222301504 Cross Platform Mobile Application Development

Basic OOPS in flutter: Function

- Functions are nothing but a piece of organized code for performing a single task.
- Functions are used for code reusability and modularity.
- Basically, functions are separated into two parts: User-Defined Functions and Predefined Functions.
- The function is a set of statements that take inputs, do some specific computation, and produce output.
- Functions are created when certain statements are repeatedly occurring in the program and a function is created to replace them.

- Predefined Functions
- Predefined Functions are nothing but some set of subroutines already coded into a programing language.
- We just call it and use the function in our code. We already use thousands of predefined functions in our daily code. For Example main(), print() and so on.

- User-Defined Functions
- A user-defined function is nothing but a piece of code to perform a specific task or a set of sub-routines created by a developer's wish. One of the powerful features of function is code reusability.

- These are as follows:
- No arguments and no return type
- With arguments and no return type
- No arguments and return type
- With arguments and with return type

- 1. Function with no argument and no return type:

-
  ```
  void myName(){
  print("FlutterDocs");
  }
  void main(){
  print("This is the best website for developers:");
  myName();
  }
  ```

- 2. Function with no arguments but return type:
- int myPrice(){

  int price = 0;

  return price;

  }

  void main(){

  int Price = myPrice();

  print("FlutterDocs is the best website for developers which costs : ${Price}/-");

  }

- 3. Function with arguments but no return type:
- myPrice(int price){

  print(price);

  }

  void main() {

  print("Flutterdocs is the best website for developers which costs : ");

  myPrice(0);

  }

- 4. Function with arguments and with return type:
- int mySum(int firstNumber, int secondNumber) {

  return (firstNumber + secondNumber);

  }

  void main(){

  int additionOfTwoNumber = mySum(100, 500);

  print(additionOfTwoNumber);

  }

Import Package

- To use an external package, library or an external class, use the import statement.
- Separating codelogic into different class files allows you to separate and group code into manageable objects.
- Theimport statement allows access to external packages and classes. It requires only one argument, which specifies the uniform resource identifier (URI) of the class/library.
- If the library is created bya package manager, then you specify the package: scheme before the URI.
- If importing a class, you specify the location and class name or the package: directive.

Examples

- // Import the material package

  import 'package:flutter/material.dart';

- // Import external class

  import 'charts.dart';

- // Import external class in a different folder

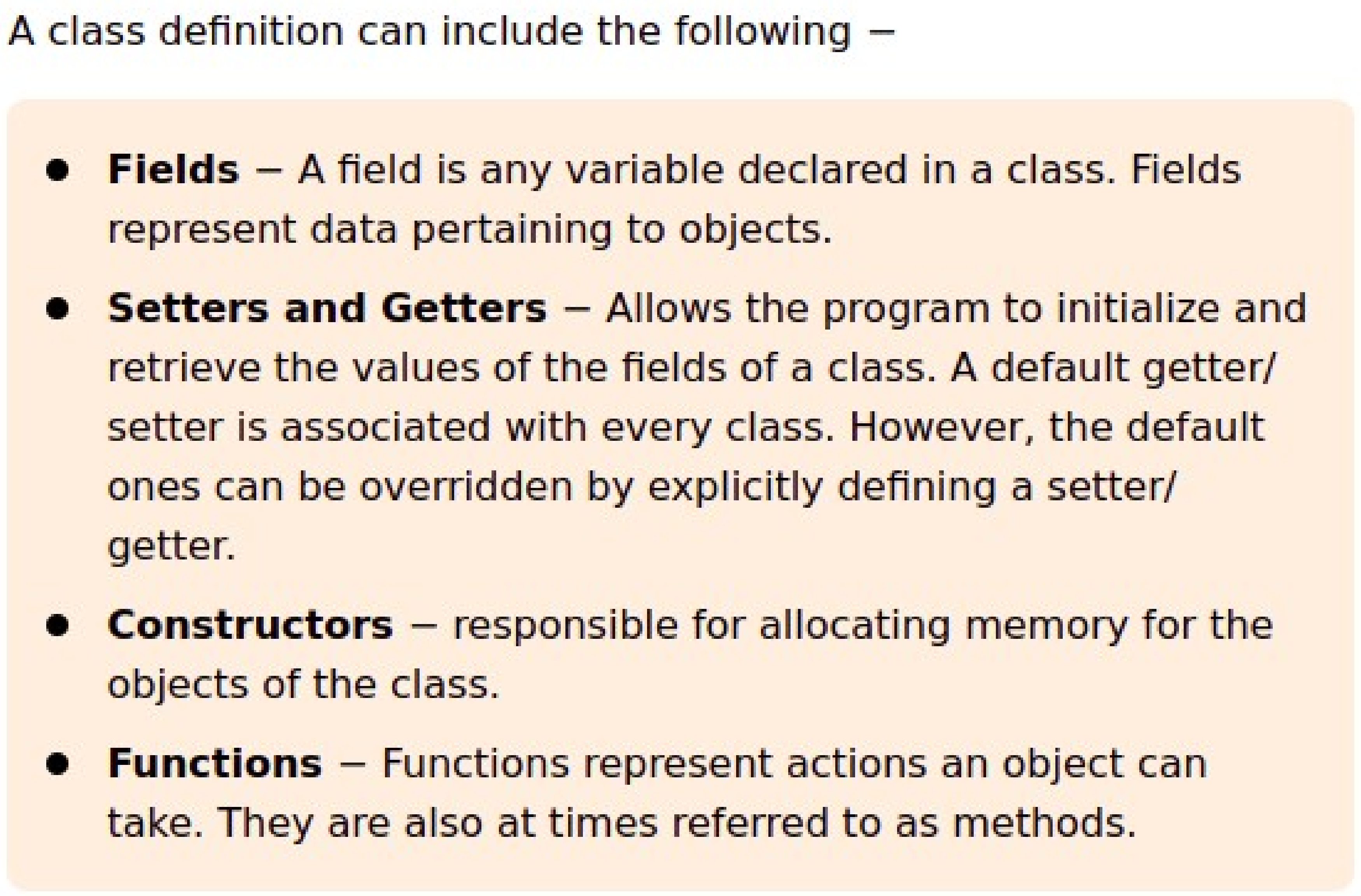
  import 'services/charts_api.dart';

- // Import external class with package: directive

  import 'package:project_name/services/charts_api.dart';

Using classes

- All classes descend from Object , the base class for all Dart objects.
- A class has members (variables and methods) and uses a constructor to create an object. If a constructor is not declared, a default constructor will be provided automatically. The default constructor provided for you has no arguments.
- Dart is an object-oriented language with classes and mixin-based inheritance.
- Every object is an instance of a class, and all classes except Null descend from Object. Mixin-based inheritance means that although every class (except for the top class, Object?) has exactly one superclass, a class body can be reused in multiple class hierarchies



