EBook Gratuito

APPRENDIMENTO

dart

Free unaffiliated eBook created from
Stack Overflow contributors.

#dart

Sommario

Di

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: dart

It is an unofficial and free dart ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official dart.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capitolo 1: Iniziare con la freccetta

Osservazioni

Dart è un linguaggio di programmazione open source, basato su classi e tipizzato opzionalmente per la creazione di applicazioni Web, sia su client sia su server, creato da Google. Gli obiettivi di Dart's design sono:

- Crea un linguaggio strutturato ma flessibile per la programmazione web.
- Fai sentire Dart familiare e naturale per i programmatori e quindi facile da imparare.
- Garantire che Dart offra prestazioni elevate su tutti i moderni browser Web e ambienti, dai piccoli dispositivi portatili all'esecuzione sul lato server.

Dart si rivolge a una vasta gamma di scenari di sviluppo, da un progetto di una sola persona senza troppa struttura a un progetto su larga scala che ha bisogno di tipi formali nel codice per dichiarare l'intenzione del programmatore.

Per supportare questa vasta gamma di progetti, Dart offre le seguenti funzionalità e strumenti:

- **Tipi opzionali:** questo significa che è possibile iniziare a programmare senza tipi e aggiungerli in seguito, se necessario.
- **Isola:** programmazione simultanea su server e client
- **Facile accesso al DOM:** usando i selettori CSS (nello stesso modo in cui lo fa jQuery)
- **Strumenti Dart IDE: i** plug-in Dart esistono per molti IDE di uso comune, ad esempio: WebStorm .
- **Dartium:** una build del browser Web Chromium con una Dart Virtual Machine integrata

link

- La homepage di Dart
- Notizie e aggiornamenti ufficiali delle freccette
- The Dartosphere - Una raccolta di post sul blog Dart recenti
- Dartisans Dartisans community su Google+
- Dart Web Development - Pagina di Google Gruppi
- Dart Language Misc - Pagina Gruppi Google
- DartLang sub-Reddit

Documentazione

- Tour of the Dart Language
- Tour delle librerie Dart

- Campioni di codice dardo
- Riferimento API Dart

FAQ

- Domande frequenti

Versioni

Versione	Data di rilascio
1.22.1	2017/02/22
1.22.0	2017/02/14
1.21.1	2016/01/13
1.21.0	2016/12/07
1.20.1	2016/10/13
1.20.0	2016/10/11
1.19.1	2016/09/07
1.19.0	2016/08/26
1.18.1	2016/08/02
1.18.0	2016/07/27
1.17.1	2016/06/10
1.17.0	2016/06/06
1.16.1	2016/05/23
1.16.0	2016/04/26
1.15.0	2016/03/09
1.14.2	2016/02/09
1.14.1	2016/02/03
1.14.0	2016/01/28
1.13.2	2016/01/05

Versione	Data di rilascio
1.13.1	2015/12/17
1.13.0	2015/11/18
1.12.2	2015/10/21
1.12.1	2015/09/08
1.12.0	2015/08/31
1.11.3	2015/08/03
1.11.1	2015/07/02
1.11.0	2015/06/24
1.10.1	2015/05/11
1.10.0	2015/04/24
1.9.3	2015/04/13
1.9.1	2015/03/25
1.8.5	2015/01/13
1.8.3	2014/12/01
1.8.0	2014/11/27
1.7.2	2014/10/14
1.6.0	2014/08/27
1.5.8	2014/07/29
1.5.3	2014/07/03
1.5.2	2014/07/02
1.5.1	2014/06/24
1.4.3	2014/06/16
1.4.2	2014/05/27
1.4.0	2014/05/20
1.3.6	2014/04/30

Versione	Data di rilascio
1.3.3	2014/04/16
1.3.0	2014/04/08
1.2.0	2014/02/25
1.1.3	2014/02/06
1.1.1	2014/01/15
1.0.0.10_r30798	2013/12/02
1.0.0.3_r30188	2013/11/12
0.8.10.10_r30107	2013/11/08
0.8.10.6_r30036	2013/11/07
0.8.10.3_r29803	2013/11/04

Examples

Installazione o configurazione

Dart SDK include tutto il necessario per scrivere ed eseguire il codice Dart: VM, librerie, analizzatore, gestore pacchetti, generatore di documenti, formattatore, debugger e altro. Se stai facendo sviluppo web, avrai anche bisogno di Dartium.

Installazione e aggiornamenti automatici

- Installazione di Dart su Windows
- Installazione di Dart su Mac
- Installazione di Dart su Linux

Installazione manuale

Puoi anche installare manualmente qualsiasi versione dell'SDK .

Ciao mondo!

Crea un nuovo file denominato `hello_world.dart` con il seguente contenuto:

```
void main() {
  print('Hello, World!');
}
```

Nel terminale, accedere alla directory contenente il file `hello_world.dart` e digitare quanto segue:

```
dart hello_world.dart
```

Premi `Invio` per visualizzare `Hello, World!` nella finestra del terminale.

