<= b	équivaut a = a << b
>>=	décalage de bit à droite et affectation raccourcie	a >>= b	équivaut a = a >> b
??=	affectation conditionnelle	a ??= b	équivaut a = b ssi a n'est pas `null`

```dart
// ../source/dart/snippets/operators/assignment_operators.dart
void main() {
  print("\nOPERATEURS D'AFFECTATION");
  print("========================");
  int i1 = 5, i2 = 10;
  print("i1 = $i1"); // i1 = 5
  print("i2 = $i2"); // i2 = 10

  i1 += i2; // i1 = i1 + i2
  print("i1 += i2 -> i1 = $i1"); // i1 += i2 -> i1 = 15

  i1 -= i2; // i1 = i1 - i2
  print("i1 -= i2 -> i1 = $i1"); // i1 -= i2 -> i1 = 5

  i1 *= i2; // i1 = i1 * i2
  print("i1 *= i2 -> i1 = $i1"); // i1 *= i2 -> i1 = 50

  // i1 /= i2; // erreur, car / retourne un double

  double d = (i1 as double) + 0.5; // explicit conversion
  print("double d = (i1 as double) + 0.5 -> d = $d"); // double d = i1 as
double -> d = 50.5
  d /= i2;
  print("d /= i2 -> d = $d"); // d /= i2 -> d = 5.05

  i1 ~/= i2; // i1 = i1 ~/ i2
  print("i1 ~/= i2 -> i1 = $i1"); // i1 ~/= i2 -> i1 = 5

  i1 %= i2; // i1 = i1 % i2
  print("i1 %= i2 -> i1 = $i1"); // i1 %= i2 -> i1 = 5

  var x;

  // l'affectation est effectuée
  // car x = null au préalable
  x ??= i1 + i2;
  print("x ??= i1 + i2 = $i1 + $i2 = $x");

  // x conserve sa valeur
  // car elle n'est pas null avant l'affectation
  x ??= i1 + i2 + i2;
  print("x ??= i1 + i2 + i2 = $x");
}
```

Opérateurs de test de types

symbol	description	utilisation	remarques
is	retourne true si un objet est d'un type spécifié, false sinon	a is Type	-
is!	retourne true si un objet n'est pas d'un type spécifié, true sinon	a is! Type	-

```dart
// ../source/dart/snippets/operators/type_test_operators.dart
void main() {
  String name = "John Doe";
  double x = 10.5;
  int y = 10;

  print("\"$name\" is a String? ${name is String}"); // true
  print("\"$name\" is an int? ${name is int}"); // false

  print("$x is not a double? ${x is! double}"); // false
  print("$x is not an int? ${x is! int}"); // true
}
```

Les constantes

`const`

1. `const` est un modificateur de variables/valeurs utilisé pour désigner une **valeur constante connue à la compilation**.
2. une valeur constante connue à la compilation peut désigner :
    - une **valeur simple** (e.g., valeurs numériques littérales, valeurs String littérales, etc.) ou
    - une **valeur complexe** (e.g., listes, objets, etc.)
3. une valeur complexe constante connue à la compilation est immuable (*i.e., son état/contenu ne peut pas être modifié*).
4. une variable ayant une valeur constante connue à la compilation ne peut pas être modifiée après son initialisation par affectation.
5. la déclaration et l'initialisation d'une variable ayant une valeur constante connue à la compilation ne peuvent pas être séparées.

`final`

1. `final` est un modificateur de variables utilisé pour désigner une variable finale (*i.e., ne pouvant avoir qu'une seule valeur affectée*).
2. une variable finale ne peut pas être modifiée après son initialisation.
3. la déclaration et l'initialisation d'une variable finale peuvent être séparées.
4. la valeur d'une variable finale peut ne peut pas être constante, ni connue à la compilation.

```dart
// ../source/dart/snippets/const_final.dart
void main() {
  /* CONST */
  /* ===== */
  /* avec les nombres */
  /* ---------------- */
  print("CONST");
  print("=====");
  print("Avec les nombres");
  print("----------------");
  int n0 = 1;
```

```dart
// interpolation de la valeur de n0 dans le string affiché
print ("n0 = $n0"); // n0 = 1

const int n1 = 10;
print("n1 = $n1"); // n1 = 10

// erreur à la compilation,
// pas de modification de la variable constante après initialisation
// n1 = 20;

// pas d'erreur,
// car la valeur est constante et connue à la compilation
const int n2 = 10 + 20 * 10;
print("n2 = 10 + 20 * 10 = $n2"); // n2 = 10 + 20 * 10 = 210

// pas d'erreur,
// car la valeur est constante et connue à la compilation,
// parce n1 et n2 sont constantes
const int n3 = n1 + n2;
print("n3 = $n3"); // n3 = 220

// erreur à la compilation,
// la valeur initiale affectée n'est pas constante,
// parce que n0 n'est pas constante
// const int n4 = n0 + n1;

/* avec les Strings */
/* ---------------- */
print("\nAvec les Strings");
print("----------------");
const String str1 = "Hello";
print("str1 = $str1"); // str1 = Hello

// erreur à la compilation,
// pas de modification de la variable constante après initialisation
// str1 = "Hi";

// pas d'erreur,
// on peut déclarer et initialiser une constante non typée
const str2 = "Hola";
print("str2 = $str2"); // str2 = Hola

// erreur à la compilation,
// pas de séparation entre déclaration et initialisation
// d'une variable constante
// const String str3;
// str3 = "Lol";

/* avec les Lists */
/* -------------- */
print("\nAvec les Lists");
print("--------------");
List<int> ints0 = [-1, 0];
```

```
print("ints0 = $ints0"); // ints0 = [-1, 0]

ints0 = [-3, -2];
ints0.add(-1);
print("ints0 = $ints0"); // ints0 = [-3, -2, -1]

const List<int> ints1 = [1,2];
print("ints1 = $ints1"); // ints1 = [1, 2]

// erreur à la compilation,
// pas de modification de la variable constante après initialisation
// ints1 = [2,5];

// pas d'erreur à la compilation,
// mais il y aura une erreur à l'exécution,
// parce que l'état de ints1 est immuable
// ints1.add(12);

List<int> ints2 = const[3,4];
print("ints2 = $ints2"); // ints2 = [3, 4]

// pas d'erreur à la compilation,
// mais il y aura une erreur à l'exécution
// parce que l'état de ints est immuable
// ints2.add(5);

// pas d'erreur,
// c'est la valeur qui est constante, pas la variable
ints2 = [5, 6];
print("ints2 = $ints2"); // ints2 = [5, 6]

/* avec les Maps */
/* ------------- */
print("\nAvec les Maps");
print("-------------");
var map0 = { "hey": "hola", "salut": 12 };
print("map0 = $map0"); // map0 = {hey: hola, salut: 12}

map0 = { "hey": "hola", "hi": 120 };
map0["hey"] = "Halo";
map0["salut"] = true;
print("map0 = $map0"); // map0 = {hey: Halo, hi: 120, salut: true}

const map1 = { "hey": "hello", "salut": 12 };
print("map1 = $map1"); // map1 = {hey: hello, salut: 12}

// erreur à la compilation
// pas de modification de la variable constante après initialisation
// map1 = { "hey": "hola", "hi": 120 };

// pas d'erreur à la compilation,
// mais il y aura une erreur à l'exécution,
// parce que l'état de map1 est immuable
// map1["hey"] = "Halo";
```

```
var map2 = const { "hey": "hi", "salut": 12 };
print("map2 = $map2"); // map2 = {hey: hi, salut: 12}

// pas d'erreur à la compilation,
// mais il y aura une erreur à l'exécution,
// parce que l'état de map2 est immuable
// map2["hey"] = "Halo";

// pas d'erreur,
// c'est la valeur qui est constante, pas la variable
map2 = { "hey": "hello", "hola": 50 };
print("map2 = $map2"); // map2 = {hey: hello, hola: 50}

/* cas particuliers */
/* ---------------- */
// pas d'erreur à la compilation,
// mais il y aura une erreur à l'exécution,
// parce que a n'est pas initialisée
// const a;
// print(a);

// erreur à la compilation,
// la valeur initiale affectée n'est pas constante,
// ni connue à la compilation
// const date = DateTime.now();

// ============================================================
print("\n============================================================");
/* FINAL */
/* ===== */
print("FINAL");
print("=====");
print("Avec les nombres");
print("----------------");
int x0 = 1;
print ("x0 = $x0"); // x0 = 1

final int x1 = 10;
print("x1 = $x1"); // x1 = 10

// erreur à la compilation,
// pas de modification de la variable finale après initialisation
// x1 = 20;

// pas d'erreur,
// pas de contraintes sur la valeur affectée à la variable finale
final int x2 = x0 + x1;
print("x2 = $x2"); // x2 = 11

// pas d'erreur,
// on peut séparer entre déclaration et initialisation
// d'une variable finale
final int x3;
```

```
    x3 = x1 + x2;
    print("x3 = $x3"); // x3 = 21

    /* cas particuliers */
    /* ---------------- */
    // pas d'erreur à la compilation,
    // mais il y aura une erreur à l'exécution,
    // parce que hey n'est pas initialisée
    // const hey;
    // print(hey);

    // erreur à la compilation,
    // la valeur initiale affectée n'est pas constante,
    // ni connue à la compilation
    // const date = DateTime.now();

    /* avec les Strings */
    /* ---------------- */
    print("\nAvec les Strings");
    print("----------------");
    final String str4 = "Salut";
    print("str4 = $str4"); // str4 = Salut

    // erreur à la compilation,
    // pas de modification après initialisation
    // str4 = "Bonjour";

    // pas d'erreur,
    // on peut déclarer et initialiser une variable finale non typée
    final str5 = "Buenos Dias";
    print("str5 = $str5"); // str5 = Buenos Dias

    // pas d'erreur,
    // on peut séparer entre déclaration et initialisation
    // d'une variable finale typée
    final String str6;
    str6 = "LOL";
    print("str6 = $str6"); // str6 = LOL

    // pas d'erreur,
    // on peut séparer entre déclaration et initialisation
    // d'une variable finale non typée
    final str7;
    str7 = "MDR";
    print("str7 = $str7"); // str7 = MDR

    // erreur à la compilation,
    // parce que str8 n'est pas initialisée
    // final String str8;
    // print(str8);

    /* avec les Lists */
    /* -------------- */
    print("\nAvec les Lists");
```

```
  print("--------------");
  final List<int> ints3 = [1,2];
  print("ints3 = $ints3"); // ints3 = [1, 2]

  // erreur à la compilation,
  // pas de modification de la variable finale après initialisation
  // ints3 = [2,5];

  // pas d'erreur,
  // il n'y a pas une nouvelle affectation effectuée
  // et l'état de ints3 n'est pas immuable
  ints3.add(12);
  print("ints3 = $ints3"); // ints3 = [1, 2, 12]

  /* avec les Maps */
  /* ------------- */
  print("\nAvec les Maps");
  print("-------------");
  final map3 = { "hey": "hello", "salut": 12 };
  print("map3 = $map3"); // map3 = {hey: hello, salut: 12}

  // erreur à la compilation
  // pas de modification de la variable finale après initialisation
  // map3 = { "hey": "hola", "hi": 120 };

  // pas d'erreur,
  // il n'y a pas une nouvelle affectation effectuée
  // et l'état de ints3 n'est pas immuable
  map3["hey"] = "hi";
  map3["hola"] = "Halo";
  print("map3 = $map3"); // map3 = {hey: hi, salut: 12, hola: Halo}

  /* Cas particuliers */
  /* ---------------- */
  print("\nCas particuliers");
  print("----------------");
  // pas d'erreur à la compilation,
  // mais il y aura une erreur à l'exécution,
  // parce que a n'est pas initialisée
  // final a;
  // print(a);

  // pas d'erreur
  final date = DateTime.now();
  print("date = $date"); // date = <date actuelle>
}
```