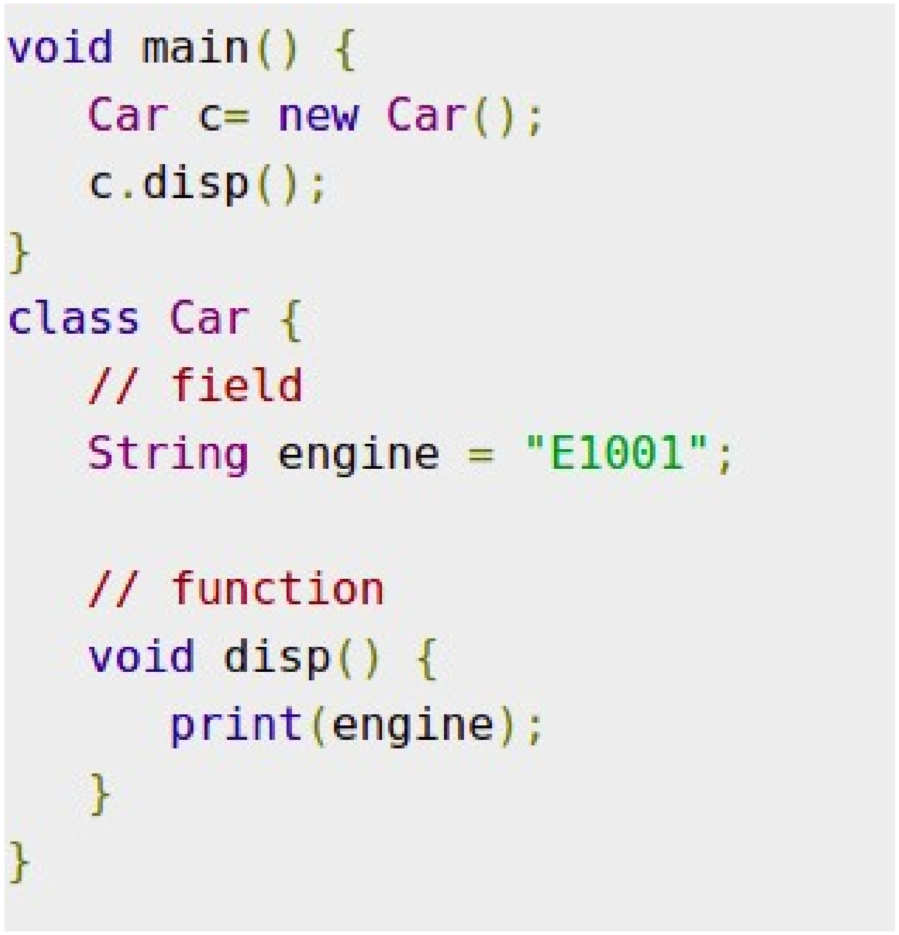
A class definition can include the following –

- **Fields** – A field is any variable declared in a class. Fields represent data pertaining to objects.
- **Setters and Getters** – Allows the program to initialize and retrieve the values of the fields of a class. A default getter/ setter is associated with every class. However, the default ones can be overridden by explicitly defining a setter/ getter.
- **Constructors** – responsible for allocating memory for the objects of the class.
- **Functions** – Functions represent actions an object can take. They are also at times referred to as methods.

Declaring the class

-
```
class Car {
 // field
String engine = "E1001";
// function
 void disp() {
print(engine);
}
}
```

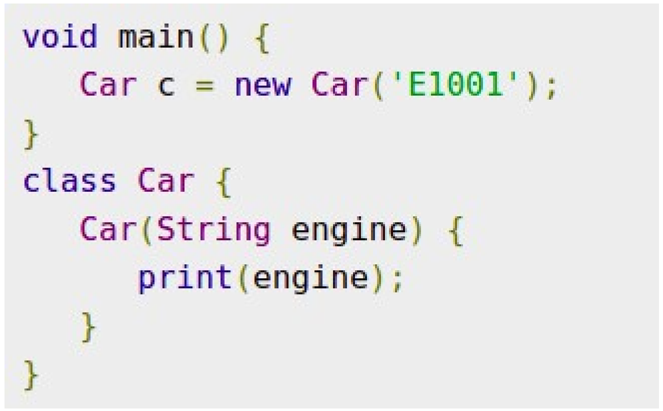
Accessing Attributes and Functions



```
void main() {
   Car c= new Car();
   c.disp();
}
class Car {
   // field
   String engine = "E1001";

   // function
   void disp() {
      print(engine);
   }
}
```