Richiesta Http

html

```
<img id="cats"></img>
```

Dardo

```
import 'dart:html';

/// Stores the image in [blob] in the [ImageElement] of the given [selector].
void setImage(selector, blob) {
  FileReader reader = new FileReader();
  reader.onLoad.listen((fe) {
    ImageElement image = document.querySelector(selector);
    image.src = reader.result;
  });
  reader.readAsDataUrl(blob);
}

main() async {
  var url = "https://upload.wikimedia.org/wikipedia/commons/2/28/Tortoiseshell_she-cat.JPG";

  // Initiates a request and asynchronously waits for the result.
  var request = await HttpRequest.request(url, responseType: 'blob');
  var blob = request.response;
  setImage("#cats", blob);
}
```

Esempio

vedi Esempio su https://dartpad.dartlang.org/a0e092983f63a40b0b716989cac6969a

Getter e setter

```
void main() {
  var cat = new Cat();

  print("Is cat hungry? ${cat.isHungry}");  // Is cat hungry? true
  print("Is cat cuddly? ${cat.isCuddly}");  // Is cat cuddly? false
  print("Feed cat.");
  cat.isHungry = false;
  print("Is cat hungry? ${cat.isHungry}");  // Is cat hungry? false
  print("Is cat cuddly? ${cat.isCuddly}");  // Is cat cuddly? true
}
```

```
class Cat {
  bool _isHungry = true;

  bool get isCuddly => !_isHungry;

  bool get isHungry => _isHungry;
  bool set isHungry(bool hungry) => this._isHungry = hungry;
}
```

I getter e i setter di classe Dart consentono alle API di incapsulare le modifiche dello stato dell'oggetto.

Vedi esempio di dartpad qui: https://dartpad.dartlang.org/c25af60ca18a192b84af6990f3313233

Leggi Iniziare con la freccetta online: https://riptutorial.com/it/dart/topic/843/iniziare-con-la-freccetta

Capitolo 2: biblioteche

Osservazioni

Le direttive di `import` e `library` possono aiutarti a creare una base di codice modulare e condivisibile. Ogni app Dart è una `library` , anche se non usa una direttiva di libreria. Le librerie possono essere distribuite usando i pacchetti. Vedi Pacchetto pub e Gestione risorse per informazioni su pub, un gestore di pacchetti incluso nell'SDK.

Examples

Utilizzando le librerie

Usa `import` per specificare come viene utilizzato uno spazio dei nomi da una libreria nell'ambito di un'altra libreria.

```
import 'dart:html';
```

L'unico argomento obbligatorio da `import` è un URI che specifica la libreria. Per le librerie integrate, l'URI ha lo speciale `dart:` schema. Per altre librerie, è possibile utilizzare un percorso del file system o il `package:` scheme. Il `package:` schema specifica le librerie fornite da un gestore di pacchetti come lo strumento pub. Per esempio:

```
import 'dart:io';
import 'package:mylib/mylib.dart';
import 'package:utils/utils.dart';
```

Librerie e visibilità

A differenza di Java, Dart non ha le parole chiave `public` , `protected` e private. Se un identificatore inizia con un carattere di sottolineatura _ , è privato della sua libreria.

Se per esempio hai classe A in un file di libreria separato (ad esempio, `other.dart`), ad esempio:

```
library other;

class A {
  int _private = 0;

  testA() {
    print('int value: $_private');  // 0
    _private = 5;
    print('int value: $_private'); // 5
  }
}
```

e quindi importalo nell'app principale, ad esempio:

```
import 'other.dart';

void main() {
  var b = new B();
  b.testB();
}

class B extends A {
  String _private;

  testB() {
    _private = 'Hello';
    print('String value: $_private'); // Hello
    testA();
    print('String value: $_private'); // Hello
  }
}
```

Ottieni l'output atteso:

```
String value: Hello
int value: 0
int value: 5
String value: Hello
```

Specifica di un prefisso di libreria

Se importi due librerie con identificatori in conflitto, puoi specificare un prefisso per una o entrambe le librerie. Ad esempio, se library1 e library2 hanno entrambi una classe Element, allora potresti avere un codice come questo:

```
import 'package:lib1/lib1.dart';
import 'package:lib2/lib2.dart' as lib2;
// ...
var element1 = new Element(); // Uses Element from lib1.
var element2 =
    new lib2.Element();       // Uses Element from lib2.
```

Importare solo parte di una libreria

Se si desidera utilizzare solo parte di una libreria, è possibile importare selettivamente la libreria. Per esempio:

```
// Import only foo and bar.
import 'package:lib1/lib1.dart' show foo, bar;

// Import all names EXCEPT foo.
import 'package:lib2/lib2.dart' hide foo;
```