Nombres

Aperçu

1. un nombre est soit un entier soit un flottant à double précision.

Quelques propriétés et méthodes

propriété	description
`hashCode`	retourne le code de hachage du nombre courant
`isFinite`	retourne true si le nombre courant est fini
`isInfinite`	retourne true si le nombre courant est infini
`isNaN`	retourne true si le nombre courant ne désigne pas une valeur numérique
`isNegative`	retourne true si le nombre courant est négatif
`Sign`	retourne -1, 0, ou si le nombre courant est négatif, nul, ou positif respectivement

```dart
// ../source/dart/snippets/numbers/numbers_some_methods.dart
/*
* parse le input et retourne la valeur numérique correspondante.
* Possiblement, on peut définir une fonction dans le cas d'une
* erreur lors du parsing
* (i.e., si input ne désigne pas une valeur numérique.
*
* Remarque : Méthode de classe
*/
num parse(String input, [num Function(String)? onError]);

/*
* retourne la valeur absolute du nombre courant
*/
num abs();

/*
* retourne la valeur plafond du nombre courant
*/
int ceil();

/*
* retourne la valeur plancher du nombre courant
*/
int floor();

/*
* retourne la valeur entière avant la virgule du nombre courant
*/
int truncate();

/*
* retourne la valeur entière arrondie du nombre courant
*/
int round();

/*
* compare le nombra courant à other et retourne :
```

```
* 1. 1 si le nombre courant courant est plus grand que other
* 2. -1 si le nombre courant est plus petit que other
* 3. 0 si le nombre courant et other sont équivalents
*/
int compareTo(num other);

/*
* convertit le nombre courant en entier
*/
int toInt();

/*
* converti le nombre courant en double
*/
double toDouble();

/*
* retourne le reste de la division réelle
* du nombre courant par other
*/
num remainder(num other);
```

```
// ../source/dart/snippets/numbers/numbers_snippets.dart
void main() {
  num age = 23;
  print(age); // 23

  age = 23.5;
  print(age); // 23.5

  int ageEntier = 20;
  print(ageEntier); // 20

  // erreur à la compilation, ageEntier est de type entier
  // ageEntier = 20.5;

  double ageDouble = 20.5;
  print(ageDouble); // 20.5

  // pas d'erreur, conversion implicite de 20 en valeur double
  ageDouble = 20;
  print(ageDouble); // 20

  String number = "666";
  num n = num.parse(number);
  print(n); // 666

  number = "one";

  /*pas d'erreur à la compilation,
    mais il y aura une erreur
```

```
    à l'exécution avec la levée
    de l'exception FormatException
    parce que number ne désigne pas
    une valeur numérique
  */
  // n = num.parse(number);

  print("\nn = $n"); // n = 666
  print("Hash code: ${n.hashCode}"); // Hash code: n = <entier>
  print("Is finite: ${n.isFinite}"); // Is finite: true
  print("Is infinite: ${n.isInfinite}"); // Is infinite: false
  print("Is NaN: ${n.isNaN}"); // Is NaN: false
  print("Is negative: ${n.isNegative}"); // Is negative: false
  print("Sign: ${n.sign}"); // Sign: 1
  print("Absolute value: ${n.abs()}"); // Absolute value: 666
  n += 0.6;
  print("n += 0.6 = $n"); // n += 0.6 = 666.6
  print("Double value: ${n.toDouble()}"); // Double value: 666.6
  print("Ceil value: ${n.ceil()}"); // Ceil value: 667
  print("Floor value: ${n.floor()}"); // Floor value: 666
  print("Truncated value: ${n.truncate()}"); // Truncated value: 666
  print("Rounded value: ${n.round()}"); // Floor value: 666
  print("Int value: ${n.toInt()}"); // Int value: 666
  print("Compared to 665: ${n.compareTo(665)}"); // Compared to 665: 1
  print("Compared to 666.6: ${n.compareTo(666.6)}"); // Compared to 666.6:
0
  print("Compared to 668: ${n.compareTo(667)}"); // Compared to 667: -1

  print("Remainder of division by 333.3: ${n.remainder(333.3)}");
  // Remainder of division by 333.3: 0

  print("Remainder of division by 100: ${n.remainder(100)}");
  // Remainder of division by 100: 66.60000000000002
}
```

Strings

Aperçu

1. un string est une **chaîne de caractères** codée sur UTF-16 par défaut et circonscrite par :
   - des `''` ou `""` pour les strings monolignes.
   - des `"""` pour les strings multilignes (avec préservation des retours à la ligne).
2. les strings sont des objets immuables.

Quelques opérateurs

symbol	description	utilisation	remarques
`$/${}`	interpoler la valeur d'une variable/expression au sein d'un string	`$nomVariable` ou `${expression}`	-
`+`	concaténer des strings	str1 + str2	-

symbol	description	utilisation	remarques
`+=`	concaténation et affectation raccourcie	str1 += str2	équivaut str1 = str1 + str2ss
`==`	égalité logique entre deux strings	str1 == str2	-

Quelques propriétés et méthodes

propriété	description
`codeUnits`	retourne une liste immuable des codes UTF-16 des caractères du string courant
`isEmpty`	retourne true si le string courant est vide, false sinon
`length`	retourne le nombre de caractères du string courant (i.e., sa longueur)

```dart
// ../source/dart/snippets/strings/strings_some_methods.dart
/*
* retourne une copie du string courant mais
avec des caractères minuscules (avec aucun effet sur les chiffres)
*/
String toLowerCase();

/*
* retourne une copie du string courant mais
avec des caractères majuscules (avec aucun effet sur les chiffres)
*/
String toUpperCase();

/*
* retourne une copie du string courant mais
sans des caractères blancs au début et à la fin
*/
String trim();

/*
* retourne le code UTF-16 du caractère à l'indexe "index"
dans le string courant
*/
int codeUnitAt(int index);

/*
* retourne le sous-string du string courant en commençant par l'indice
"start" [et en terminant par l'indice "end - 1"]
si les indices sont valides,
sinon renvoie un string vide
*/
String substring(int start, [int? end]);

/*
* compare le string courant à other selon l'ordre naturel défini
* sur leurs caractères et retourne :
```

```
 * 1. une valeur positive si le string courant est plus grand que other
 * 2. une valeur négative si le string courant est plus petit que other
 * 3. zéro si le string courant et other sont équivalents
 */
int compareTo(String other);

/*
* remplace toutes les occurrences matchant
* from (Pattern obtenu depuis une regex)
* par le String replace dans le string courant
*/
String replaceAll(Pattern from, String replace);

/*
* coupe le string courant pour toute occurrrence
* matchant pattern et retourne la liste
* des sous-string obtenus
*/
List<String> split(Pattern pattern);
```

```
// ../source/dart/snippets/strings/strings_snippets.dart
void main() {
  String mono = "This is a single-line string";
  print("single-line string: $mono"); // using string interpolation
  print("length: ${mono.length}");
  print("is empty?: ${mono.isEmpty}\n");

  String multi = """This is a
  multiline
  string""";
  print("multiline string: \"$multi\"");
  print("length: ${multi.length}");
  print("is empty?: ${multi.isEmpty}\n");

  print(" \"\" is empty?: ${"".isEmpty}\n");

  print("UTF-16 code of character at index 2 in \"$mono\":
${mono.codeUnitAt(2)}");
  print("\"$mono\" in lowercase: ${mono.toLowerCase()}");
  print("\"$mono\" in uppercase: ${mono.toUpperCase()}");
  print("A substring of \"$mono\" starting at index 8 (included):
${mono.substring(8)}");
  print("A substring of \"$mono\" starting at index 10 (included) and
ending at index 16 (not included): ${mono.substring(10, 16)}");
  print("Replacing whitespace characters by _ in \"$mono\":
${mono.replaceAll(RegExp(r"\s"), "_")}");
  print("Comparing \"$mono\" with itself: ${mono.compareTo(mono)}");
  print("Comparing \"$mono\" with \"$multi\": ${mono.compareTo(multi)}");
  print("Comparing \"$multi\" with \"$mono\": ${multi.compareTo(mono)}");
  print("Splitting \"$mono\" on whitespace characters:
${mono.split(RegExp(r'\s'))}\n");
```

```
  String monoTrailingSpaces = "    " + mono; // using the concatenation
operator
  monoTrailingSpaces += "    "; // using the shortcut assignment and
concatenation operator

  print("string with trailing spaces: \"$monoTrailingSpaces\"");
  print("string with trailing spaces after trimming:
\"${monoTrailingSpaces.trim()}\"\n");
}
```