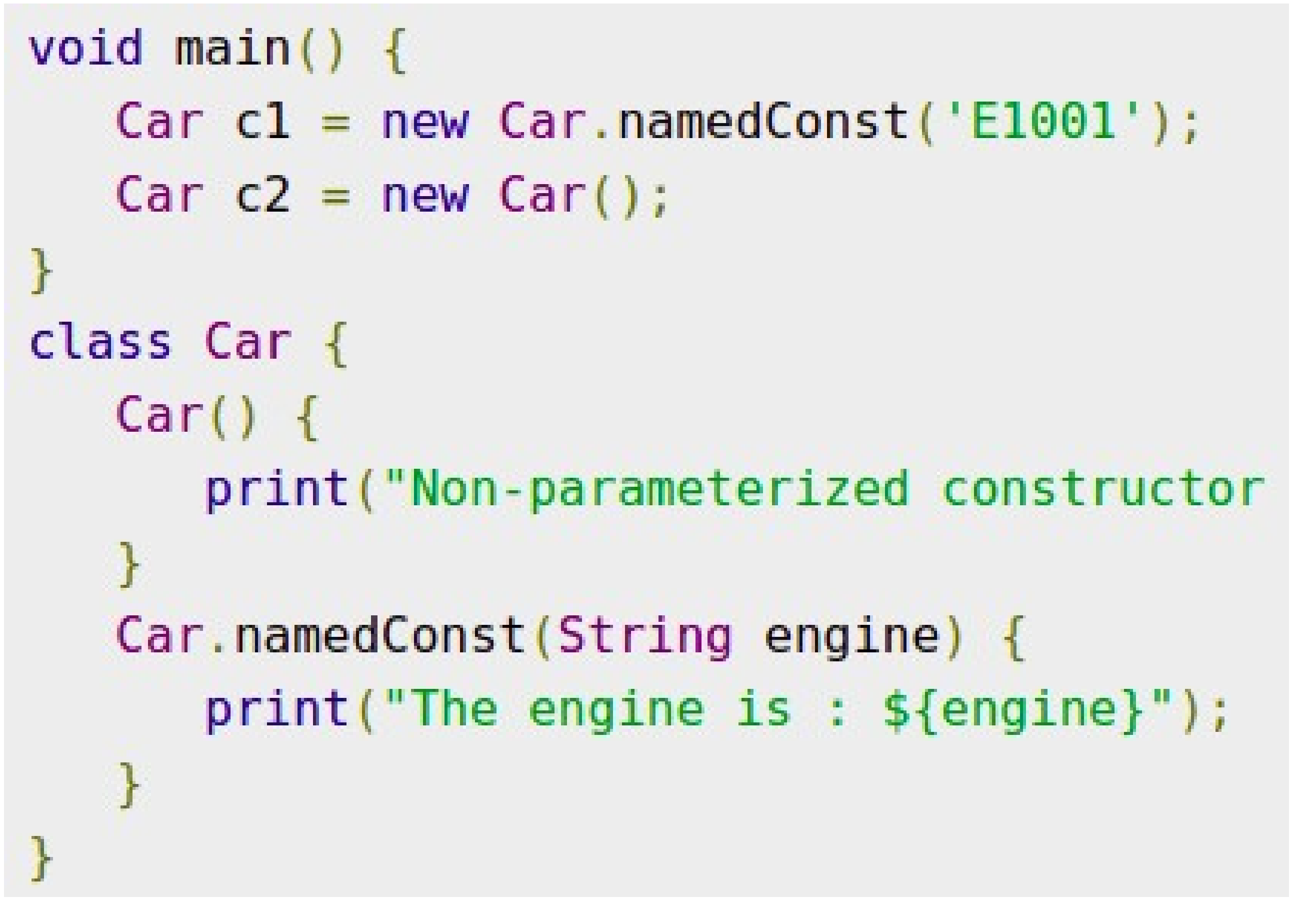
Dart Constructors

- A constructor is a special function of the class that is responsible for initializing the variables of the class.
- Dart defines a constructor with the same name as that of the class.
- A constructor is a function and hence can be parameterized. However, unlike a function, constructors cannot have a return type.
- If you don’t declare a constructor, a default no-argument constructor is provided for you.



```
void main() {
   Car c = new Car('E1001');
}
class Car {
   Car(String engine) {
      print(engine);
   }
}
```

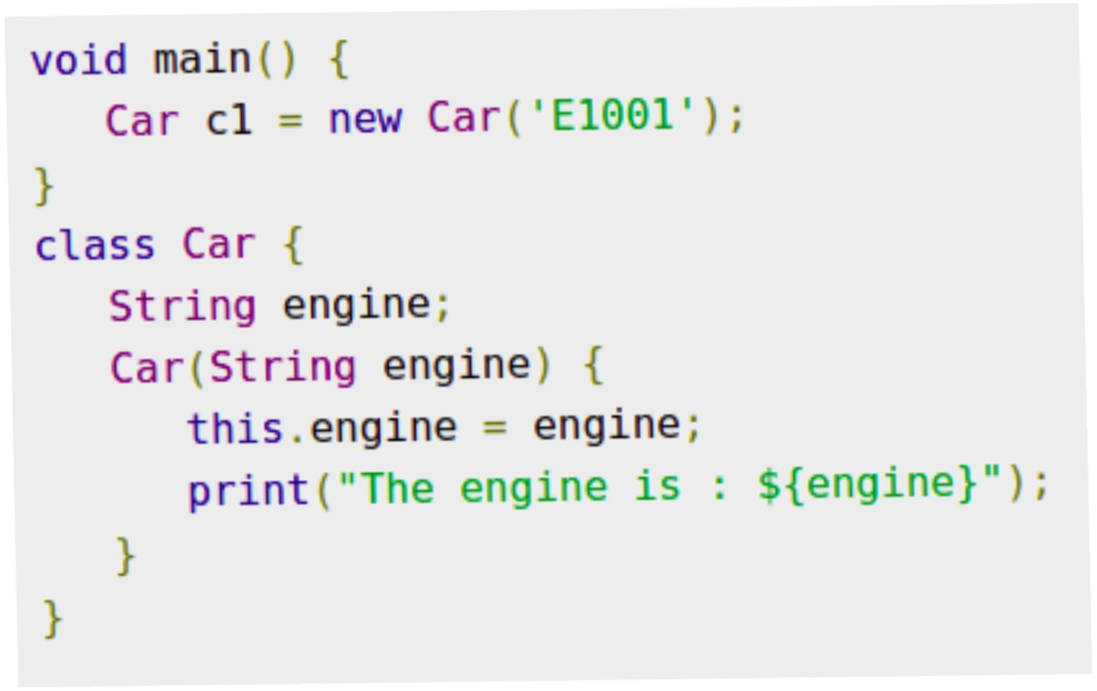
Named Constructors

- Dart provides named constructors to enable a class define multiple constructors.
- 

```
void main() {
   Car c1 = new Car.namedConst('E1001');
   Car c2 = new Car();
}
class Car {
   Car() {
      print("Non-parameterized constructor
   }
   Car.namedConst(String engine) {
      print("The engine is : ${engine}");
   }
}
```