Caricamento lento di una libreria

Il caricamento differito (chiamato anche caricamento lazy) consente a un'applicazione di caricare una libreria su richiesta, se e quando è necessaria. Per caricare pigramente una libreria, devi

prima importarla usando differita come.

```
import 'package:deferred/hello.dart' deferred as hello;
```

Quando hai bisogno della libreria, richiama loadLibrary () usando l'identificatore della libreria.

```
greet() async {
  await hello.loadLibrary();
  hello.printGreeting();
}
```

Nel codice precedente, la parola chiave `await` sospende l'esecuzione fino a quando la libreria non viene caricata. Per ulteriori informazioni su `async` e `await` , vedere altri esempi qui supporto asincronia o visitare la parte di supporto asincronia del tour linguistico.

Leggi biblioteche online: https://riptutorial.com/it/dart/topic/3332/biblioteche

Capitolo 3: Classi

Examples

Creare una classe

Le classi possono essere create come segue:

```
class InputField {
  int maxLength;
  String name;
}
```

La classe può essere istanziata usando la `new` parola chiave dopo la quale i valori del campo saranno nulli di default.

```
var field = new InputField();
```

È possibile accedere ai valori del campo:

```
// this will trigger the setter
field.name = "fieldname";

// this will trigger the getter
print(field.name);
```

Utenti

Una classe può avere membri.

Le variabili di istanza possono essere dichiarate con / senza annotazioni di tipo e facoltativamente inizializzate. I membri non inizializzati hanno il valore `null` , a meno che non siano impostati su un altro valore dal costruttore.

```
class Foo {
  var member1;
  int member2;
  String member3 = "Hello world!";
}
```

Le variabili di classe sono dichiarate usando la parola chiave `static` .

```
class Bar {
  static var member4;
  static String member5;
  static int member6 = 42;
}
```

Se un metodo non accetta argomenti, è veloce, restituisce un valore e non ha effetti collaterali visibili, quindi è possibile utilizzare un metodo getter:

```
class Foo {
  String get bar {
    var result;
    // ...
    return result;
  }
}
```

I getter non accettano mai argomenti, quindi le parentesi per l'elenco dei parametri (vuoto) vengono omesse sia per dichiarare i getter, come sopra, sia per chiamarli, in questo modo:

```
main() {
  var foo = new Foo();
  print(foo.bar); // prints "bar"
}
```

Ci sono anche metodi setter, che devono prendere esattamente un argomento:

```
class Foo {
  String _bar;

  String get bar => _bar;

  void set bar(String value) {
    _bar = value;
  }
}
```

La sintassi per chiamare un setter è la stessa dell'assegnazione variabile:

```
main() {
  var foo = new Foo();
  foo.bar = "this is calling a setter method";
}
```

Costruttori

Un costruttore di classi deve avere lo stesso nome della sua classe.

Creiamo un costruttore per una classe Persona:

```
class Person {
  String name;
  String gender;
  int age;

  Person(this.name, this.gender, this.age);
}
```

L'esempio sopra è un modo più semplice e migliore per definire il costruttore rispetto al modo

seguente, che è anche possibile:

```
class Person {
  String name;
  String gender;
  int age;

  Person(String name, String gender, int age) {
    this.name = name;
    this.gender = gender;
    this.age = age;
  }
}
```

Ora puoi creare un'istanza di Person in questo modo:

```
var alice = new Person('Alice', 'female', 21);
```

Leggi Classi online: https://riptutorial.com/it/dart/topic/1511/classi

Capitolo 4: collezioni

Examples

Creare una nuova lista

Gli elenchi possono essere creati in più modi.

Il modo consigliato è usare un letterale di `List` :

```
var vegetables = ['broccoli', 'cabbage'];
```

Il costruttore `List` può essere utilizzato anche:

```
var fruits = new List();
```

Se preferisci una digitazione più forte, puoi anche fornire un parametro di tipo in uno dei seguenti modi:

```
var fruits = <String>['apples', 'oranges'];
var fruits = new List<String>();
```

Per la creazione di un piccolo elenco crescente, vuoto o contenente alcuni valori iniziali noti, viene preferita la forma letterale. Esistono costruttori specializzati per altri tipi di elenchi:

```
var fixedLengthList1 = new List(8);
var fixedLengthList2 = new List.filled(8, "initial text");
var computedValues = new List.generate(8, (n) => "x" * n);
var fromIterable = new List<String>.from(computedValues.getRange(2, 5));
```

Vedi anche la guida allo stile di Dart efficace sulle collezioni .

Creare un nuovo set

I set possono essere creati tramite il costruttore:

```
var ingredients = new Set();
ingredients.addAll(['gold', 'titanium', 'xenon']);
```

Creare una nuova mappa

Le mappe possono essere create in più modi.

