

main Function

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

Variables, Data Types & Comments

```
// Use var with type inference or use type  
// directly  
var myAge = 35; // inferred int created with var  
var pi = 3.14; // inferred double created with var  
int yourAge = 27; // type name instead of var  
// dynamic can have value of any type  
dynamic numberOfKittens;  
// dynamic String  
numberOfKittens = 'There are no kittens!';  
numberOfKittens = 0; // dynamic int  
bool areThereKittens = true; // bool  
// Compile-time constants  
const speedOfLight = 299792458;  
// Immutables with final  
final planet = 'Jupiter';  
// planet = 'Mars'; // error: planet is immutable  
late double defineMeLater; // late variables  
// --- Comments ---  
// This is a comment  
print(myAge); // This is also a comment  
/*  
And so is this, spanning  
multiple lines  
*/  
/// This is a doc comment  
// --- Simple enums ---  
enum Direction {  
  north, south, east, west  
}  
final sunset = Direction.west;
```

Nullables and Non-nullables

```
// Non-nullable, can't be used until assigned  
int age;  
print(age); // error: must be assigned before use  
double? height; // null by default  
print(height); // null  
String? err;
```

```
// "if null" check  
var error = err ?? 'No error'; // No error  
// Null-check compound assignment  
err ??= error;  
// Null-aware operator(?.) for property access  
print(err?.toUpperCase());  
err!.length; // '!' casts err to non-nullable
```

Operators

```
// --- Arithmetic ---  
40 + 2; // 42  
44 - 2; // 42  
21 * 2; // 42  
84 / 2; // 42  
84.5 ~/ 2.0; // int value 42  
392 % 50; // 42  
// Types can be implicitly converted  
var answer = 84.0 / 2; // int 2 to double  
// --- Equality and Inequality ---  
42 == 43; // false  
42 != 43; // true  
// --- Increment and Decrement ---  
print(answer++); // 42, prints first  
print(--answer); // 42, decrements first  
// --- Comparison ---  
42 < 43; // true  
42 > 43; // false  
42 <= 43; // true  
42 >= 43; // false  
// --- Compound assignment ---  
answer += 1; // 43  
answer -= 1; // 42  
answer *= 2; // 84  
answer /= 2; // 42  
// --- Logical ---  
(41 < answer) && (answer < 43); // true  
(41 < answer) || (answer > 43); // true  
!(41 < answer); // false  
// Conditional/ternary operator  
var error = err == null  
  ? 'No error': err; // No error
```

Strings

```
// Can use double or single quotes  
// for String type  
var firstName = 'Albert';  
String lastName = "Einstein";  
// You can interpolate variables in  
// strings with $  
var fullName = '$firstName $lastName';  
// Strings support special characters(\n, \t, etc)  
print('Name:\t$fullName');  
// Name: Albert Einstein  
// Concatenate adjacent strings  
var quote = 'If you can\'t explain it simply\n'  
"you don't understand it well enough.";  
// Concatenate with +  
var energy = "Mass" + ' times ' + "c squared";  
// Preserving formatting with """  
var model = """I'm not creating the universe.  
I'm creating a model of the universe,  
which my or may not be true.""";  
// Raw string with r prefix  
var rawString = "I'll\nbe\nback!";  
// I'll\nbe\nback!
```

Control Flow: Conditionals

```
var animal = 'fox';  
if (animal == 'cat' || animal == 'dog') {  
  print('Animal is a house pet.');//  
} else if (animal == 'rhino') {  
  print('That\'s a big animal.');//  
} else {  
  print('Animal is NOT a house pet.');//  
}  
// switch statement  
switch (animal) {  
  case 'cat':  
  case 'dog':  
    print('Animal is a house pet.');//  
    break;  
  case 'rhino':  
    print('That\'s a big animal.');//  
    break;  
  default:  
    print('Animal is NOT a house pet.');//  
}
```

Control Flow: While Loops

```
var i = 1;
// while, print 1 to 9
while (i < 10) {
  print(i);
  i++;
}
// do while, print 1 to 9
do {
  print(i);
  ++i;
} while (i < 10);
// break at 5
do {
  print(i);
  if (i == 5) {
    break;
  }
  ++i;
} while (i < 10);
```

Control Flow: For Loops

```
var sum = 0;
// Init; condition; action for loop
for (var i = 1; i <= 10; i++) {
  sum += i;
}
// for-in loop for list
var numbers = [1, 2, 3, 4];
for (var number in numbers) {
  print(number);
}
// Skip over 3 with continue
for (var number in numbers) {
  if (number == 3) {
    continue;
  }
  print(number);
}
// forEach
numbers.forEach(print);
```