The this Keyword

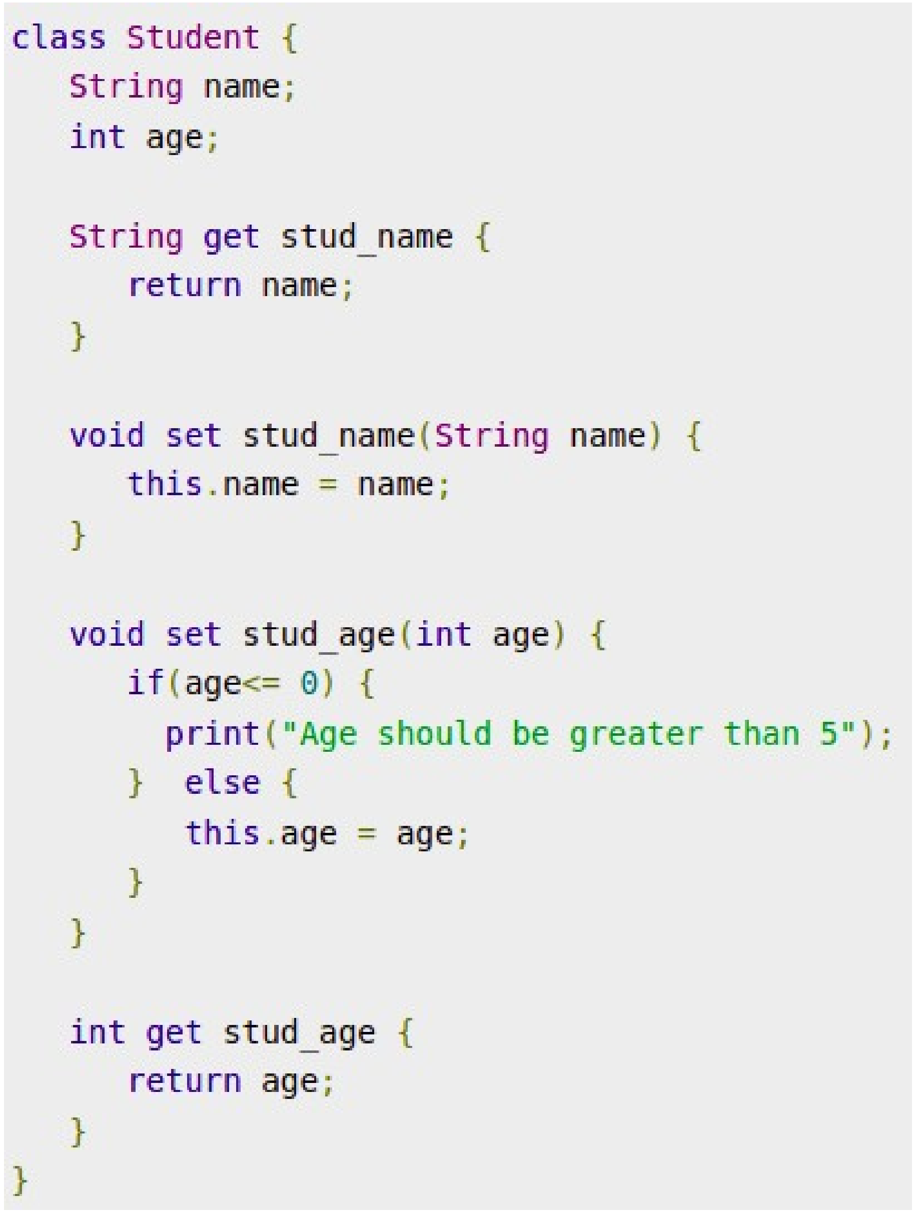
- The this keyword refers to the current instance of the class.
- Here, the parameter name and the name of the class’s field are the same. Hence to avoid ambiguity, the class’s field is prefixed with the this keyword.



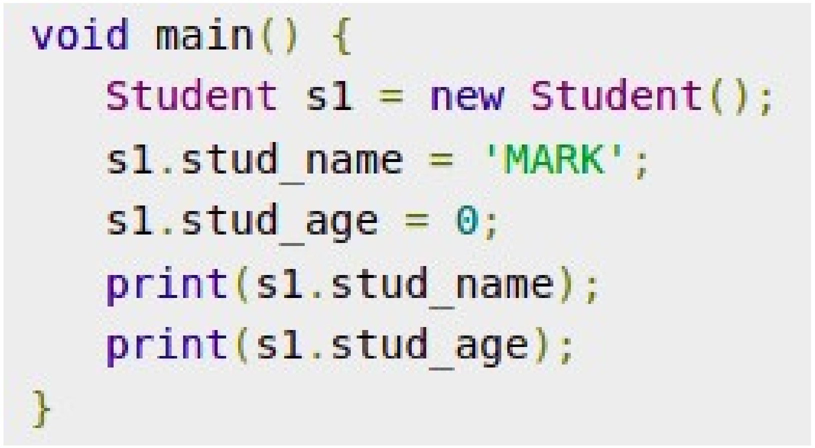
```
void main() {
   Car c1 = new Car('E1001');
}
class Car {
   String engine;
   Car(String engine) {
      this.engine = engine;
      print("The engine is : ${engine}");
   }
}
```

Dart Class – Getters and Setters

- Getters and Setters, also called as accessors and mutators, allow the program to initialize and retrieve the values of class fields respectively.
- Getters or accessors are defined using the get keyword. Setters or mutators are defined using the set keyword.
- A default getter/setter is associated with every class. However, the default ones can be overridden by explicitly defining a setter/ getter.
- A getter has no parameters and returns a value, and the setter has one parameter and does not return a value.



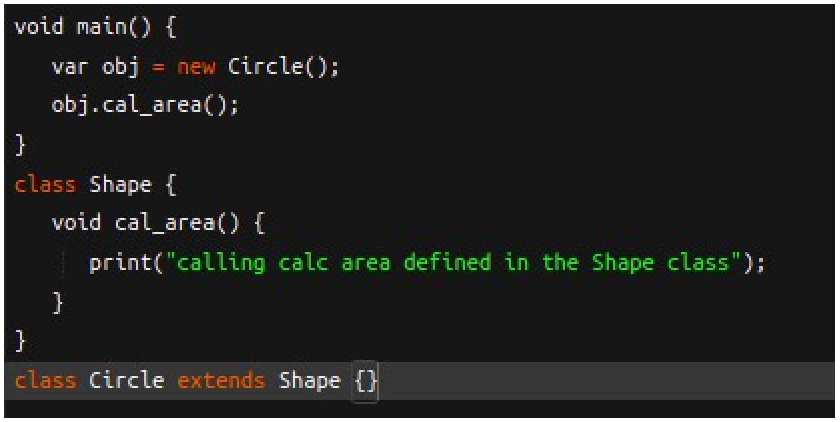
```
class Student {
   String name;
   int age;

   String get stud_name {
      return name;
   }

   void set stud_name(String name) {
      this.name = name;
   }

   void set stud_age(int age) {
      if(age<= 0) {
        print("Age should be greater than 5");
      }  else {
         this.age = age;
      }
   }

   int get stud_age {
      return age;
   }
}
```



