Complete Dart Cheat Sheet for Students & Developers

Main Function Entry

All Dart applications start execution from the `main()` function.

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

Variables

Variables are declared using `var`, `final`, or `const` with type inference or explicit typing.

```
var age = 30;
final name = 'Lily';
const PI = 3.14159;
```

Data Types

Dart has strong, static typing with support for null safety.

```
int x = 5;
String greeting = "Hi";
bool isOn = true;
List<int> numbers = [1, 2, 3];
Map<String, int> scores = {'a': 1};
```

Conditionals

Control flow using if-else and switch statements.

```
if (temp > 30) {
  print('Hot');
} else {
  print('Cool');
}

switch(day) {
  case 'Monday':
    print('Start');
    break;
}
```

Loops

Dart supports for, while, do-while, and for-in loops.

```
for (int i = 0; i < 5; i++) {
  print(i);
}

List<String> items = ['a', 'b'];
for (var item in items) {
  print(item);
}
```

Functions

Functions are reusable blocks of code with optional parameters and return types.

```
void greet(String name) {
  print('Hello, $name');
}

greet('Alice');
```

Optional & Named Parameters

Functions can have positional or named parameters with default values.

```
void sayHello(String name, [String greeting = 'Hi']) {
  print('$greeting, $name!');
}

void describe({required String name, int age = 18}) {
  print('$name is $age');
}
```

Arrow Functions

Short-hand syntax for single-expression functions.

```
int square(int x) => x * x;
```

Classes & Objects

Classes define blueprints; objects are created from them.

```
class Dog {
  String name;
  Dog(this.name);
  void bark() => print('$name barks!');
}

var d = Dog('Buddy');
d.bark();
```

Constructors

Special methods to initialize objects. Supports default, named, and factory constructors.

```
class Point {
  double x, y;
  Point(this.x, this.y);
  Point.origin() : x = 0, y = 0;
}
```

Null Safety

Dart uses sound null safety to prevent null reference errors. Use '?' for nullable and '!' to assert non-null.

```
String? name;
name = 'Alice';
print(name!.length);
```

Collections: List, Set, Map

Dart provides powerful collection types with built-in methods.

```
List<int> nums = [1, 2, 3];
nums.add(4);
Set<String> tags = {'a', 'b'};
tags.add('c');
Map<String, String> user = {'id': '123'};
```

Collection Methods

Useful methods on lists, sets, and maps.

```
nums.contains(2);
tags.remove('a');
user.keys;
user.values;
user['id'] = '456';
```

Spread & Null-aware Spread Operator

Use `...` to spread elements, `...?` for null-aware spread.

```
var list1 = [1, 2];
var list2 = [...list1, 3];
List<int>? list3;
var list4 = [...?list3, 4];
```

Cascade Notation

Allows chaining multiple operations on the same object using `..`.

```
myObj
  ..method1()
  ..property = value;
```

Async & Await

Dart supports asynchronous programming with `Future`, `async`, and `await`.

```
Future<String> getData() async {
  return 'data';
}

void main() async {
  String d = await getData();
  print(d);
}
```

Exception Handling

Use try-catch-finally to manage exceptions gracefully.

```
try {
  var x = 5 ~/ 0;
} catch (e) {
  print('Error: $e');
} finally {
  print('Done');
}
```

Complete Dart Cheat Sheet for Students & Developers

Type Conversion & Parsing

Convert between types using `toString`, `int.parse`, etc.

```
String numStr = '42';
int number = int.parse(numStr);
String back = number.toString();
```

Getters and Setters

Encapsulate fields using getter and setter methods.

```
class Circle {
  double radius;
  Circle(this.radius);
  double get area => 3.14 * radius * radius;
  set updateRadius(double r) => radius = r;
}
```

Static Members

Static members belong to the class, not instances.

```
class Utils {
  static const pi = 3.14;
  static void printInfo() => print('Utility');
}
```

Generics

Enable type-safe code using generic types.

```
List<String> names = ['A', 'B'];
Map<String, int> scores = {};
```

Anonymous Functions & Lambdas

Use unnamed functions for callbacks and short logic blocks.

```
list.forEach((item) {
  print(item);
});
```

Higher-Order Functions

Functions that take other functions as parameters.

```
void process(int x, Function fn) {
  fn(x);
}

process(5, (n) => print(n * 2));
```