Usando il costruttore, puoi creare una nuova mappa come segue:

```
var searchTerms = new Map();
```

I tipi per la chiave e il valore possono anche essere definiti utilizzando i generici:

```
var nobleGases = new Map<int, String>();
var nobleGases = <int, String>{};
```

Le mappe possono altrimenti essere create usando la mappa letterale:

```
var map = {
    "key1": "value1",
    "key2": "value2"
};
```

Mappare ogni elemento nella collezione.

Tutti gli oggetti di raccolta contengono un metodo `map` che accetta una `Function` come argomento, che deve assumere un singolo argomento. Ciò restituisce un `Iterable` supportato dalla raccolta. Quando `Iterable` viene iterato, ogni passaggio chiama la funzione con un nuovo elemento della raccolta e il risultato della chiamata diventa l'elemento successivo dell'iterazione.

Puoi trasformare un `Iterable` in una raccolta usando i `Iterable.toSet()` o `Iterable.toList()` , o usando un costruttore di collezioni che prende un iterabile come `Queue.from` o `List.from` .

Esempio:

```
main() {
  var cats = [
    'Abyssinian',
    'Scottish Fold',
    'Domestic Shorthair'
  ];

  print(cats); // [Abyssinian, Scottish Fold, Domestic Shorthair]

  var catsInReverse =
  cats.map((String cat) {
    return new String.fromCharCodes(cat.codeUnits.reversed);
  })
  .toList(); // [nainissybA, dloF hsittocS, riahtrohS citsemoD]

  print(catsInReverse);
}
```

Vedi esempio di dartpad qui: https://dartpad.dartlang.org/a18367ff767f172b34ff03c7008a6fa1

Filtra una lista

Dart permette di filtrare facilmente un elenco usando `where` .

```
var fruits = ['apples', 'oranges', 'bananas'];
fruits.where((f) => f.startsWith('a')).toList(); //apples
```

Ovviamente è possibile utilizzare alcuni operatori AND o OR nella clausola where.

Leggi collezioni online: https://riptutorial.com/it/dart/topic/859/collezioni

Capitolo 5: Commenti

Sintassi

- // commento a riga singola
- / * Commento su più righe / in linea * /
- /// commento Dartdoc

Osservazioni

È buona norma aggiungere commenti al codice per spiegare perché viene fatto qualcosa o per spiegare cosa fa qualcosa. Ciò aiuta i futuri lettori del tuo codice a comprendere più facilmente il tuo codice.

Argomenti correlati non su StackOverflow:

- Dardo efficace: documentazione

Examples

Commento di fine riga

Tutto a destra di `//` nella stessa riga viene commentato.

```
int i = 0; // Commented out text
```

Commento a più righe

Tutto tra `/*` e `*/` è commentato.

```
void main() {
  for (int i = 0; i < 5; i++) {
    /* This is commented, and
    will not affect code */
    print('hello ${i + 1}');
  }
}
```

Documentazione usando Dartdoc

L'uso di un commento di un doc anziché di un commento regolare consente a dartdoc di trovarlo e generare documentazione per esso.

```
/// The number of characters in this chunk when unsplit.
int get length => ...
```

Hai il permesso di utilizzare la maggior parte Markdown formattazione nei vostri commenti doc e dartdoc elaborerà di conseguenza utilizzando il pacchetto di mark-down .

```
/// This is a paragraph of regular text.
///
/// This sentence has *two* _emphasized_ words (i.e. italics) and **two**
/// __strong__ ones (bold).
///
/// A blank line creates another separate paragraph. It has some `inline code`
/// delimited using backticks.
///
/// * Unordered lists.
/// * Look like ASCII bullet lists.
/// * You can also use `-` or `+`.
///
/// Links can be:
///
/// * http://www.just-a-bare-url.com
/// * [with the URL inline](http://google.com)
/// * [or separated out][ref link]
///
/// [ref link]: http://google.com
///
/// # A Header
///
/// ## A subheader
```

Leggi Commenti online: https://riptutorial.com/it/dart/topic/2436/commenti

Capitolo 6: Conversione di dati

Examples

JSON

```
import 'dart:convert';

void main() {
    var jsonString = """
      {
        "cats": {
                "abysinnian": {
                    "origin": "Burma",
                    "behavior": "playful"
                }
            }
    }
  """;

  var obj = JSON.decode(jsonString);

  print(obj['cats']['abysinnian']['behavior']); // playful
}
```

Vedi esempio su dartpad: https://dartpad.dartlang.org/7d5958cf10e611b36326f27b062108fe

Leggi Conversione di dati online: https://riptutorial.com/it/dart/topic/2778/conversione-di-dati

Capitolo 7: Data e ora

Examples

Utilizzo di base di DateTime

```
DateTime now = new DateTime.now();
DateTime berlinWallFell = new DateTime(1989, 11, 9);
DateTime moonLanding = DateTime.parse("1969-07-20 20:18:00");  // 8:18pm
```

Puoi trovare maggiori informazioni approfondite qui .