Entrées/Sorties standards

Aperçu

1. `Stdin`: une classe représentant l'entrée standard (*clavier, par défaut*) et permettant d'interagir avec elle d'une façon (a)synchrone.
2. `Stdout`: une classe représentant la sortie standard (*écran, par défaut*) et permettant d'interagir avec elle.
3. `Stderr`: une classe représentant la sortie standard des erreurs (*écran, par défaut*) et permettant d'interagir avec elle.
4. `dart:io` : une bibliothèque à importer pour utiliser les entrées/sorties.

Quelques méthodes

```
// ../source/dart/snippets/io/io_some_methods.dart
/*
* lit d'une façon synchrone (bloquante) une ligne
* saisie via l'entrée standard
* et la retourne si elle n'est pas null
*/
String? readLineSync();

/*
* lit d'une façon synchrone (bloquante) un octet
* saisi via l'entrée standard
* et le retourne s'il n'est pas null
*/
int readByteSync();

/*
* écrit object.toString() d'une façon asynchrone
* sur la sortie standard (ou la sortie standard des erreurs)
* sans retour à la ligne
*/
void write(Object? object);

/*
* écrit object.toString() d'une façon asynchrone
* sur la sortie standard (ou la sortie standard des erreurs)
```

```
 * avec retour à la ligne
 */
void writeln(Object? object);
```

```
// ../source/dart/snippets/io/io_snippet.dart
import 'dart:io';

void main(){
  print("Please enter your name: ");
  String? name = stdin.readLineSync();

  stdout.write("Hello $name"); // affichage sans retour à la ligne
  print("Hello $name"); // affichage avec retour à la ligne
}
```

```
// ../source/dart/examples/io/sum.dart
import 'dart:io';

void main(){
  print("Welcome to a simple addition program");
  print("====================================");
  print("Enter your first number:");
  int? n1 = int.parse(stdin.readLineSync()!); // ? and ! for null safety

  print("Enter your second number:");
  int? n2 = int.parse(stdin.readLineSync()!);

  print("$n1 + $n2 = ${n1 + n2}");
}
```

Instructions conditionnelles

if, else if, else

```
// ../source/dart/snippets/conditionals/if_elseif_else.dart
if (expressionBooleene1){
  /*code si <expressionBooleene1> vaut true*/
}

else if (<expressionBooleene2>){
  /*code si <expressionBooleene2> vaut true*/
}
...
else {
  /*code sinon*/
}
```

```dart
// ../source/dart/examples/conditionals/if_elseif_else_age.dart
import 'dart:io';

void main(){
  print("Veuillez saisir votre age :");
  int? age = int.parse(stdin.readLineSync()!);

  if (age > 0 && age < 12){
    print("Enfant");
  }
  else if (age >=12 && age < 18){
    print("Ado");
  }
  else if (age >=18 && age < 30){
    print("Jeune Adulte");
  }
  else if (age >=30 && age < 70){
    print("Adulte");
  }
  else if (age >=70 && age < 120){
    print("Vieux");
  }
  else {
    print("Vous n'êtes pas vivant !");
  }

  print("End of program");
}
```

switch

```dart
// ../source/dart/snippets/conditionals/switch.dart
switch(expression){ // integer | string
  case n:
    /*code si expression == n*/
    break;
  case m:
    /*code si expression == m*/
    break;
  ...
  default:
    /*code sinon*/
}
```

```dart
// ../source/dart/examples/conditionals/favorite_color_switch.dart
import 'dart:io';
```

```dart
void main(){
  print("Veuillez choisir votre couleur préférée parmi les valeurs suivantes :");
  print("1. Rouge");
  print("2. Bleu");
  print("3. Vert");
  print("4. Jaune");
  print("5. Noir");
  print("6. Blanc");
  print("7. Marron");
  print("8. Violet");
  print("9. Gris");
  print("10. Rose");

  int? choix = int.tryParse(stdin.readLineSync()!); // retourne null si on a une exception lors du parsing
  String couleur = "rien";

  // using if/else..if/else
  print("\nusing if/else..if/else");
  print("------------------------");
  if (choix == 1){
    couleur = "Rouge";
  }
  else if (choix == 2){
    couleur = "Bleu";
  }
  else if (choix == 3){
    couleur = "Vert";
  }
  else if (choix == 4){
    couleur = "Jaune";
  }
  else if (choix == 5){
    couleur = "Noir";
  }
  else if (choix == 6){
    couleur = "Blanc";
  }
  else if (choix == 7){
    couleur = "Marron";
  }
  else if (choix == 8){
    couleur = "Violet";
  }
  else if (choix == 9){
    couleur = "Gris";
  }
  else if (choix == 10){
    couleur = "Rose";
  }
  else{
    stderr.writeln("Erreur : votre choix ne correspond à aucune couleur dans la liste");
```

```
  }

  print("Votre couleur préférée est $couleur");

  print("\nusing switch");
  print("--------------");
  switch(choix){
    case 1:
      couleur = "Rouge";
      break;
    case 2:
      couleur = "Bleu";
      break;
    case 3:
      couleur = "Vert";
      break;
    case 4:
      couleur = "Jaune";
      break;
    case 5:
      couleur = "Noir";
      break;
    case 6:
      couleur = "Blanc";
      break;
    case 7:
      couleur = "Marron";
      break;
    case 8:
      couleur = "Violet";
      break;
    case 9:
      couleur = "Gris";
      break;
    case 10:
      couleur = "Rose";
      break;
    default:
      stderr.writeln("Erreur : votre choix ne correspond à aucune couleur
dans la liste");
  }

  print("Votre couleur préférée est $couleur");
  print("\nFin du programme");
}
```

Opérateurs conditionnels

symbol	description	utilisation	remarques

symbol	description	utilisation	remarques
`? :`	si `condition` est true, retourne `expression1`, sinon retourne `expression2`	`condition? expression1 : expression2`	-
`??`	si `expression1` n'est pas null, la retourne, sinon retourne `expression2`	`expression1 ?? expression2`	-

```dart
// ../source/dart/snippets/conditionals/conditional_operators.dart
import 'dart:io';

void main(){

  print("Pair ou impair");
  print("--------------");
  print("Veuillez saisir un entier");
  int? n = int.parse(stdin.readLineSync()!);
  String statut = "";

  // using if/else
  print("\nusing if/else");
  print("==============");
  if (n%2 == 0){
    statut = "pair";
  }
  else {
    statut = "impair";
  }

  print("$n est $statut\n");

  // using conditional operators
  print("using ternary conditional operator");
  print("==================================");
  statut = (n%2 == 0)? "pair" : "impair";
  print("$n est $statut");

  print("\nValeur par défaut en cas d'un null");
  print("----------------------------------");
  print("Veuillez saisir n'importe quel expression");
  String? value = stdin.readLineSync() ?? null;

  // using if/else
  print("\nusing if/else");
  print("==============");
  if (value != null){
    statut = value;
  }
  else {
    statut = "N/A";
  }
```

```
  print("Valeur saisie = $value");
  print("Statut = $statut");

  // using conditional operators
  print("\nusing binary conditional operator");
  print("================================");
  statut = value ?? "N/A";
  print("Valeur saisie = $value");
  print("Statut = $statut");

  print("\nFin du programme");
}
```

Boucles

La boucle while

```
// ../source/dart/snippets/loops/while.dart
while(expressionBooleenne){
  /*code*/
}
```

La boucle do while

```
// ../source/dart/snippets/loops/do_while.dart
do {
  /*code*/
} while(expressionBooleenne);
```

Les boucles for

```
// ../source/dart/snippets/loops/for.dart
/*La boucle for généralisée*/
for (/*initialisation*/; /*condition d'arrêt*/; /*changement d'état*/){
  /*code*/
}

/*La boucle for adaptée aux objets itérables*/
for (var element in iterateurObjetIterable){
  /*code*/
}
```