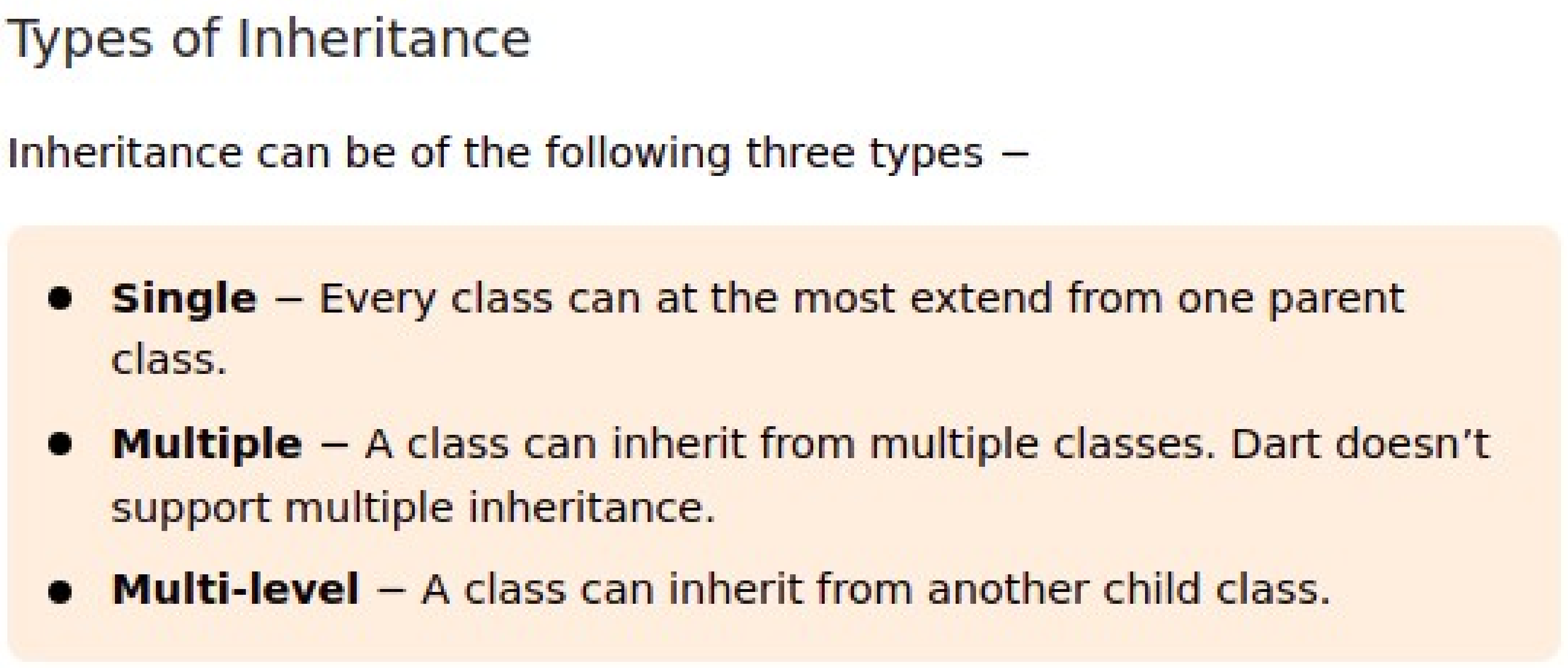
```
void main() {
   Student s1 = new Student();
   s1.stud_name = 'MARK';
   s1.stud_age = 0;
   print(s1.stud_name);
   print(s1.stud_age);
}
```

Class Inheritance

- Dart supports the concept of Inheritance which is the ability of a program to create new classes from an existing class.
- The class that is extended to create newer classes is called the parent class/super class. The newly created classes are called the child/sub classes.
- A class inherits from another class using the ‘extends’ keyword. Child classes inherit all properties and methods except constructors from the parent class.



```
void main() {
   var obj = new Circle();
   obj.cal_area();
}
class Shape {
   void cal_area() {
      print("calling calc area defined in the Shape class");
   }
}
class Circle extends Shape {}
```



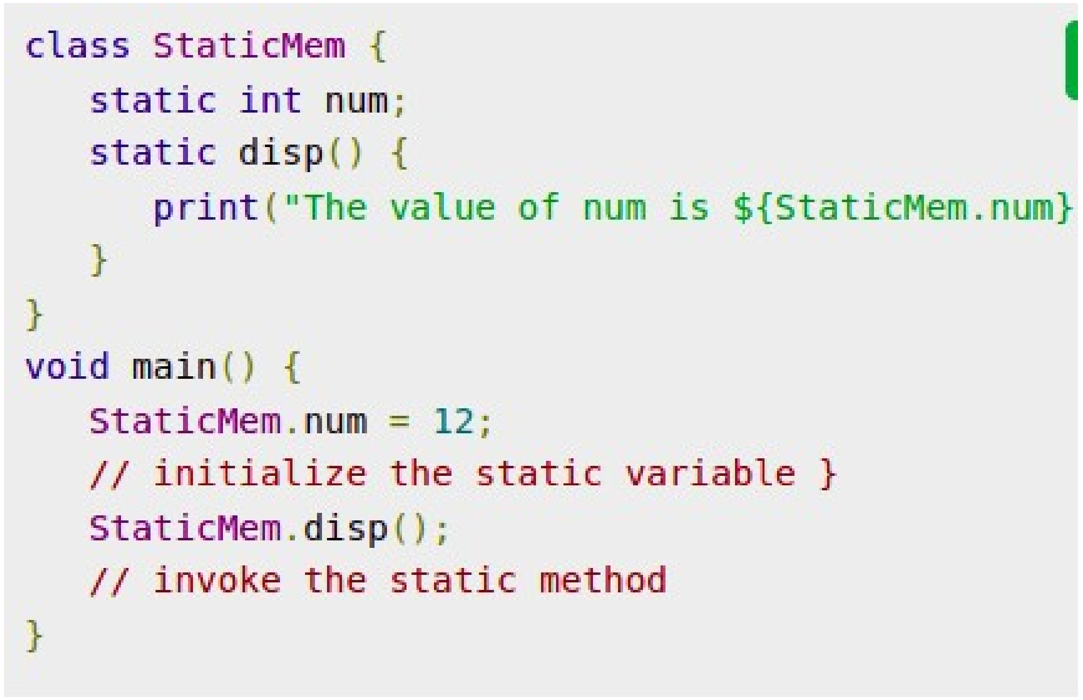
Types of Inheritance

Inheritance can be of the following three types −

- **Single** − Every class can at the most extend from one parent class.
- **Multiple** − A class can inherit from multiple classes. Dart doesn't support multiple inheritance.
- **Multi-level** − A class can inherit from another child class.