Leggi Data e ora online: https://riptutorial.com/it/dart/topic/3322/data-e-ora

Capitolo 8: eccezioni

Osservazioni

Il codice Dart può lanciare e catturare eccezioni. Le eccezioni sono errori che indicano che è successo qualcosa di inaspettato. Se l'eccezione non viene rilevata, l'isolato che ha generato l'eccezione viene sospeso e in genere l'isolante e il relativo programma vengono terminati.

A differenza di Java, tutte le eccezioni di Dart sono eccezioni non controllate. I metodi non dichiarano quali eccezioni potrebbero lanciare e non è necessario rilevare alcuna eccezione.

Dart fornisce tipi di eccezioni ed errori , oltre a numerosi sottotipi predefiniti. Puoi, ovviamente, definire le tue eccezioni. Tuttavia, i programmi Dart possono lanciare qualsiasi oggetto non nullo, non solo oggetti Exception ed Error, come eccezione.

Examples

Eccezione personalizzata

```
class CustomException implements Exception {
  String cause;
  CustomException(this.cause);
}

void main() {
  try {
    throwException();
  } on CustomException {
    print("custom exception is been obtained");
  }
}

throwException() {
  throw new CustomException('This is my first custom exception');
}
```

Leggi eccezioni online: https://riptutorial.com/it/dart/topic/3334/eccezioni

Capitolo 9: Elenco dei filtri

introduzione

I filtri Dart vengono elencati tramite i metodi `List.where` e `List.retainWhere` . La funzione `where` accetta un argomento: una funzione booleana applicata a ciascun elemento dell'elenco. Se la funzione restituisce `true` l'elemento list viene mantenuto; se la funzione restituisce `false` , l'elemento viene rimosso.

Chiamare `theList.retainWhere(foo)` è praticamente equivalente all'impostazione di `theList = theList.where(foo)` .

Examples

Filtro di un elenco di numeri interi

```
[-1, 0, 2, 4, 7, 9].where((x) => x > 2) --> [4, 7, 9]
```

Leggi Elenco dei filtri online: https://riptutorial.com/it/dart/topic/10948/elenco-dei-filtri

Capitolo 10: Enums

Examples

Utilizzo di base

```
enum Fruit {
  apple, banana
}

main() {
  var a = Fruit.apple;
  switch (a) {
    case Fruit.apple:
      print('it is an apple');
      break;
  }

  // get all the values of the enums
  for (List<Fruit> value in Fruit.values) {
    print(value);
  }

  // get the second value
  print(Fruit.values[1]);
}
```

Leggi Enums online: https://riptutorial.com/it/dart/topic/5107/enums

Capitolo 11: Espressioni regolari

Sintassi

- var regExp = RegExp (r '^ (. *) $', multiLine: true, caseSensitive: false);

Parametri

Parametro	Dettagli
`String source`	L'espressione regolare come una `String`
`{bool multiline}`	Se si tratta di un'espressione regolare multilinea. (corrisponde `^` e `$` all'inizio e alla fine di ogni riga singolarmente non dell'intera stringa)
`{bool caseSensitive}`	Se l'espressione è sensibile al maiuscolo e minuscolo

Osservazioni

Le espressioni regolari di Dart hanno la stessa sintassi e semantica delle espressioni regolari JavaScript. Vedi http://ecma-international.org/ecma-262/5.1/#sec-15.10 per la specifica delle espressioni regolari JavaScript.

Ciò significa che qualsiasi risorsa JavaScript che trovi su Regular Expressions online si applica a dart.

Examples

Crea e usa un'espressione regolare

```
var regExp = new RegExp(r"(\w+)");
var str = "Parse my string";
Iterable<Match> matches = regExp.allMatches(str);
```

È consigliabile utilizzare "stringhe non elaborate" (prefisso con `r`) durante la scrittura di espressioni regolari in modo da poter utilizzare barre rovesciate senza escape nell'espressione.

Leggi Espressioni regolari online: https://riptutorial.com/it/dart/topic/3624/espressioni-regolari

Capitolo 12: Flusso di controllo

Examples

Se altro

Dart ha If Else:

```
if (year >= 2001) {
  print('21st century');
} else if (year >= 1901) {
  print('20th century');
} else {
  print('We Must Go Back!');
}
```

Dart ha anche un operatore ternario `if` :

```
var foo = true;
print(foo ? 'Foo' : 'Bar'); // Displays "Foo".
```

Mentre Loop

Mentre i loop e i loop while sono consentiti in Dart:

```
while(peopleAreClapping()) {
  playSongs();
}
```

e:

```
do {
  processRequest();
} while(stillRunning());
```

I loop possono essere interrotti utilizzando una pausa:

```
while (true) {
  if (shutDownRequested()) break;
  processIncomingRequests();
}
```