Instructions de rupture des boucles

1. `continue` : sauter à la **prochaine itération** dans la **boucle**.
2. `break` : **sortir** de la **boucle**.

```dart
// ../source/dart/examples/loops/loops_lists_maps.dart
void main() {
  var l = [1, 9, 4, -5, 15, 20, 100, 5, 3, 0, 10];
  var m = { "id" : 1, "firstName" : "John", "lastName" : "Doe"};
  int i = 0;

  /* avec les listes */
  print("Parcourir des listes");
  print("====================");
  // while loop
  print("Avec la boucle while");
  print("--------------------");
  while (i < l.length){
    print("l[$i] = ${l[i]}");
    i++;
  }

  // reset i
  i = 0;

  // do..while loop
  print("\nAvec la boucle do...while");
  print("--------------------------");
  do {
    print("l[$i] = ${l[i]}");
    i++;
  } while (i < l.length);


  // generalized for loop
  print("\nAvec la boucle for généralisée");
  print("------------------------------");

  // reset i
  for (i = 0; i < l.length; i++){
    print("l[$i] = ${l[i]}");
  }

  // for..in loop
  print("\nAvec la boucle for..in");
  print("----------------------");

  for (var element in l){
    print(element);
  }

  /* avec les maps */
  print("\nParcourir des maps");
  print("====================");
```

```
  // while loop
  print("Avec la boucle while");
  print("--------------------");

  // reset i
  i = 0;
  String key = "";
  while (i < m.keys.length){
    key = m.keys.elementAt(i);
    print("m[$key] = ${m[key]}");
    i++;
  }

  // do..while loop
  print("\nAvec la boucle do...while");
  print("--------------------------");

  // reset i
  i = 0;

  do {
    key = m.keys.elementAt(i);
    print("m[$key] = ${m[key]}");
    i++;
  } while (i < m.keys.length);

  // generalized for loop
  print("\nAvec la boucle for généralisée");
  print("------------------------------");

  // reset i
  for (i = 0; i < m.keys.length; i++) {
    key = m.keys.elementAt(i);
    print("m[$key] = ${m[key]}");
  }

  // for..in loop
  print("\nAvec la boucle for..in");
  print("----------------------");

  for (var key in m.keys){
    print("m[$key] = ${m[key]}");
  }
}
```

Listes

Aperçu

1. **définition** : une liste est une collection ordonnée d'objets accessible par des indexes numériques.
2. **features** : une liste :
    - possède une taille fixe ou dynamique.

- permet des valeurs d'un seul type ou de type dynamique.
- permet les valeurs dupliquées et `null`.
- ne peut pas être modifiée lors de son parcours.

Quelques propriétés et méthodes

propriété	description
`hashCode`	retourne le code de hachage de la liste courante
`length`	retourne le nombre d'éléments de la liste courante
`isEmpty`	retourne true si la liste courante est vide, false sinon
`isNotEmpty`	retourne true is la liste courante n'est pas vide, false sinon
`first`	retourne le premier élément de la liste courante si elle n'est pas vide, sinon lance une exception
`last`	retourne le dernier élément de la liste courante si elle n'est pas vide, sinon lance une exception
`single`	si la liste courante consiste d'un seul élément, le retourne, sinon lance une exception
`reversed`	retourne un itérateur sur l'ordre inverse de la liste courante

```dart
// ../source/dart/snippets/lists/lists_some_methods.dart
/**
 * crée une liste de taille "length" (dynamique par défaut).
 * chaque élément est généré en utilisant
 * la fonction paramétré par l'indexe
 * de chaque élément et retournant une expression.
 *
 * Remarque : méthode de classe
 */
List List.generate(int length, (int index) => <expression>, {bool growable:
true});

/**
 * crée une liste de taille "length" (fixe par défaut).
 * chaque élément possède la valeur fill
 *
 * Remarque : méthode de classe
 */
List List.filled(int length, E fill, {bool growable: false});

/**
 * crée une liste vide de taille fixe par défaut
 */
List List.empty({bool growable: false});

/**
 * crée une liste vide de taille dynamique et de type E
```

```
 */
var liste = <E>[];

/**
 * accès en lecture/écriture au "indexe"ième élément
 * de la liste courante.
 * Remarque : selon le cas, l'opérateur en écriture
 * peut lancer une exception
 */
liste[indexe];

/**
 * retourne le "indexe"ième élément si celui-ci
 * est accessible, sinon lance un exception
 */
<E> elementAt(int indexe);

/**
 * ajoute l'élément element à la fin de la liste
 */
void add(E element);

/**
 * ajoute les éléments de iterable à la fin de la liste
 */
void addAll(Iterable<E> iterable);

/**
 * si l'indexe est valide, insère l'élément à l'indexe "indexe"
 * et pousse l'ancien élément et les éléments qui le suivent à droite.
 * sinon, lance une exception
 */
void insert(int indexe, E element);

/**
 * si l'indexe est valide, insère les éléments de iterable à l'indexe
"indexe"
 * et pousse l'ancien élément et les éléments qui le suivent à droite.
 * sinon, lance une exception
 */
void insertAll(int indexe, Iterable<E> iterable);

/**
 * si start et end sont valides et la taille de iterable est valide,
 * remplace les éléments dans l'intervalle [start (inclus), end (exclus)]
 * par les éléments de iterable.
 * sinon lance une exception
 */
void replaceRange(int start, int end, Iterable<E> iterable);

/**
 * supprime la première occurrence de element dans la liste.
 * retourne true si element était dans la liste et a été supprimé,
 * false sinon
```

```
 */
bool remove(E? element);

/**
 * si l'indexe est valide, supprime l'élément à l'indexe "indexe"
 * et le retourne.
 * sinon, lance une exception
 */
E removeAt(int index);

/**
 * si la liste est non vide, supprime le dernier élément
 * et le retourne.
 * sinon, lance une exception
 */
E removeLast();

/**
 * si start et end sont valides, supprime les éléments dans
 * l'intervalle [start (inclus), end (exclus)]
 * sinon lance une exception
 */
void removeRange(int start, int end);

/**
 * vide tout le contenu de la liste courante
 */
void clear();

/**
 * retourne true si la liste contient element,
 * false sinon
 */
bool contains(Object? element);

/**
 * itère sur chaque élément "element" de la liste et applique
 * la fonction action paramétrée par l'élément
 */
void forEach(void Function(E element) action);

/**
 * retourne un ensemble contenant les éléments de la liste courante
 * (supprime les éléments dupliqués)
 */
Set<E> toSet();
```

```
// ../source/dart/snippets/lists/lists_snippets.dart
void main() {
  // une liste vide avec une taille fixe
  List empty  = List.empty();
```

```
    listDetails(empty);

    // erreur d'accès, car la liste est vide et sa taille est fixe
    // empty[0] = 12;
    // empty[1] = 15;
    // empty[3] = 30;
    // empty.add(10);
    // listDetails(empty);

    // une liste vide avec une taille dynamique
    List empty2 = [];
    listDetails(empty2);

    // erreur d'accès car la liste est vide
    // empty2[0] = 12;
    // empty2[1] = 15;
    // empty2[3] = 30;

    // pas d'erreur, car la taille est dynamique
    empty2.add(10);
    listDetails(empty2);

    // plus d'erreur d'accès
    print("empty2[0] = ${empty2[0]}\n");

    // une liste vide avec une taille dynamique
    empty  = List.empty(growable: true);
    listDetails(empty);

    // erreur d'accès car la liste est vide
    // empty[0] = 12;
    // empty[1] = 15;
    // empty[2] = 30;

    // pas d'erreur, car la taille est dynamique
    empty.add(10);
    listDetails(empty);

    // plus d'erreur d'accès
    print("empty[0] = ${empty.elementAt(0)}\n");

    // une liste de 10 strings générés avec "N/A" comme valeur par défaut
    List<String> names = List.generate(10, (index) => "N/A");
    listDetails(names);

    // vider la liste
    names.clear();
    listDetails(names);

    // réinitialiser la liste
    names = List.generate(10, (index) => "N/A");
    listDetails(names);

    // mettre à jour et ajouter de nouveaux éléments
```

```
  names[0] = "John";
  names[2] = "Alice";
  names.addAll(["Bob", "Daniel"]);
  names.insert(1, "Tom");
  names.insertAll(5, ["Jessica", "Donald"]);
  names.replaceRange(6, 8, ["Joe", "Peter"]);
  listDetails(names);

  // une liste de 10 entiers générés avec "-1" comme valeur par défaut
  List<int> ints = List.filled(10, -1);
  listDetails(ints);

  // une liste d'entiers générés de 1 à 10
  ints = List.generate(10, (index) => index + 1);
  listDetails(ints);

  // supprimer quelques éléments de la liste
  ints.remove(2);
  ints.removeAt(5);
  ints.removeLast();
  ints.removeRange(4, 6);
  listDetails(ints);

  List<double> doubles = [1.0, 25.5, 86.6, -12.5, 12, 0, -12.5, -45, 11.8];
  print("$doubles");
  print("iterating using a for in loop:");
  for (var e in doubles){
    print(e);
  }

  double sum = 0.0;
  doubles.forEach((dbl) => sum += dbl);
  print("\nsum: $sum");

}

void listDetails(List<dynamic> list){
  print(list);
  print("length: ${list.length}"); // nombre d'éléments
  print("is empty: ${list.isEmpty}"); // true si vide, false sinon
  print("is not empty: ${list.isNotEmpty}"); // true si non vide, false
sinon
  if (list.isNotEmpty){
    print("first: ${list.first}"); // premier élément
    print("last: ${list.last}"); // dernier élément
  }
  print("reversed: ${list.reversed.toList()}\n"); // reversed : itérateur
inverse
}
```

Sets

Aperçu

1. **définition** : un ensemble est une collection non ordonnée d'objets accessible par des indexes numériques.
2. **features** : un ensemble :
    - possède une taille fixe ou dynamique.
    - permet des valeurs d'un seul type ou de type dynamique.
    - permet les valeurs `null`;
    - ne permet pas les valeurs dupliquées.
    - ne peut pas être modifiée lors de son parcours.