Functions

```
// Named function
bool isTeen(int age) {
  return age > 12 && age < 20;
}
const myAge = 19;
print(isTeen(myAge)); // true
// Arrow syntax for one line functions
int multiply(int a, int b) => a * b;
multiply(14, 3); // 42
// Optional positional parameters
String titledName(
  String name, [String? title]) {
  return '${title ?? ''} $name';
}
titledName('Albert Einstein');
// Albert Einstein
titledName('Albert Einstein', 'Prof');
// Prof Albert Einstein
// Named & function parameters
int applyTo(
  int Function(int) op,
  {required int number}) {
  return op(number);
}
int square(int n) => n * n;
applyTo(number: 3, square); // 9
// Optional named & default parameters
bool withinTolerance(
  int value, {int min = 0, int? max}) {
  return min <= value &&
    value <= (max ?? 10);
}
withinTolerance(5); // true
```

Anonymous Functions

```
// Assign anonymous function to a variable
var multiply = (int a, int b) => a * b;
// Call the function variable
multiply(14, 3); // 42
```

// Closures

```
Function applyMultiplier(num multiplier) {
  return (num value) => value * multiplier;
}
var triple = applyMultiplier(3);
triple(14.0); // 42.0
```

Collections: Lists

```
// Fixed-size list of 3, filled with empty
// values
var pastries = List.filled(3, '');
// List assignment using index
pastries[0] = 'cookies';
pastries[1] = 'cupcakes';
pastries[2] = 'donuts';
// Empty, growable list
List<String> desserts = [];
var desserts
  = List<String>.empty(growable: true);
desserts.add('cookies');
// Growable list literals
var desserts = ['cookies', 'cupcakes', 'pie'];
// List properties
desserts.length; // 3
desserts.first; // 'cookies'
desserts.last; // 'pie'
desserts.isEmpty; // false
desserts.isNotEmpty; // true
desserts.firstWhere((str) => str.length < 4);
// pie
// Collection if
var peanutAllergy = true;
List<String>? candy;
candy = [
  'junior mints',
  'twizzlers',
  if (!peanutAllergy) 'reeses'
];
// Collection for
var numbers = [1, 2, 3];
var doubleNumbers = [
  for (var number in numbers) 2 * number
]; // [2, 4, 6]
```

Collections: Lists Operations

```
// Spread and null-spread operators
var pastries = ['cookies', 'cupcakes'];
var desserts = ['donuts', ...pastries, ...?candy];
// Using map to transform lists
final squares = numbers.map(
  (number) => number * number)
.toList(); // [1, 4, 9]
// Filter list using where
var evens = squares.where(
  (number) => number.isEven); // (4)
// Reduce list to combined value
var amounts = [199, 299, 299, 199, 499];
var total = amounts.reduce(
  (value, element) => value + element); // 1495
```

Collections: Sets

```
// Create a set of int
var someSet = <int>[];
// Set type inference
var evenSet = {2, 4, 6, 8};
evenSet.contains(2); // true
evenSet.contains(99); // false
// Adding and removing elements
someSet.add(0);
someSet.add(2112);
someSet.remove(2112);
// Add a list to a set
someSet.addAll([1, 2, 3, 4]);
someSet.intersection(evenSet); // {2, 4}
someSet.union(evenSet); // {0, 1, 2, 3, 4, 6, 8}
```

Collections: Maps

```
// Map from String to int
final emptyMap = Map<String, int>();
```

```
// Map from String to String
final avengers = {
  'Iron Man': 'Suit',
  'Captain America': 'Shield', 'Thor': 'Hammer'
};
// Element access by key
final ironManPower = avengers['Iron Man']; // Suit
avengers.containsKey('Captain America'); // true
avengers.containsValue('Arrows'); // false
// Access all keys and values
avengers.forEach(print);
// Iron Man, Captain America, Thor
avengers.values.forEach(print);
// Suit, Shield, Hammer
// Loop over key-value pairs
avengers.forEach(
  (key, value) => print('$key -> $value'));
```

Classes and Objects

```
class Actor {
  // Properties
  String name;
  var filmography = <String>[];
  // Constructor
  Actor(this.name, this.filmography);
  // Named constructor
  Actor.rey({this.name = 'Daisy Ridley'}) {
    filmography = ['The Force Awakens',
      'Murder on the Orient Express'];
  }
  // Calling other constructors
  Actor.inTraining(String name): this(name, []);
  // Constructor with initializer list
  Actor.gameOfThrones(String name)
    : this.name = name,
      this.filmography = ['Game of Thrones'] {
        print('My name is ${this.name}');
  }
  // Getters and setters
  String get debut
    => '$name debuted in ${filmography.first}';
  set debut(String value)
    => filmography.insert(0, value);
```

```
// Methods
void signOnForSequel(String franchiseName) {
  filmography.add('Upcoming $franchiseName sequel');
}
// Override from Object
String toString()
=> '${[name, ...filmography].join('\n- ')}\n';
}
var gotgStar = Actor('Zoe Saldana', []);
gotgStar.name = 'Zoe Saldana';
gotgStar.filmography.add('Guardians of the Galaxy');
gotgStar.debut = 'Center Stage';
print(Actor.rey().debut); // The Force Awakens
var kit = Actor.gameOfThrones('Kit Harington');
var star = Actor.inTraining('Super Star');
// Cascade syntax
gotg // Get an object
..name == 'Zoe' // Set property
..signOnForSequel('Star Trek'); // Call method
```

