



# Dart

---

## The beginners guide

Prepared By: Aamir Pinger



/AamirPinger



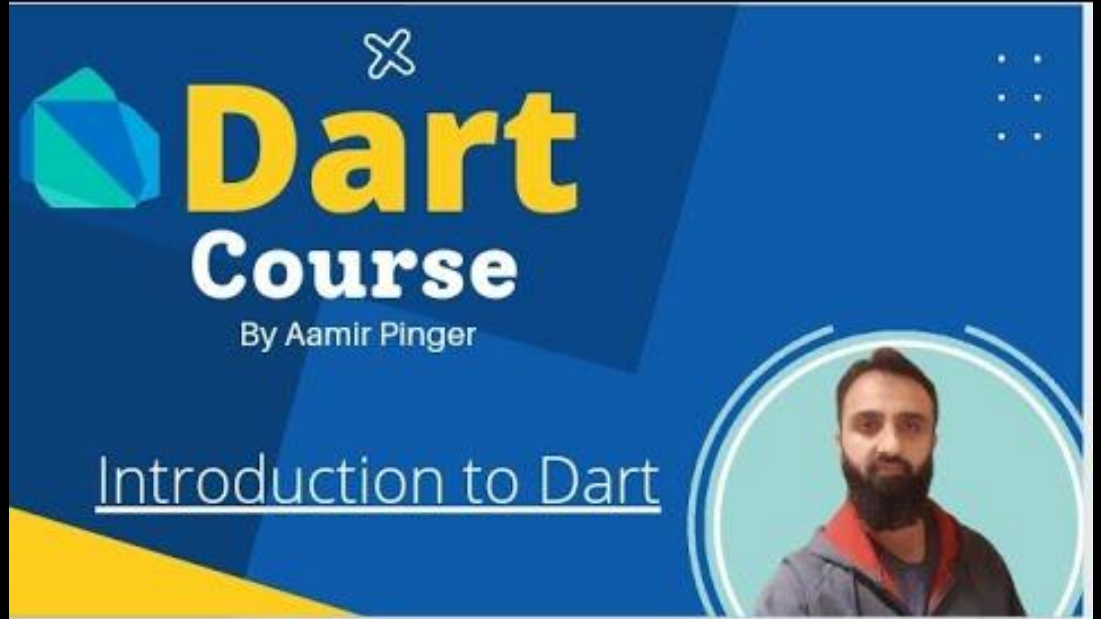
/AamirPinger



/AamirPingerOfficial



[Click for Dart  
Course Playlist](#)

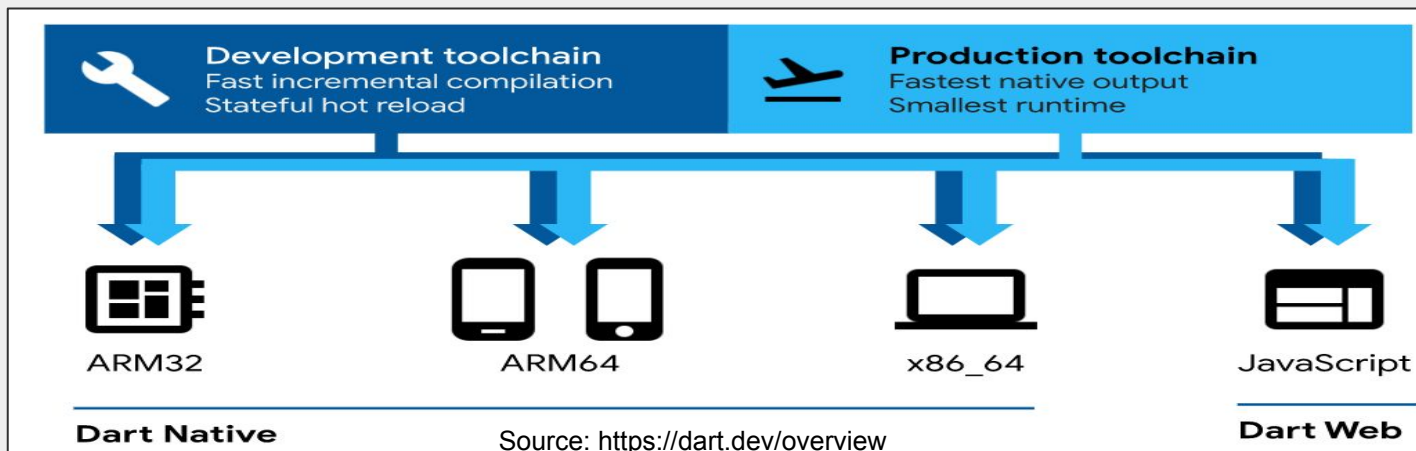


- Dart is an open-source, general-purpose, object-oriented programming language with C-style syntax.
- Developed by Google in 2011.
- A language behind the Flutter apps.
- Dart is a compiled language, the compiler parses it and transfer it into machine code.