Quelques propriétés et méthodes

propriété	description
`hashCode`	retourne le code de hachage de l'ensemble courant
`length`	retourne le nombre d'éléments de l'ensemble courant
`isEmpty`	retourne true si l'ensemble courant est vide, false sinon
`isNotEmpty`	retourne true is l'ensemble courant n'est pas vide, false sinon
`first`	retourne le premier élément de l'ensemble courant s'il n'est pas vide, sinon lance une exception
`last`	retourne le dernier élément de l'ensemble courant s'il n'est pas vide, sinon lance une exception
`single`	si l'ensemble courant consiste d'un seul élément, le retourne, sinon lance une exception

```dart
// ../source/dart/snippets/sets/sets_some_methods.dart
/**
 * crée un ensemble vide de taille dynamique et de type E
 */
var s = <E>{};

/**
 * crée un ensemble de taille dynamique et contenant des éléments de type E
 */
var s = <E>{e1, e2, ...};

/**
 * retourne le "indexe"ième élément si celui-ci
 * est accessible, sinon lance un exception
 */
<E> elementAt(int indexe);

/**
 * ajoute l'élément element à la fin de l'ensemble
 */
void add(E element);
```

```dart
/**
 * ajoute les éléments de iterable à la fin de l'ensemble
 */
void addAll(Iterable<E> iterable);

/**
 * supprime l'element dans l'ensemble.
 * retourne true si element était dans l'ensemble et a été supprimé,
 * false sinon
 */
bool remove(E? element);

/**
 * supprime tous les éléments de iterable dans l'ensemble
 */
void removeAll(Iterable<E> iterable);

/**
 * supprime tous les éléments de l'ensemble matchant
 * la fonction de test
 */
void removeWhere(bool Function(E element) test);

/**
 * vide tout le contenu de l'ensemble courant
 */
void clear();

/**
 * retourne true si l'ensemble contient element,
 * false sinon
 */
bool contains(Object? element);

/**
 * retourne true si l'ensemble contient tous
 * les éléments de iterable,
 * false sinon
 */
bool containsAll(Iterable<Object?> iterable);

/**
 * itère sur chaque élément "element" de l'ensemble et applique
 * la fonction action paramétrée par l'élément
 */
void forEach(void Function(E element) action);

/**
 * retourne une liste contenant les éléments de l'ensemble courant
 */
List<E> toList();

/**
```

```
 * retourne l'intersection de l'ensemble courant avec other
 * (i.e., les éléments en commun)
 */
Set<E> intersection(Set<Object?> other);

/**
 * retourne l'union de l'ensemble courant avec other
 * (i.e., les éléments dans les deux)
 */
Set<E> union(Set<E> other);

/**
 * retourne la différence de l'ensemble courant avec other
 * (i.e., les éléments dans l'ensemble courant mais non pas dans other)
 */
Set<E> difference(Set<Object?> other);
```

```
// ../source/dart/snippets/sets/sets_snippets.dart
void main() {
  // un ensemble vide de taille et contenu dynamiques
  var s = <dynamic>{};

  // ajout de quelques éléments
  s.add(10);
  s.add(11);
  s.add("lol");
  s.add("lol"); // n'ajoutera rien (dupliquée)
  s.add(null);
  s.addAll([-1, true, [12, 13], "hello"]);
  s.addAll({80, 50, 12});
  print("s = $s");
  setDetails(s);

  // vide le contenu de l'ensemble
  s.clear();
  print("s = $s (après vidage de son contenu)");
  setDetails(s);

  // un ensemble d'entiers de taille dynamique
  var s2 = {1, 2, 3, 4};
  print("\ns2 = $s2");
  print("s2[1] = ${s2.elementAt(1)}");
  print("s2.contains(4): ${s2.contains(4)}");
  print("s2.contains(0): ${s2.contains(0)}");
  print("s2.containsAll([2, 4]): ${s2.containsAll([2, 4])}");

  // supprime quelques éléments de l'ensemble
  s2.remove(3);
  print("\ns2 = $s2 (après suppression de 3)");
  s2.removeAll([2, 4]);
  print("s2 = $s2 (après suppression de 2 et 4)");
```

```
  s2.addAll([5, 6, 7, 8, 9]);
  print("s2 = $s2 (après ajout de 5, 6, 7, 8, 9)");
  s2.removeWhere((element) => element.isEven);
  print("s2 = $s2 (après suppression des entiers pairs)");

  // convertit l'ensemble en une liste
  print("liste contenant les éléments de s2: ${s2.toList()}\n");

  print("s2 = $s2");
  setDetails(s2);

  int sum = 0;
  s2.forEach((element) => sum += element);
  print("somme des éléments de $s2 = $sum");

  var s3 = {1, 2, 3, 5};
  print("\ns3 = $s3");
  setDetails(s3);

  print("\n$s2 intersection $s3 = ${s2.intersection(s3)}");
  print("$s2 union $s3 = ${s2.union(s3)}");
  print("$s2 difference $s3 = ${s2.difference(s3)}");
}

void setDetails(Set<dynamic> s){
  print(s);
  print("length: ${s.length}"); // nombre d'éléments
  print("is empty: ${s.isEmpty}"); // true si vide, false sinon
  print("is not empty: ${s.isNotEmpty}"); // true si non vide, false sinon
  if (s.isNotEmpty){
    print("first: ${s.first}"); // premier élément
    print("last: ${s.last}"); // dernier élément
  }
}
```

Maps

Aperçu

1. **définition** : un map est dictionnaire de paires clé/valeur.
2. **features** : un map :
    - possède une taille fixe ou dynamique.
    - permet des clés/valeurs de types statique ou dynamique.
    - ne permet pas les clés dupliquées.
    - permet des clés ou valeurs `null`;
    - ne peut pas être modifiée lors de son parcours.

Quelques propriétés et méthodes

propriété	description

propriété	description
`hashCode`	retourne le code de hachage du map courant
`length`	retourne le nombre d'éléments du map courant
`isEmpty`	retourne true si le map courant est vide, false sinon
`isNotEmpty`	retourne true is le map courant n'est pas vide, false sinon
`keys`	retourne un itérable sur les clés du map courant
`values`	retourne un itérable sur les valeurs du map courant
`entries`	retourne un itérable sur les entrées clé/valeur du map courant

```dart
// ../source/dart/snippets/maps/maps_some_methods.dart
/**
 * crée un map vide de taille et de contenu dynamiques
 */
var m = {};

/**
 * crée un map de taille dynamique et contenant des clés de type E et
valeurs de type V
 */
var m = <E, V>{k1: v1, k2: v2, ...};

/**
 * accès en lecture/écriture à la valeur associée à la clé "cle"
 * du map courant.
 * si cle n'est pas une clé du map, retourne null
 */
liste[cle];

/**
 * ajoute les paires clé/valeur de other dans le map courant.
 * si other contient une clé déjà existante dans le map courant,
 * la valeur de other écrase l'ancienne valeur.
 */
void addAll(Map<K, V> other);

/**
 * ajoute les paires clé/valeur de l'itérable newEntries dans le map
courant.
 * si newEntries contient une clé déjà existante dans le map courant,
 * la valeur associée dans newEntries écrase l'ancienne valeur.
 */
void addEntries(Iterable<MapEntry<K, V>> newEntries);

/**
 * supprime la clé "cle" dans le map courant.
 * retourne la valeur associée à "cle" si celle-ci est une clé du map
courant
```

```
 * null sinon
 */
V? remove(K? cle);

/**
 * supprime tous les paires clé/valeur du map matchant
 * la fonction de test
 */
void removeWhere(bool Function(K cle, V valeur) test);

/**
 * vide tout le contenu du map courant
 */
void clear();

/**
 * retourne true si key est une clé du map courant
 * false sinon
 */
bool containsKey(Object? key);

/**
 * retourne true si value est une valeur du map courant
 * false sinon
 */
bool containsValue(Object? value);

/**
 * itère sur chaque paire clé/valeur du map courant et applique
 * la fonction action paramétrée par key (clé) et value (valeur)
 */
void forEach(void Function(K key, V value) action);
```

```
// ../source/dart/snippets/maps/maps_snippets.dart
void main(){
  // un map vide de taille et contenu dynamiques
  var m1 = {};
  mapDetails(m1);

  m1 = {"id": "jkhsdq537qsd", "email": "someuser@example.com"};
  mapDetails(m1);

  // vider le contenu du map
  m1.clear();
  mapDetails(m1);

  // un map de clés String et de valeurs int
  var m2 = <String, int>{"John": 1, "Alice": 2};
  mapDetails(m2);

  // ajouter des paires clé/valeur et modifier des valeurs existantes avec
```

```
[]
  m2["Bob"] = 30;
  m2["Alice"] = 15;
  mapDetails(m2);

  print("m2['Alice'] = ${m2["Alice"]}");
  print("m2['Bob'] = ${m2["Bob"]}");
  print("m2['Steve'] = ${m2["Steve"]}\n");

  // ajouter des paires clé/valeur avec addAll()
  m2.addAll({'Steve': 100, 'Peter': 50});
  mapDetails(m2);

  // ajouter des paires clé/valeurs avec addEntries()
  m2.addEntries({'Tom': 15, 'Nancy': 89}.entries);
  mapDetails(m2);

  // supprimer quelques paires
  print("ancienne valeur associée à la clé 'Tom': ${m2.remove("Tom")}");
  print("ancienne valeur associée à la clé 'Christina':
${m2.remove("Christina")}\n");
  mapDetails(m2);

  m2.removeWhere((key, value) => value.isEven);
  print("après suppression des paires ayant des valeurs pairs");
  mapDetails(m2);

  print("m2 contains la clé 'John': ${m2.containsKey("John")}");
  print("m2 contains la clé 'Christina': ${m2.containsKey("Christina")}");
  print("m2 contains la valeur 89: ${m2.containsValue(89)}");
  print("m2 contains la valeur 100: ${m2.containsValue(100)}");

  print("\nparcours de $m2 avec la méthode forEach()");
  m2.forEach((key, value) => print("$key -> $value"));

  var data = [
    {
      'title': 'Q1',
      'answers': [
        {
          'title': 'A11',
          'correct': true
        },
        {
          'title': 'A12',
          'correct': false
        }
      ]
    },
    {
      'title': 'Q2',
      'answers': [
        {
          'title': 'A21',
```

```
            'correct': false
          },
          {
            'title': 'A22',
            'correct': true
          }
        ]
      }
    ];

    data.forEach((question) {
      print(question['title']);
      (question['answers'] as List<dynamic>).forEach((answer){
        print("\t${answer['title']}: ${answer['correct']}");
      });
    });
}

void mapDetails(Map<dynamic, dynamic> map){
  print(map);
  print("length: ${map.length}"); // nombre de paires clé/valeur
  print("clés: ${map.keys.toSet()}"); // ensemble des clés
  print("valeurs: ${map.values.toList()}"); // liste des valeurs
  print("entrées: ${map.entries.toList()}"); // liste des paires clé/valeur
  print("is empty: ${map.isEmpty}"); // true si vide, false sinon
  print("is not empty: ${map.isNotEmpty}\n"); // true si non vide, false
sinon
}
```

```
// ../source/dart/examples/maps/encoding_decoding_maps_example.dart
import "dart:convert";

void main(){
  var data = """[
    {
      "id": 1,
      "name": "John"
    },
    {
      "id": "2",
      "name": "Alice"
    }
  ]""";

  print("data = $data\n");

  var parsedData = json.decode(data); // transforme le string en objet
dynamique correspondant

  for (var d in parsedData){
    print("id: ${d['id']}");
```

```
    print("name: ${d['name']}\n");
  }

  var john = json.encode(parsedData[0]); // transforme un objet personne en
string correspondant
  print(john);

  var alice = json.encode(parsedData[1]);
  print(alice);
}
```