Puoi saltare le iterazioni in un ciclo usando continua:

```
for (var i = 0; i < bigNumber; i++) {
  if (i.isEven){
    continue;
  }
  doSomething();
```

```
}
```

Per Loop

Sono ammessi due tipi di loop for:

```
for (int month = 1; month <= 12; month++) {
  print(month);
}
```

e:

```
for (var object in flybyObjects) {
  print(object);
}
```

Il ciclo `for-in` è comodo quando si `Iterable` semplicemente su una collezione `Iterable` . Esiste anche un metodo `forEach` che puoi chiamare su oggetti `Iterable` che si comportano come `for-in` :

```
flybyObjects.forEach((object) => print(object));
```

o, più concisamente:

```
flybyObjects.forEach(print);
```

Scatola dell'interruttore

Dart ha un interruttore che può essere usato al posto di lunghe istruzioni if-else:

```
var command = 'OPEN';

switch (command) {
  case 'CLOSED':
    executeClosed();
    break;
  case 'OPEN':
    executeOpen();
    break;
  case 'APPROVED':
    executeApproved();
    break;
  case 'UNSURE':
    // missing break statement means this case will fall through
    // to the next statement, in this case the default case
  default:
    executeUnknown();
}
```

È possibile confrontare solo le costanti di intero, stringa o compilazione. Gli oggetti confrontati devono essere istanze della stessa classe (e non di nessuno dei suoi sottotipi) e la classe non deve sovrascrivere ==.

Un aspetto sorprendente dello switch in Dart è che le clausole case non vuote devono terminare con break, o meno comunemente, continuare, lanciare o restituire. Cioè, le clausole non vuote non possono fallire. È necessario terminare esplicitamente una clausola non vuota, solitamente con una pausa. Si otterrà un avviso statico se si omette l'interruzione, la continuazione, il lancio o il ritorno e il codice eseguirà un errore in quella posizione in fase di esecuzione.

```
var command = 'OPEN';
switch (command) {
  case 'OPEN':
    executeOpen();
    // ERROR: Missing break causes an exception to be thrown!!

  case 'CLOSED': // Empty case falls through
  case 'LOCKED':
    executeClosed();
    break;
}
```

Se vuoi ricorrere in un `case` non vuoto, puoi usare `continue` e un'etichetta:

```
      var command = 'OPEN';
      switch (command) {
        case 'OPEN':
          executeOpen();
          continue locked;
locked: case 'LOCKED':
          executeClosed();
          break;
      }
```

Leggi Flusso di controllo online: https://riptutorial.com/it/dart/topic/923/flusso-di-controllo

Capitolo 13: funzioni

Osservazioni

Dart è un vero linguaggio orientato agli oggetti, quindi anche le funzioni sono oggetti e hanno un tipo, Function. Ciò significa che le funzioni possono essere assegnate a variabili o passate come argomenti ad altre funzioni. Puoi anche chiamare un'istanza di una classe Dart come se fosse una funzione.

Examples

Funziona con parametri denominati

Quando si definisce una funzione, utilizzare {param1, param2, ...} per specificare i parametri denominati:

```
void enableFlags({bool bold, bool hidden}) {
  // ...
}
```

Quando si chiama una funzione, è possibile specificare i parametri denominati utilizzando paramName: valore

```
enableFlags(bold: true, hidden: false);
```

Scopo della funzione

Le funzioni Dart possono anche essere dichiarate anonime o nidificate. Ad esempio, per creare una funzione nidificata, basta aprire un nuovo blocco funzione all'interno di un blocco funzione esistente

```
void outerFunction() {

    bool innerFunction() {
        /// Does stuff
    }
}
```

La funzione `innerFunction` può ora essere utilizzata all'interno e solo all'interno di `outerFunction` . Nessun'altra altra funzione ha accesso ad essa.

Le funzioni in Freccetta possono anche essere dichiarate anonimamente, che è comunemente usato come argomento di funzione. Un esempio comune è il metodo di `sort` dell'oggetto `List` . Questo metodo accetta un argomento facoltativo con la seguente firma:

```
int compare(E a, E b)
```

La documentazione afferma che la funzione deve restituire `0` se `a` e `b` sono uguali. Restituisce `-1` se `a < b` e `1` se `a > b` .

Sapendo questo, possiamo ordinare una lista di numeri interi usando una funzione anonima.

```
List<int> numbers = [4,1,3,5,7];

numbers.sort((int a, int b) {
   if(a == b) {
      return 0;
   } else if (a < b) {
      return -1;
   } else {
      return 1;
   }
});
```

La funzione anonima può anche essere associata a identificatori in questo modo:

```
Function intSorter = (int a, int b) {
   if(a == b) {
      return 0;
   } else if (a < b) {
      return -1;
   } else {
      return 1;
   }
}
```

e usato come una variabile ordinaria.

```
numbers.sort(intSorter);
```

Leggi funzioni online: https://riptutorial.com/it/dart/topic/2965/funzioni

Capitolo 14: Interoperabilità Dart-JavaScript

introduzione

L'interoperabilità Dart-JavaScript ci consente di eseguire codice JavaScript dai nostri programmi Dart.