The static Keyword

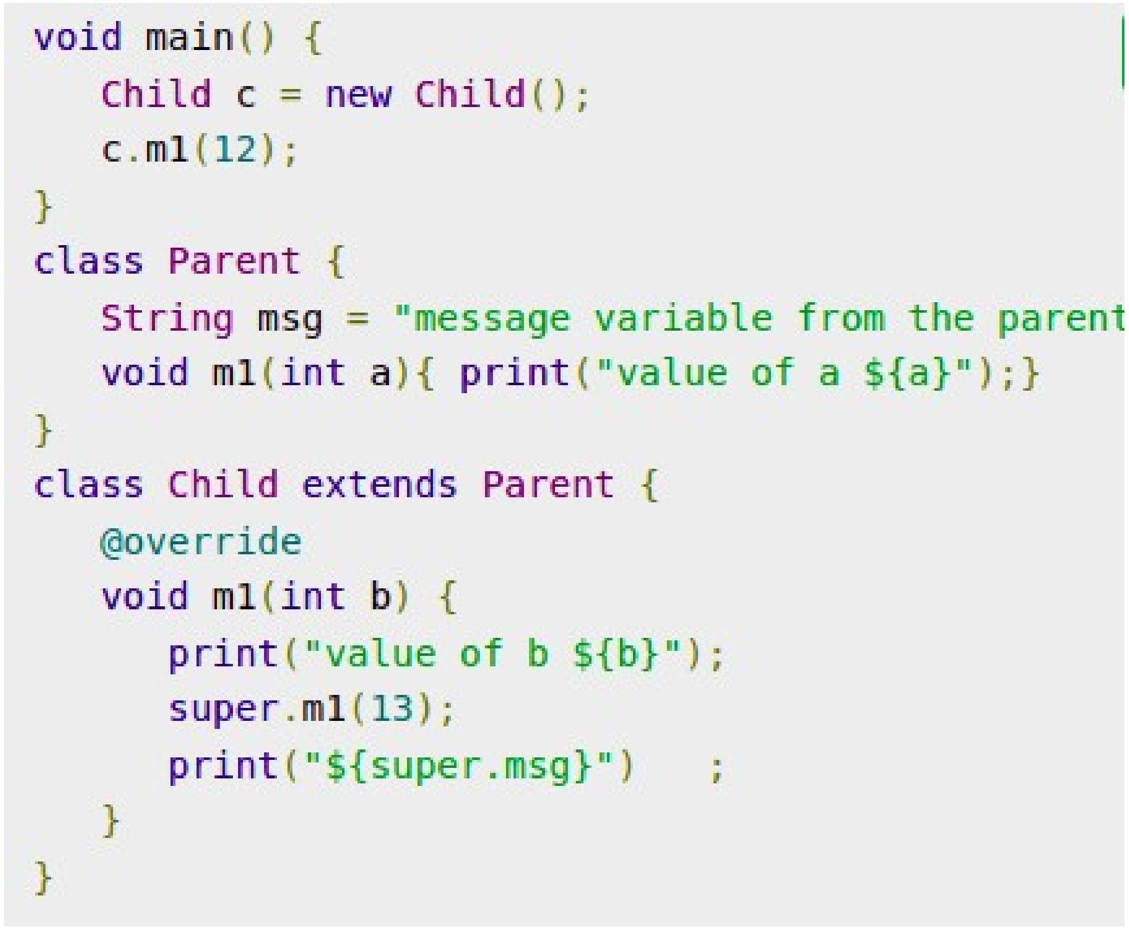
- The static keyword can be applied to the data members of a class, i.e., fields and methods. A static variable retains its values till the program finishes execution. Static members are referenced by the class name.



```
class StaticMem {
   static int num;
   static disp() {
      print("The value of num is ${StaticMem.num}
   }
}
void main() {
   StaticMem.num = 12;
   // initialize the static variable }
   StaticMem.disp();
   // invoke the static method
}
```

The super Keyword

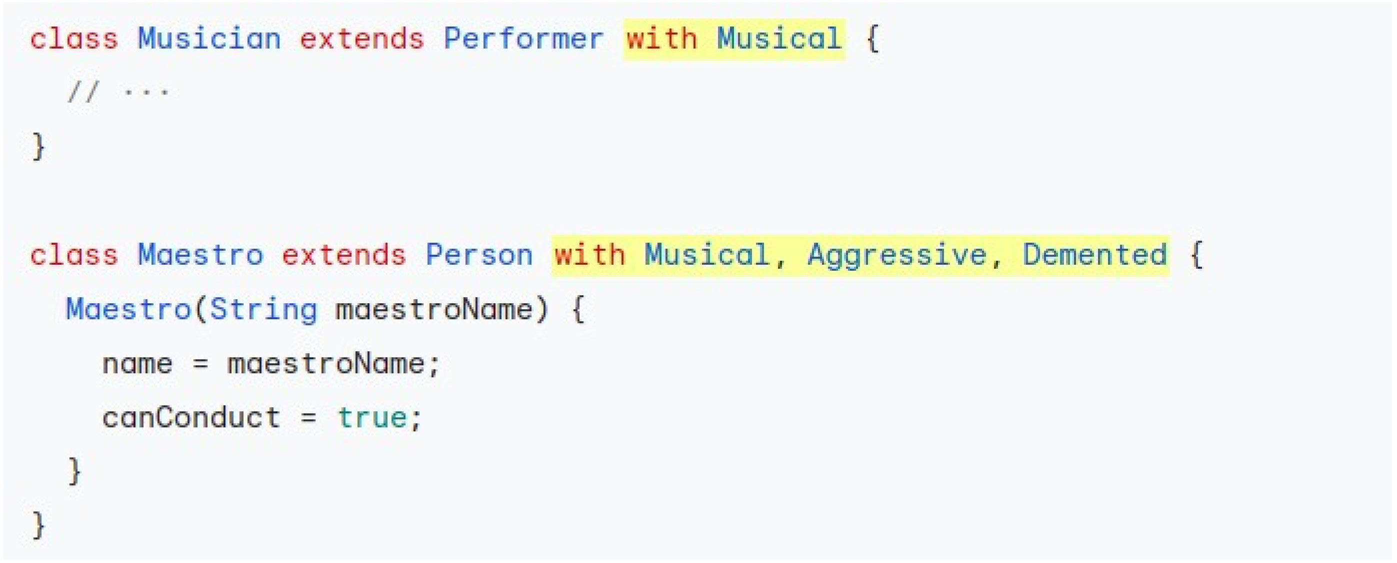
- The super keyword is used to refer to the immediate parent of a class. The keyword can be used to refer to the super class version of a variable, property, or method.



```
void main() {
   Child c = new Child();
   c.m1(12);
}
class Parent {
   String msg = "message variable from the parent
   void m1(int a){ print("value of a ${a}");}
}
class Child extends Parent {
   @override
   void m1(int b) {
      print("value of b ${b}");
      super.m1(13);
      print("${super.msg}")   ;
   }
}
```