Fonctions

Aperçu

1. **définition** : une fonction est un ensemble d'instructions réutilisable et éventuellement paramétrable, possiblement avec une valeur de retour.
2. les paramètres des fonctions peuvent être **obligatoires** ou **optionnels**.
3. les paramètres optionnels sont soit **positionnels** (*circonscrits par des `[]` lors de la déclaration de la fonction*) soit **nommés** (*circonscrits par des `[]` lors de la déclaration de la fonction*)

```
// ../source/dart/snippets/functions/functions_syntax.dart
// définition d'une procédure sans paramètres
void nomProcedure(){
  // statements;

  // on peut possiblement sortir de la procédure avec return
  // return;
}

// invocation de nomProcedure
nomProcedure();

// définition d'une fonction sans paramètres avec valeur de retour
ReturnType nomFonction1(){
  // statements;

  return expression; // expression est de type "ReturnType"
}

// invocation de nomFonction1
ReturnType result = nomFonction1();

// définition d'une fonction avec paramètres obligatoires
ReturnType nomFonction2(Type1 param1, Type2 param2, ...){
  // statements;

  return expression; // expression est de type "ReturnType"
}

// invocation de nomFonction2
```

```
Type1 arg1 = <expression>;
Type2 arg2 = <expression>;
...

ReturnType result = nomFonction2(arg1, arg2, ...);

// définition d'une fonction avec paramètres obligatoires
// et paramètres positionnels optionnels
ReturnType nomFonction3(Type1 param1, Type2 param2, [Type3 param3, Type4
param4]){
  // statements;

  return expression; // expression est de type "ReturnType"
}

// invocation de nomFonction3
Type1 arg1 = <expression>;
Type2 arg2 = <expression>;
Type3 arg3 = <expression>;
Type4 arg4 = <expression>;

ReturnType result = nomFonction3(arg1, arg2);
result = nomFonction3(arg1, arg2, arg3);
result = nomFonction3(arg1, arg2, arg3, arg4);

// définition d'une fonction avec paramètres obligatoires
// et paramètres nommé optionnels
ReturnType nomFonction4(Type1 param1, Type2 param2, {Type3 param3, Type4
param4}){
  // statements;

  return expression; // expression est de type "ReturnType"
}

// invocation de nomFonction4
Type1 arg1 = <expression>;
Type2 arg2 = <expression>;
Type3 arg3 = <expression>;
Type4 arg4 = <expression>;

ReturnType result = nomFonction4(arg1, arg2);
result = nomFonction3(arg1, arg2, param3: arg3);
result = nomFonction3(arg1, arg2, param4: arg4, param3: arg3);

// définition d'une fonction avec paramètres obligatoires
// et paramètres nommé optionnels et valeurs par défaut
ReturnType nomFonction5(Type1 param1, Type2 param2, {Type3 param3, Type4
param4 = <expression>}){
  // statements;

  return expression; // expression est de type "ReturnType"
}

// invocation de nomFonction5
```

```
Type1 arg1 = <expression>;
Type2 arg2 = <expression>;
Type3 arg3 = <expression>;
Type4 arg4 = <expression>;

ReturnType result = nomFonction5(arg1, arg2);
result = nomFonction3(arg1, arg2, param3: arg3);
result = nomFonction3(arg1, arg2, param4: arg4, param3: arg3);

// définition d'une fonction monoligne
ReturnType nomFonction6(parameters_list) => { /* statements */}

// définition d'une fonction anonyme/lambda
return_type (parametre_lists) {
  // statements
}

// définition d'une fonction anonyme en utilisant l'opérateur =>
(parametre_lists) => { /* statements */}

// affectation d'une fonction anonyme/lambda à une variable
var fonctionAnonyme = (parametre_lists) => { /* statements */}
```

```
// ../source/dart/snippets/functions/lexical_scope_closures.dart
var topLevel = 1;

void main(){
  var mainLevel = 2;

  // nested function
  void nestedFunction(){
    var nestedLevel = 3;

    // nested function
    void nestedFunction2(){
      var nestedLevel2 = 4;

      print("top level: $topLevel");
      print("main level: $mainLevel");
      print("nested level: $nestedLevel");
      print("nested level 2: $nestedLevel2");
    }

    nestedFunction2();

    // erreur car la variable n'existe pas dans cette portée
    // print(nestedLevel2);
  }

  nestedFunction();
```

```
  // erreur car la fonction n'existe pas dans cette portée
  // nestedFunction2();

  // erreur car la variable n'existe pas dans cette portée
  // print(nestedLevel);
}

// erreur car la fonction n'existe pas dans cette portée
// nestedFunction();

// erreur car la variable n'existe pas dans cette portée
// print(mainLevel);
```

```
// ../source/dart/snippets/functions/functions_snippets.dart
void main() {
  var c = sum1(30, 20);
  var d = sum2(30, 20);
  print("c = 30 + 20 = $c");
  print("d = 30 + 20 = $d");

  // définition d'une fonction anonyme et affectation à une variable
  var sum3 = (int a, int b) => a + b;
  var e = sum3(30, 20);
  print("e = 30 + 20 = $e");

  // utilisation d'une fonction anonyme avec forEach
  [10, 11, 15, 18].forEach((item) => print("$item est pair:
${item.isEven}"));

  print(factorial(6)); // 720

}

// définition classique d'une fonction
int sum1(int a, int b){
  return a + b;
}

// définition d'une fonction monoligne
int sum2(int a, int b) => a + b;

int factorial(int n){
  return (n <= 1)? 1 : n * factorial(n - 1);
}
```

Programmation orientée-objet

Classes, objets, attributs, méthodes, constructeurs

1. en Dart, tout est objet, et tout type hérite de la classe `Object`.

2. tout objet est instancié à partir d'une classe définissant ses attributs et ses méthodes.
3. les attributs/méthodes sont privés dans la bibliothèque (*library*) contenant leur classe si leurs noms commencent par `_`.
4. par défaut, un constructeur est fourni par le compilateur, sinon on peut définir explicitement plusieurs constructeurs (*sans paramètres, paramètrés, nommés*).
5. `this` désigne l'instance courante dans le bloc courant (e.g., constructeur, méthode) et permet l'accès aux attributs et méthodes de celle-ci.

```
// ../source/dart/snippets/oop/classes_syntax.dart
// définition d'une classe
class ClassName {
  <fields>
  <getters/setters>
  <constructors>
  <methods>
}

// instanciation d'un(e) objet/instance à partir d'une classe concrète (non
abstraite)
ClassName obj = new ClassName(<constructor_args>);
```

```
// ../source/dart/snippets/oop/constructors_syntax.dart
class ClassName {

  // constructeur sans arguments (par défaut)
  ClassName(){
    // statements
  }

  // constructeur avec des paramétres positionnels obligatoires
  ClassName(Type1 param1, Type2 param2, ...){
    this.attr1 = param1;
    this.attr2 = param2;
    ...
  }

  // Idem, syntaxe à privilégier
  ClassName(this.attr1, this.attr2, ...);

  // constructeur nommé
  ClassName.someNamedConstructor(Type1 param1, Type2 param2, ...){
    // statements
  }
}
```

Attributs et méthodes statiques

1. les attributs et méthodes statiques sont associés à la classe elle-même et non pas ses instances.