L'interoperabilità si ottiene utilizzando la libreria `js` per creare stub Dart. Questi stub descrivono l'interfaccia che vorremmo avere con il codice JavaScript sottostante. In fase di runtime, la chiamata allo stub Dart invocherà il codice JavaScript.

Examples

Chiamare una funzione globale

Supponiamo di voler invocare la funzione JavaScript `JSON.stringify` che riceve un oggetto, lo codifica in una stringa JSON e lo restituisce.

Tutto quello che dovremmo fare è scrivere la firma della funzione, contrassegnarla come esterna e annotarla con l'annotazione `@JS` :

```
 @JS("JSON.stringify")
external String stringify(obj);
```

L'annotazione `@JS` verrà utilizzata da qui in avanti per contrassegnare le classi Dart che vorremmo utilizzare anche in JavaScript.

Disposizione di classi JavaScript / spazi dei nomi

Supponiamo di voler racchiudere l'API JavaScript di Google Maps `google.maps` :

```
@JS('google.maps')
library maps;

import "package:js/js.dart";

@JS()
class Map {
  external Map(Location location);
  external Location getLocation();
}
```

Ora abbiamo la classe Map Dart che corrisponde alla classe JavaScript `google.maps.Map` .

L'esecuzione di una `new Map(someLocation)` in Dart invocherà la `new google.maps.Map(location)` in JavaScript.

Si noti che non è necessario assegnare un nome alla classe Dart uguale alla classe JavaScript:

```
@JS("LatLng")
class Location {
  external Location(num lat, num lng);
}
```

La classe Dart `Location` corrisponde alla classe `google.maps.LatLng` .

L'uso di nomi incoerenti è scoraggiato in quanto possono creare confusione.

Passando ai letterali degli oggetti

È prassi comune in JavaScript passare i letterali degli oggetti alle funzioni:

```
// JavaScript
printOptions({responsive: true});
Unfortunately we cannot pass Dart Map objects to JavaScript in these cases.
```

Quello che dobbiamo fare è creare un oggetto Dart che rappresenti il letterale dell'oggetto e contenga tutti i suoi campi:

```
// Dart
@JS()
@anonymous
class Options {
  external bool get responsive;

  external factory Options({bool responsive});
}
```

Si noti che la classe Dart `Options` non corrisponde a nessuna classe JavaScript. Come tale, dobbiamo contrassegnarlo con l'annotazione `@anonymous` .

Ora possiamo creare uno stub per la funzione printOptions originale e chiamarlo con un nuovo oggetto Options:

```
// Dart
@JS()
external printOptions(Options options);

printOptions(new Options(responsive: true));
```

Leggi Interoperabilità Dart-JavaScript online:
https://riptutorial.com/it/dart/topic/9240/interoperabilita-dart-javascript

Capitolo 15: Programmazione asincrona

Examples

Restituire un futuro usando un Completatore

```
Future<Results> costlyQuery() {
  var completer = new Completer();

  database.query("SELECT * FROM giant_table", (results) {
    // when complete
    completer.complete(results);
  }, (error) {
    completer.completeException(error);
  });

  // this returns essentially immediately,
  // before query is finished
  return completer.future;
}
```

Async e attendere

```
import 'dart:async';

Future main() async {
  var value = await _waitForValue();
  print("Here is the value: $value");
  //since _waitForValue() returns immediately if you un it without await you won't get the
result
  var errorValue = "not finished yet";
  _waitForValue();
  print("Here is the error value: $value");// not finished yet
}

Future<int> _waitForValue() => new Future((){

  var n = 100000000;

  // Do some long process
  for (var i = 1; i <= n; i++) {
    // Print out progress:
    if ([n / 2, n / 4, n / 10, n / 20].contains(i)) {
      print("Not done yet...");
    }

    // Return value when done.
    if (i == n) {
      print("Done.");
      return i;
    }
  }
});
```

Vedi esempio su Dartpad: https://dartpad.dartlang.org/11d189b51e0f2680793ab3e16e53613c

Conversione di callback in Futures

Dart ha una robusta libreria asincrona, con Future , Stream e altro. Tuttavia, a volte potresti imbatterti in un'API asincrona che utilizza *callback* anziché *Futures* . Per colmare il divario tra callback e Futures, Dart offre la classe *Completer* . È possibile utilizzare un Completamento per convertire una richiamata in un futuro.

I completatori sono ottimi per il collegamento di un'API basata sul callback con un'API basata sul futuro. Ad esempio, supponiamo che il tuo driver di database non usi Futures, ma devi restituire un futuro. Prova questo codice:

```
// A good use of a Completer.

Future doStuff() {
  Completer completer = new Completer();
  runDatabaseQuery(sql, (results) {
    completer.complete(results);
  });
  return completer.future;
}
```

Se si utilizza un'API che restituisce già un Futuro, non è necessario utilizzare un Completamento.