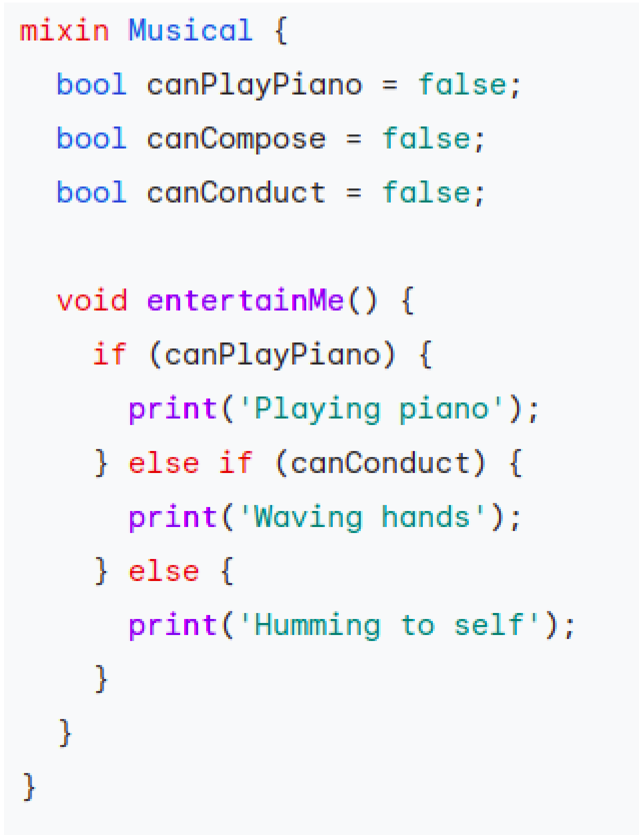
Mixins

- Mixins are a way of defining code that can be reused in multiple class hierarchies. They are intended to provide member implementations en masse.
- To use a mixin, use the with keyword followed by one or more mixin names.

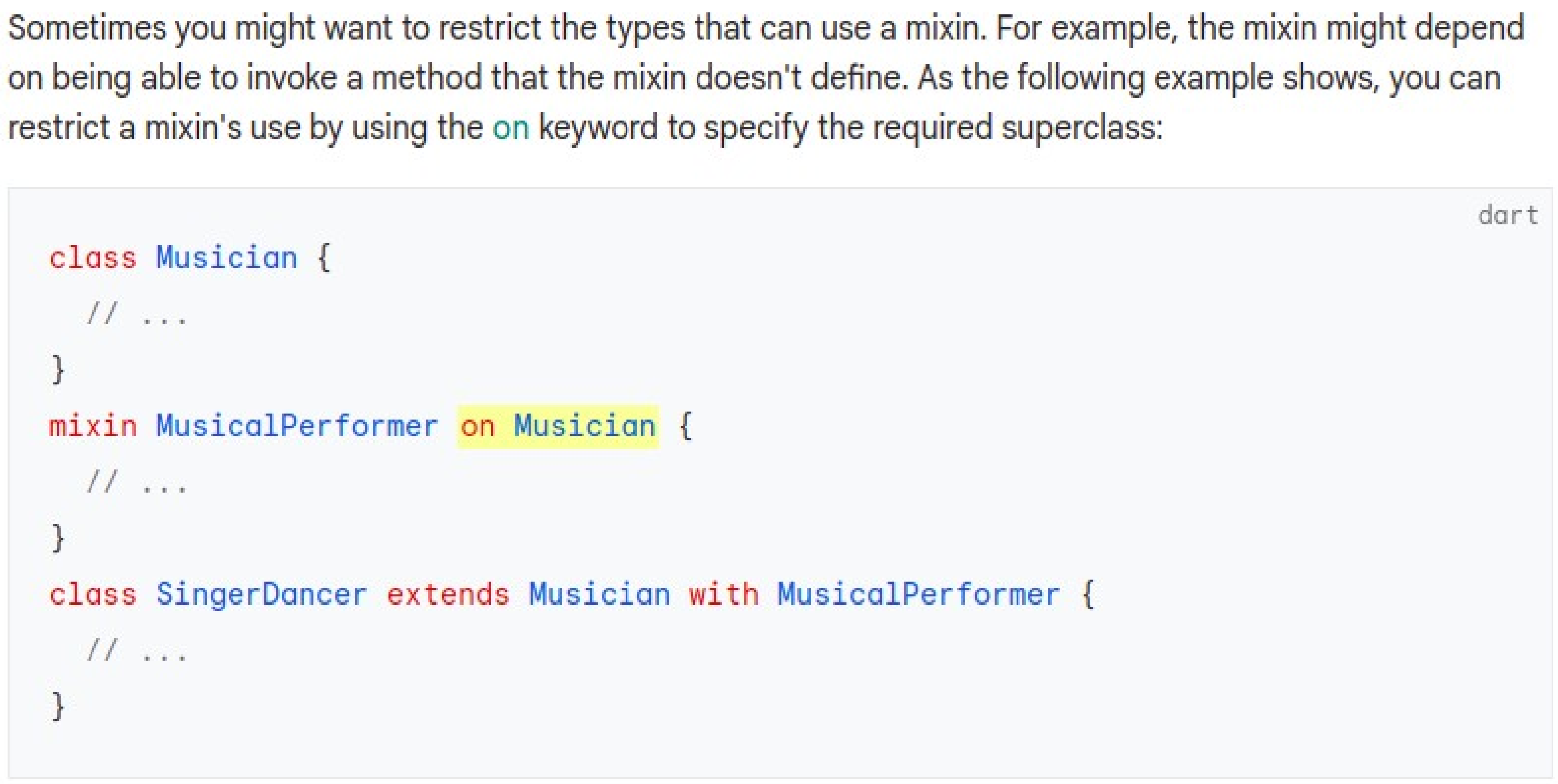


```
class Musician extends Performer with Musical {
  // ···
}

class Maestro extends Person with Musical, Aggressive, Demented {
  Maestro(String maestroName) {
    name = maestroName;
    canConduct = true;
  }
}
```

- To define a mixin, use the mixin declaration. In the rare case where you need to define both a mixin and a class, you can use the mixin class declaration.
- Mixins and mixin classes cannot have an extends clause, and must not declare any generative constructors.



```
mixin Musical {
  bool canPlayPiano = false;
  bool canCompose = false;
  bool canConduct = false;

  void entertainMe() {
    if (canPlayPiano) {
      print('Playing piano');
    } else if (canConduct) {
      print('Waving hands');
    } else {
      print('Humming to self');
    }
  }
}
```



Sometimes you might want to restrict the types that can use a mixin. For example, the mixin might depend on being able to invoke a method that the mixin doesn't define. As the following example shows, you can restrict a mixin's use by using the on keyword to specify the required superclass:

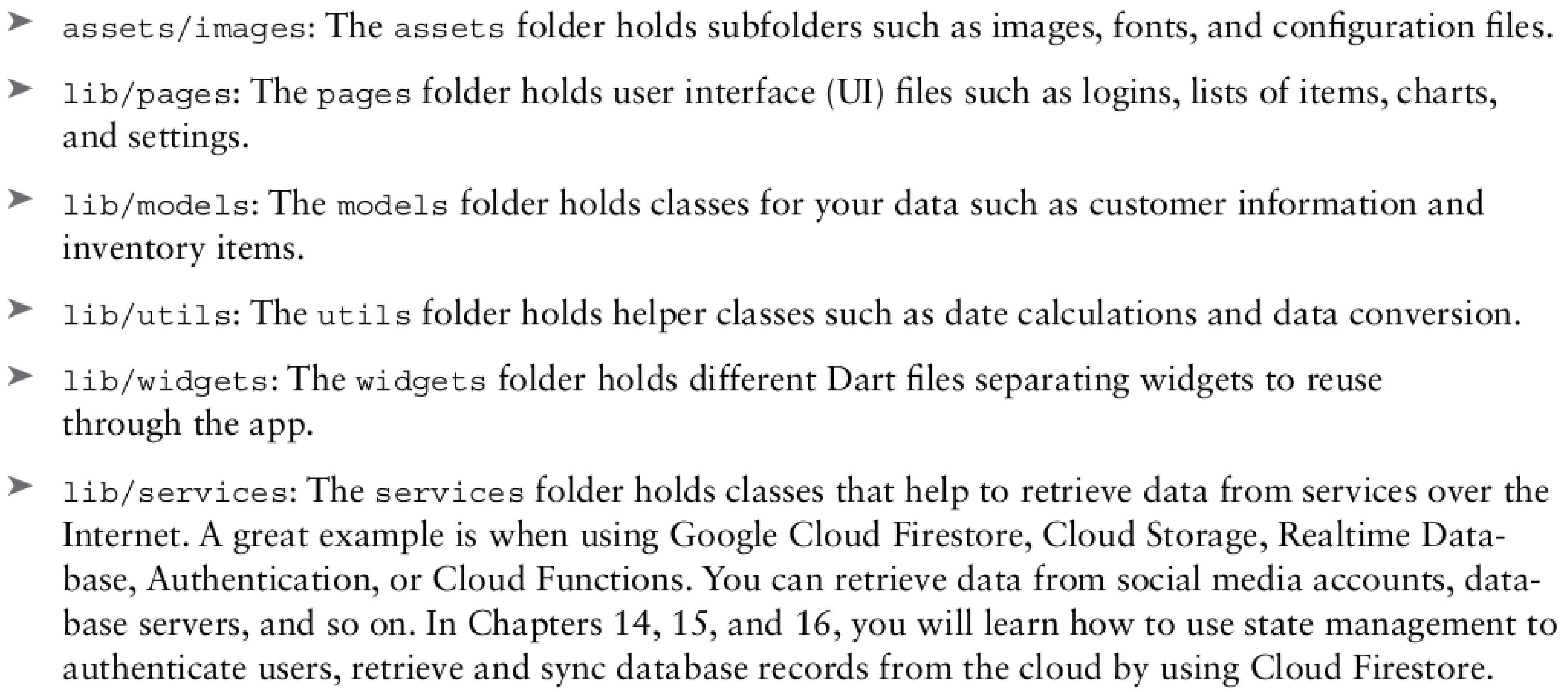
```dart
class Musician {
  // ...
}
mixin MusicalPerformer on Musician {
  // ...
}
class SingerDancer extends Musician with MusicalPerformer {
  // ...
}
```

class, mixin, or mixin class?

- A mixin declaration defines a mixin. A class declaration defines a class.
- A mixin class declaration defines a class that is usable as both a regular class and a mixin, with the same name and the same type.
- Any restrictions that apply to classes or mixins also apply to mixin classes:
  - Mixins can't have extends or with clauses, so neither can a mixin class.
  - Classes can't have an on clause, so neither can a mixin class.

Creating and Organizing Folders and Files

- Configuring the first project in flutter and exploring the folder structure(Android Studio)
- From terminal, aslo create the subfolders
- mkdir -p assets/images



➤ `assets/images`: The `assets` folder holds subfolders such as images, fonts, and configuration files.

➤ `lib/pages`: The `pages` folder holds user interface (UI) files such as logins, lists of items, charts, and settings.

➤ `lib/models`: The `models` folder holds classes for your data such as customer information and inventory items.

➤ `lib/utils`: The `utils` folder holds helper classes such as date calculations and data conversion.

➤ `lib/widgets`: The `widgets` folder holds different Dart files separating widgets to reuse through the app.

➤ `lib/services`: The `services` folder holds classes that help to retrieve data from services over the Internet. A great example is when using Google Cloud Firestore, Cloud Storage, Realtime Database, Authentication, or Cloud Functions. You can retrieve data from social media accounts, database servers, and so on. In Chapters 14, 15, and 16, you will learn how to use state management to authenticate users, retrieve and sync database records from the cloud by using Cloud Firestore.

STRUCTURING WIDGETS

- Structuring widgets in an organized manner improves the code’s readability and maintainability.
- When creating a new Flutter project, the software development kit (SDK) does not automatically create the separate home.dart file, which contains the main presentation page when the app starts.
- Therefore, to have code separation, you must manually create the pages folder and the home.dart file inside it. The main.dart file contains the main() function that starts the app and calls the Home widget in the home.dart file.

- The main.dart file has three main sections.

  ➤ ➤ The import package/file

  ➤ ➤ The main() function

  ➤ ➤ A class that extends a StatelessWidget widget and returns the app as a widget

- Let’s start by adding the code to the main.dart file and saving it.

- Import the package/file. The default import is the material.dart library to allow the use of Material Design. (To use the Cupertino iOS-style widgets, you import the cupertino.dart library instead of material.dart .
- Then import the home.dart page located in the pages folder.
- import 'package:flutter/material.dart';
- import 'package:ch4_starter_exercise/pages/home.dart';

- After the two import statements, leave a blank line and enter the main() function listed next.
- The main() function is the entry point to the app and calls the MyApp class.
- void main() => runApp(MyApp());

Type the MyApp class that extends StatelessWidget.

The MyApp class returns a MaterialApp widget declaring title, theme, and home properties. There are many other MaterialApp properties available. Notice that the home property calls the Home() class, which is created later in the home.dart file.

```
class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      title: 'Starter Template',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: Home(),
    );
  }
}
```