2. toutes les instances d'une classe partagent une seule copie de chaque attribut et méthode statique de la classe.

```
// ../source/dart/snippets/oop/static_attribute_method_syntax.dart
// définition d'une classe
class ClassName {
  // attributs statiques
  static Type nomAttribut;
  ...

  // méthodes statiques
  static ReturnType nomMethode(parametres_list){

  }

  ...
}

// accès à une propriété statique d'une classe
ClassName.nomAttribut; // attribut statique
ClassName.nomMethode(args_list); // méthode statique
```

Getters/setters

1. tous les attributs d'une classe possèdent une paire getter/setter implémentée par défaut.
2. on peut redéfinir explicitement le getter/setter d'un attribut.

```
// ../source/dart/snippets/oop/getters_setters_syntax.dart
class ClassName {
  Type attr1;

  // getter de l'attribut attr1
  Type get getAttr1{
    // statements
    return attr1;
  }

  // setter de l'attribut attr1
  set setAttr1(Type attr1){
    // statements
    this.attr1 = attr1;
  }
}

// création d'une instance
ClassName obj = new ClassName(arg);

// utilisation du getter
print(obj.getAttr1);
```

```
// utilisation du setter
obj.setAttr1 = <expression>;
```

```
// ../source/dart/snippets/oop/classes_snippets.dart
void main(){
  // création d'un objet Student avec le constructeur paramétré
  var student = Student("Albert", 12, "123456");

  // invocation d'une méthode d'instance
  student.printInfo();

  // accès aux attributs d'instance en lecture (getters)
  print("Nom: ${student.name}");
  print("Age: ${student.age}");
  print("Numéro de souscription: ${student.rollNumber}\n");

  // accès aux attributs d'instance en écriture (setters)
  student.name = "Steve";
  student.age = 18;
  student.rollNumber = "456789";
  student.printInfo();

  // création d'un objet Student avec un constructeur nommé
  var student2 = Student.withoutRollNumber("John", 21);
  student2.printInfo();

  // affichage du nombre d'instances Student créées
  // en invoquant une méthode statique
  Student.printInstanceCounter();
}

// définition d'une classe
class Student {
  // attributs d'instance (privés)
  String _name = ""; //
  int _age = 0;
  String _rollNumber = "N/A";

  // attributs statiques
  static int instanceCounter = 0;

  // constructeur paramétré
  Student(this._name, this._age, this._rollNumber){
    instanceCounter++;
  }

  // constructeurs nommés
  Student.withoutName(this._age, this._rollNumber): _name = "N/A" {
    instanceCounter++;
  }
  Student.withoutRollNumber(this._name, this._age): _rollNumber = "N/A" {
```

```
    instanceCounter++;
  }

  // getters & setters
  String get name{
    return _name;
  }

  set name(String name){
    _name = name;
  }

  int get age{
    return _age;
  }

  set age(int age){
    if (age > 0){
      _age = age;
    }
  }

  String get rollNumber{
    return _rollNumber;
  }

  set rollNumber(String rollNumber){
    _rollNumber = rollNumber;
  }

  // méthode d'instance
  void printInfo(){
    print("Student Info:");
    print("=============");
    print("Name: $_name");
    print("Age: $_age");
    print("Rolling Number: $_rollNumber\n");
  }

  // méthode statique
  static void printInstanceCounter(){
    print("Nombre d'instances Student créées: $instanceCounter");
  }
}
```

```
// ../source/dart/examples/oop/car_class_example.dart
void main(){
  var c = Car(modelName: "Mercedes", color: "Red",
              manufactureYear: 2001, makeDate: "2001-12-12");
  c.printInfo();
}
```

```
class Car {
  String makeDate = "N/A";
  String modelName = "N/A";
  int manufactureYear = 0;
  int carAge = 0;
  String color = "N/A";

  Car({this.makeDate = "N/A", this.modelName = "N/A",
       this.manufactureYear = 0, this.color = "N/A",});

  int get age{
    return carAge;
  }

  set age(int currentYear){
    carAge = currentYear - manufactureYear;
  }

  void printInfo(){
    print("MakeDate: $makeDate");
    print("Model Name: $modelName");
    print("Manufacture Year: $manufactureYear");
    print("Age: $carAge");
    print("Color: $color");
  }

}
```

L'opérateur de cascade d'invocations sur un objet

1. **définition** : l'opérateur permet d'avoir une séquence d'accès aux attributs/méthodes d'un objet.
2. **syntaxe** : `obj..someAttr = <expression>..someMethod(args)..someOtherMethod()..someOtherAttr = <expression>;`.

```
// ../source/dart/examples/oop/car_cascade_notation_example.dart
void main(){
  var c = Car();

  c
    ..modelName = "Mercedes"
    ..modelDate = "2015-08-15"
    ..engineName = "Solid Technologies"
    ..printInfo()
    ..drive();
}
class Car {
  String modelName;
  String modelDate;
  String engineName;
```

```
  Car({this.modelName = "N/A", this.modelDate = "N/A", this.engineName =
"N/A"});

  void drive(){
    print("Driving a car");
  }

  void printInfo(){
    print("Model Name: $modelName");
    print("Model Date: $modelDate");
    print("Engine Name: $engineName");
  }
}
```

Héritage et polymorphisme

Aperçu

1. en Dart, les classes sont classifiées dans des hiérarchies construites pas héritage simple (*i.e., une classe enfant peut dériver d'une seule classe parent*).
2. une classe dérivée/enfant hérite d'une classe base/parent : `class ChildClass extends ParentClass {/* statements */}`.
3. quand une classe enfant hérite d'une classe parent, elle hérite tous ses attributs et toutes ses méthodes et introduit ses propres informations.
4. le constructeur par défaut d'une classe parent est invoqué implicitement lors de l'appel d'un constructeur d'une classe enfant.
5. on peut expliciter et paramétrer le constructeur d'une classe parent en utilisant `super` lors de l'appel d'un constructeur d'une classe enfant.
6. `super` peut être aussi utilisé pour accéder aux propriétés et méthodes héritées de la superclasse.
7. une classe peut redéfinir des méthodes héritées qui seront annotées par `@override`.

```
// ../source/dart/snippets/oop/inheritance_syntaxe.dart
class ParentClass {
  // content

  // constructeur par défaut
  ParentClass(){
    // statements
  }

  // constructeur paramétré
  ParentClass(params){
    // statements
  }
}

// une classe enfant de ParentClass
class ChildClass extends ParentClass {
```

```
  // (1) invocation implicite du constructeur par défaut
  // de la classe ParentClass
  ChildClass(){
    // (2) statements
  }

  // (1) invocation explicite du constructeur par défaut
  // de la classe ParentClass
  ChildClass(): super(){
    // (2) statements
  }

  // (1) invocation explicite d'un constructeur paramétré
  // de la classe ParentClass
  ChildClass(): super(args){
    // (2) statements
  }
}
```

Classes et méthodes abstraites

Aperçu

1. pour une classe abstraite :
    - on l'utilise pour regrouper des propriétés communes extensibles à un ensemble d'objets.
    - on ne peut pas l'instancier.
    - elle contient soit des méthodes abstraites, soit des méthodes concrètes, soit les deux.
    - sa déclaration est précédée par `abstract`.
2. une méthode abstraite :
    - doit être déclarée au sein d'une classe abstraite `AC`.
    - doit être implémentée par toute classe concrète dérivant directement ou indirectement de `AC`.

```
// ../source/dart/snippets/oop/abstract_classes_syntax.dart
// définition d'une classe abstraite
abstract class AbstractClassName {
  // méthodes concrètes
  ReturnType nomMethode(params){
    // statements
  }
  ...

  // méthodes abstraites
  AnotherReturnType nomMethodeAbstraite([params]);
}

class ConcreteClass extends AbstractClassName {
  // implementation des méthodes concrètes
  @override
  AnotherReturnType nomMethodeAbstraite([params]){
```

```
    // statements
  }

  ...
}
```

```
// ../source/dart/examples/oop/vehicles_example.dart
void main(){
  var car = Car(modelName: "BMW",
                modelDate: "2001-12-12",
                engineName: "Strong engine");
  car.printInfo();
  car.drive();

  var bike = Bike(modelName: "Harley-Davidson",
                  modelDate: "2012-03-18",
                 withHat: true);
  bike.printInfo();
  bike.drive();
}

// abstract superclass
abstract class Vehicle {
  String modelName;
  String modelDate;

  Vehicle({this.modelName = "N/A", this.modelDate = "N/A"});

  // abstract method declaration
  void drive();

  // concrete method
  void printInfo(){
    print("Model Name: $modelName");
    print("Model Date: $modelDate");
  }
}

// concrete subclass
class Car extends Vehicle {
  String engineName;

  Car({String modelName = "N/A", String modelDate = "N/A", this.engineName
= "N/A"}):
    super(modelName: modelName, modelDate: modelDate);

  // abstract method implementation
  @override
  void drive(){
    print("Driving a car");
  }
```

```
  // concrete method redefinition
  @override
  void printInfo(){
    super.printInfo(); // parent class implementation
    print("Engine Name: $engineName");
  }
}

class Bike extends Vehicle {
  bool withHat;

  Bike({String modelName = "N/A", String modelDate = "N/A", this.withHat =
false}):
    super(modelName: modelName, modelDate: modelDate);

  // abstract method implementation
  @override
  void drive(){
    print("Driving a bike");
  }

  // concrete method redefinition
  @override
  void printInfo(){
    super.printInfo(); // parent class implementation
    print("Comes with a hat: $withHat");
  }
}
```