Static Class Members & Enhanced Enums

```
enum PhysicistType {
  theoretical, experimental, both
}
class Physicist {
  String name;
  PhysicistType type;
  // Internal constructor
  Physicist._internal(this.name, this.type);
  // Static property
  static var physicistCount = 0;
  // Static method
  static Physicist newPhysicist(
    String name, PhysicistType type
  ) {
    physicistCount++;
    return Physicist._internal(name, type);
  }
}
// Calling static class members
final emmy = Physicist.newPhysicist(
  'Emmy Noether', PhysicistType.theoretical);
final lise = Physicist.newPhysicist(
  'Lise Meitner', PhysicistType.experimental);
print(Physicist.physicistCount);
```

```
// Enhanced enums with members & constructors
enum Season {
    // Call constructor when the enum's created
    winter(['December', 'January', 'February']),
    spring(['March', 'April', 'May']),
    summer(['June', 'July', 'August']),
    autumn(['September', 'October', 'November']);
    // Enum property
    final List<String> months;
    // Constructor
    const Season(this.months);
    // Methods
    bool get isCold => name == 'winter';
    // Override methods
    @override
    String toString()
    => 'Season: $name, months: ${months.join(', ')}';
}
final summer = Season.summer;
print(summer); // Prints the toString() value
print(summer.isCold); // false
```

Class Inheritance

```
// Base aka parent class
class Person {
    // Properties inherited by child
    String firstName;
    String lastName;
    // Parent class constructor
    Person(this.firstName, this.lastName);
    // Parent class method
    String get fullName => '$firstName $lastName';
    // Optional, override toString() from Object
    // All classes have Object as root class
    @override
    String toString() => fullName;
}
```

```
// Subclass aka child class
class Student extends Person {
    // Properties specific to child
    var grades = <String>[];
    // Pass initializers to parent constructor
    Student(super.firstName, super.lastName);
    // Same as:
    /**
     * Student(String firstName, String lastName)
     *   : super(firstName, lastName)
     */
    // Optional, override parent method
    @override
    String get fullName => '$lastName, $firstName';
}
final jon = Person('Jon', 'Snow');
// Calls parent constructor
final jane = Student('Jane', 'Snow');
print(jon); // Jon Snow
print(jane); // Snow, Jane
```

Abstract Classes, Interfaces & Mixins

```
enum BloodType { warm, cold }
abstract class Animal {
    abstract BloodType bloodType; //Abstract Property
    // Abstract method, not implemented
    void goSwimming();
}
mixin Milk {
    bool? hasMilk;
    bool doIHaveMilk() => hasMilk ?? false;
}
// Concrete class inheriting abstract class
class Cat extends Animal with Milk {
    // Set property
    BloodType bloodType = BloodType.warm;
    Cat() { hasMilk = true; } // Set mixin property
    // Concrete subclass must implement
    // abstract methods
    @override
    void goSwimming() => print('No thanks!');
```

```
// Concrete class that also implements
// Comparable interface
class Dolphin extends Animal implements Comparable<Dolphin> {
    // Must implement abstract property
    BloodType bloodType = BloodType.cold;
    // Class property
    double length;
    // Concrete subclass constructor
    Dolphin(this.length);
    // Must implement abstract methods
    @override
    void goSwimming() => print('Click! Click!');
    // Must also implement interface methods
    @override
    int compareTo(Dolphin other)
        => length.compareTo(other.length);
    @override
    String toString() => '$length meters';
}
class Reptile extends Animal with Milk {
    BloodType bloodType = BloodType.warm;
    Reptile() { hasMilk = true; }
    @override
    void goSwimming() => print('Sure! ');
}
// var snake = Animal();
// error: can't instantiate abstract class
// Can instantiate concrete classes
var garfield = Cat();
var flipper = Dolphin(4.0);
var snake = Reptile();
// Call concrete methods
flipper.goSwimming(); // Click! Click!
garfield.goSwimming(); // No thanks!
// Use interface implementation
var orca = Dolphin(8.0);
var alpha = Dolphin(5.0);
var dolphins = [alpha, orca, flipper];
// Uses length to sort based on compareTo
dolphins.sort();
print(dolphins);
// [4.0 meters, 5.0 meters, 8.0 meters]
print(snake.doIHaveMilk()); // false
print(garfield.doIHaveMilk()); // true
```