Leggi Programmazione asincrona online: https://riptutorial.com/it/dart/topic/2520/programmazione-asincrona

Capitolo 16: pub

Osservazioni

Quando installi Dart SDK, uno degli strumenti che ottieni è pub. Lo strumento pub fornisce comandi per una varietà di scopi. Un comando installa i pacchetti, un altro avvia un server HTTP per il test, un altro prepara la tua app per la distribuzione e un altro pubblica il tuo pacchetto su pub.dartlang.org . È possibile accedere ai comandi pub tramite un IDE, come WebStorm o sulla riga di comando.

Per una panoramica di questi comandi, vedere Comandi pub .

Examples

costruzione di pub

Utilizza la versione di un pub quando sei pronto per distribuire la tua app Web. Quando esegui pub build, genera le risorse per il pacchetto corrente e tutte le sue dipendenze, inserendole in una nuova directory chiamata build.

Per utilizzare la versione di `pub build` , basta eseguirlo nella directory principale del pacchetto. Per esempio:

```
$ cd ~/dart/helloworld
$ pub build
Building helloworld......
Built 5 files!
```

pub servire

Questo comando avvia un server di sviluppo, o server di sviluppo, per l'app Web Dart. Il server di sviluppo è un server HTTP su localhost che serve le risorse dell'app Web.

Avviare il server di sviluppo dalla directory che contiene il file `pubspec.yaml` dell'app Web:

```
$ cd ~/dart/helloworld
$ pub serve
Serving helloworld on http://localhost:8080
```

Leggi pub online: https://riptutorial.com/it/dart/topic/3335/pub

Capitolo 17: stringhe

Examples

Concatenazione e interpolazione

È possibile utilizzare l'operatore più (`+`) per concatenare le stringhe:

```
'Dart ' + 'is ' + 'fun!'; // 'Dart is fun!'
```

È anche possibile utilizzare valori letterali stringa adiacenti per la concatenazione:

```
'Dart ' 'is ' 'fun!';    // 'Dart is fun!'
```

Puoi usare `${}` per interpolare il valore delle espressioni Dart all'interno delle stringhe. Le parentesi graffe possono essere omesse quando si valutano gli identificatori:

```
var text = 'dartlang';
'$text has ${text.length} letters'; // 'dartlang has 8 letters'
```

Stringhe valide

Una stringa può essere singola o multilinea. Le stringhe a linea singola vengono scritte utilizzando le virgolette singole o doppie corrispondenti e le stringhe multilinea vengono scritte utilizzando le virgolette triple. Le seguenti sono tutte stringhe Dart valide:

```
'Single quotes';
"Double quotes";
'Double quotes in "single" quotes';
"Single quotes in 'double' quotes";

'''A
multiline
string''';

"""
Another
multiline
string""";
```

Costruire da parti

La generazione di una stringa a livello di programmazione viene eseguita al meglio con un StringBuffer . Un StringBuffer non genera un nuovo oggetto String finché non viene chiamato `toString()` .

```
var sb = new StringBuffer();
```

```
sb.write("Use a StringBuffer");
sb.writeAll(["for ", "efficient ", "string ", "creation "]);
sb.write("if you are ")
sb.write("building lots of strings");

// or you can use method cascades:

sb
  ..write("Use a StringBuffer")
  ..writeAll(["for ", "efficient ", "string ", "creation "])
  ..write("if you are ")
  ..write("building lots of strings");

var fullString = sb.toString();

print(fullString);
// Use a StringBufferfor efficient string creation if you are building lots of strings

sb.clear(); // all gone!
```

Leggi stringhe online: https://riptutorial.com/it/dart/topic/5003/stringhe

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con la freccetta	4444, Challe, Community, Damon, Florian Loitsch, Gomiero, Kleak, losnake, martin, Raph, Timothy C. Quinn
2	biblioteche	Challe, Ganymede
3	Classi	Ganymede, Hoylen, Jan Vladimir Mostert, Raph
4	collezioni	Alexi Coard, Damon, Jan Vladimir Mostert, Kleak, lrn, Pacane, Raph
5	Commenti	Challe
6	Conversione di dati	Damon
7	Data e ora	Challe
8	eccezioni	Challe
9	Elenco dei filtri	jxmorris12
10	Enums	Challe
11	Espressioni regolari	enyo
12	Flusso di controllo	Ganymede, Jan Vladimir Mostert, Pacane, Raph
13	funzioni	Jan Vladimir Mostert, Kim Rostgaard Christensen
14	Interoperabilità Dart-JavaScript	Meshulam Silk
15	Programmazione asincrona	Challe, Damon, Ray Hulha, Zied Hamdi
16	pub	Challe
17	stringhe	Challe