Interfaces

1. une interface est une classe fournissant des signatures de services (méthodes) à implémenter par d'autres classes.
2. toute classe implémentant une interface doit (re-)définir toutes ses méthodes fournies.
3. une classe peut implémenter plusieurs interfaces : `class A implements IA, IB, IC` où `IA`, `IB`, `IC` sont des classes jouant le rôle d'interfaces à A.

```
// ../source/dart/examples/oop/arithmetic_service_interface_example.dart
void main(){
  var service = ArithmeticOperationsService();
  double a = 10.0, b = 20.0;

  print("$a + $b = ${service.add(a, b)}");
  print("$a - $b = ${service.sub(a, b)}");
  print("$a * $b = ${service.mul(a, b)}");
  print("$a / $b = ${service.div(a, b)}");
  print("inverse($a) = ${service.inv(a)}");

}
```

```dart
abstract class IArithmeticBinaryOperationsService {
  double add(double a, double b);
  double sub(double a, double b);
  double mul(double a, double b);
  double div(double a, double b);
}

abstract class IArithmeticUnaryOperationsService {
  double inv(double a);
}

class ArithmeticOperationsService implements
IArithmeticUnaryOperationsService, IArithmeticBinaryOperationsService {
  @override
  double add(double a, double b){
    return a + b;
  }

  @override
  double sub(double a, double b){
    return a - b;
  }

  @override
  double mul(double a, double b){
    return a * b;
  }

  @override
  double div(double a, double b){
    if (b != 0) {
      return a / b;
    }

    return 1;
  }

  @override
  double inv(double a){
    return -a;
  }
}
```

Enums

Aperçu

1. **définition** : un type de données dont on peut énumérer toutes les valeurs uniques possibles, indexées à partir de 0.
2. **exemple** : `Civilité = {Mr, Mme, Mlle}` est une énumération.
3. **utilité** :

- **réduire les erreurs de type** : une méthode qui attend un type d'énumération ne peut prendre aucune autre valeur que celles définies par l'énumération.
- **meilleure organisation du code**.

```dart
// ../source/dart/snippets/enums/enums_syntax.dart
enum NameEnum {
  nameConst1, // index 0
  nameConst2, // index 1
  ...
  nameConstN, // index N - 1
}

/**
 * retourne la liste des valeurs énumérées de l'énumération NameEnum
 *
 * Remarque : propriété de classe
 */
List<NameEnum> values;

/**
 * getter du nom de la valeur énumérée courante
 */
String get name;

/**
 * getter du index de la valeur énumérée courante
 */
int get index;
```

```dart
// ../source/dart/snippets/enums/enums_snippets.dart
void main(){
  for (var month in MonthOfYear.values){
    print("month: $month");
    print("name: ${month.name}, index: ${month.index}\n");
  }
}

enum MonthOfYear {
  jan,
  feb,
  mar,
  apr,
  may,
  jun,
  jul,
  aug,
  sep,
  oct,
  nov,
```

```
  dec
}
```

Futures

1. Les "Futures" en Dart représentent une valeur potentielle ou une erreur qui sera disponible dans le futur.
2. Ils sont utilisés pour des calculs qui pourraient prendre du temps, comme le travail lié aux entrées/sorties, des calculs coûteux et des requêtes réseau.
3. Ils sont fournies par la classe `Future` faisant partie du package `dart:async`.

Aperçu

1. **Valeur unique** : Contrairement aux *Streams*, chaque **Future** est un événement unique qui se termine avec une valeur ou une erreur.
2. **Chaînable** : Les Futures peuvent être chaînés en utilisant la méthode `then`, qui enregistre des callbacks à appeler lorsque le Future se termine.
3. **Gestion des erreurs** : Les Futures ont une méthode `catchError` pour une gestion centralisée des erreurs.
4. **Synchronisation** : Les mots-clés `async` et `await` sont utilisés avec les Futures pour écrire du code asynchrone de manière synchrone. `await` attend que le Future soit terminé avant de continuer l'exécution du code, alors que `async` indique qu'une fonction/méthode s'exécute d'une façon asynchrone (i.e., utilise `await` dans son implémentation).

Quelques méthodes

1. `then()`: Enregistre des callbacks à appeler lorsque le Future se termine avec une valeur.
2. `catchError()`: Gère les erreurs qui pourraient survenir dans le Future.
3. `whenComplete()`: Enregistre une fonction à appeler lorsque le Future se termine, qu'il se termine avec une valeur ou une erreur.
4. `Future.value()`: Crée un Future qui se termine immédiatement avec une valeur fournie.
5. `Future.error()`: Crée un Future qui se termine immédiatement avec une erreur.
6. `Future.delayed()`: Crée un Future qui se termine après un délai.

Exemple

Dans cet exemple nous définissions une méthode pour simuler la récupération des données d'un utilisateur d'une façon asynchrone en utilisant des Futures. Si la récupération des données réussit, on les affichera sur la console, sinon on traitera leur cas d'erreur. Enfin, quel que soit le résultat de la récupération des données de l'utilisateur on affiche la fin de l'opération à la fin.

```dart
// ../source/dart/examples/async/futures.dart
import 'dart:async';

Future<String> recupererDonneesUtilisateur() {
  // Simule une requête réseau pour récupérer les données de l'utilisateur
  return Future.delayed(Duration(seconds: 2), () {
    return 'Données Utilisateur Récupérées';
```

```
    // Décommentez la ligne suivante pour simuler une erreur
    // throw Exception('Erreur lors de la récupération des données
utilisateur');
  });
}

void main() async {
  print('Récupération des données utilisateur...');
  try {
    // Attendre le Future et imprimer le résultat
    String donneesUtilisateur = await recupererDonneesUtilisateur();
    print(donneesUtilisateur);
  } catch (e) {
    // Gérer les erreurs survenues pendant l'exécution du Future
    print(e);
  } finally {
    // Ce bloc sera exécuté après que le Future soit complété ou si une
erreur survient
    print('Terminé de récupérer les données utilisateur.');
  }
}
```

Streams

Aperçu

1. Les "Streams" en Dart fournissent un moyen de recevoir une séquence d'événements asynchrones.
2. Ils sont particulièrement utiles pour lire des données continues et potentiellement infinies, comme les entrées utilisateur, les E/S de fichiers ou les connexions réseau.
3. Ils sont similaires aux *Futures* mais pour plusieurs valeurs.
4. Ils sont fournies par la classe `Stream` faisant partie du package `dart:async`.
5. types:

- **Streams à souscription unique** : Ils peuvent être écoutés une seule fois. Après annulation de l'écoute, aucun événement supplémentaire ne peut être écouté.
- **Streams en diffusion** : Ils permettent un nombre quelconque d'auditeurs, et il est possible d'écouter le stream plusieurs fois.

Quelques méthodes

- `listen()`: Commence à écouter le stream. C'est là que vous gérez les données et les erreurs envoyées par le stream.
- `StreamController`: Utilisé pour créer un stream et pouvoir ajouter des événements de manière programmatique.
- `stream`: La propriété stream sur un StreamController donne accès au stream qu'il contrôle.
- `add()`: Ajoute des données au stream. Cette méthode est appelée sur le StreamController.
- `addError()`: Ajoute un événement d'erreur au stream.
- `close()`: Ferme le stream. Aucun événement supplémentaire ne peut être ajouté au stream une fois qu'il est fermé.

- `first`: Obtient le premier élément du stream.
- `last`: Obtient le dernier élément du stream.
- `transform()`: Transforme les événements du stream.
- `where()`: Filtre les événements du stream en fonction d'une condition.
- `map()`: Transforme chaque élément du stream.

Exemple

Dans cet exemple, nous avons créé un `StreamController` que nous utilisons pour ajouter des entiers au stream. Nous avons ensuite configuré un abonnement qui filtre les nombres pairs, les convertit en chaînes de caractères et les imprime. Les erreurs sont également gérées et imprimées, et nous imprimons un message de fin lorsque le stream est terminé. Enfin, nous ajoutons quelques nombres et une erreur au stream et fermons le stream lorsque nous avons terminé.

```dart
// ../source/dart/examples/async/streams.dart
import 'dart:async';

void main() {
  // Créer un contrôleur de stream
  final StreamController<int> controleur = StreamController<int>();

  // S'abonner au stream
  final abonnement = controleur.stream
      .where((data) => data.isEven) // Filtre pour les nombres pairs
      .map((data) => data.toString()) // Convertit l'entier en chaîne de
caractères
      .listen(
        (data) => print('Donnée: $data'), // Gère les données
        onError: (error) => print('Erreur: $error'), // Gère les erreurs
        onDone: () => print('Stream terminé'), // Gère la fin du stream
        cancelOnError: false, // Spécifie si l'abonnement doit être annulé
en cas d'erreur
      );

  // Ajouter des données au stream
  controleur.add(1); // Cela sera ignoré par le filtre where
  controleur.add(2); // Cela sera imprimé
  controleur.addError('Ceci est une erreur'); // Cela déclenchera le
callback onError
  controleur.add(3); // Cela sera également ignoré
  controleur.add(4); // Cela sera imprimé

  // Fermer le stream
  controleur.close().then((_) {
    // Cela garantit que nous imprimons 'Stream terminé' après que le
stream est réellement fermé
    abonnement.cancel();
  });
}
```