# Dart

- **Native platform:** For apps targeting mobile and desktop devices, Dart includes both a Dart VM with just-in-time (JIT) compilation and an ahead-of-time (AOT) compiler for producing native machine code like android or ios.
- **Web platform:** For apps targeting the web, Dart includes both a development time compiler (dartdevc) and a production time compiler (dart2js). Both compilers translate Dart into JavaScript.



- Strongly Typed.
- It supports most of the common concepts of programming languages like
  - Classes,
  - Interfaces,
  - Functions,
  - Collections etc
- Dart offers sound null safety, to protect you from null exceptions at runtime through static code analysis.

Let's do some coding!

---

# main function



```
void main() {  
    print('This is my first print statement using Dart!');  
}
```

- The main() function is the entry point function in Dart.

```
1 void main() {  
2   for (int i = 0; i < 5; i++) {  
3     print('hello ${i + 1}');  
4   }  
5 }  
6
```

▶ Run

Console

# DartPad

<https://dartpad.dev/>



# Dart SDK

---

- The Dart SDK has the libraries and command-line tools that you need to develop Dart web, command-line, and server apps.

**To Install:**

**<https://dart.dev/get-dart>**

```
void main() {  
    print('This is my first print statement from CLI.');
```

msg.dart

```
E:\> dart msg.dart
```

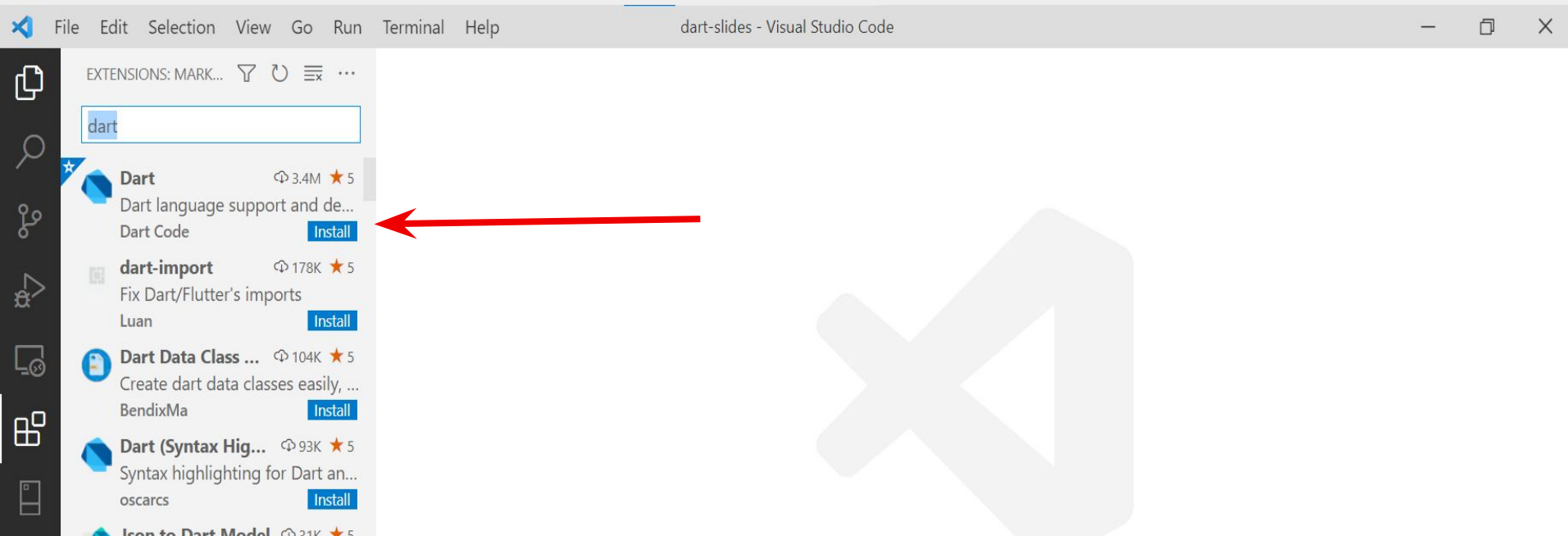
```
This is my first print statement from CLI.
```

# Install VS Code Editor And Dart Extension



To install VS Code

<https://code.visualstudio.com/download>



# Install Android Studio And Dart Extension



## To install Android Studio

<https://code.visualstudio.com/download>

The screenshot shows the Visual Studio Code interface. On the left, the 'Settings' sidebar is open, with the 'Plugins' section selected at the bottom. A red arrow points from the 'Plugins' section in the sidebar to the 'Dart' extension in the main 'Plugins' view. The 'Plugins' view shows a search for 'dart' with 59 results. The 'Dart' extension by JetBrains is listed first, with a download size of 10.5M and a star rating of 3.98. It is currently installed. Below it are other extensions like 'JsonToDart (JSON To Dart)', 'dart.extensions', 'JSON To Dart Class (JsonTo...', 'Json2Dart', and 'Dart Data Class', all with 'Install' buttons. On the right, the 'Dart' extension's details page is shown, including its logo, download size (10.5M), star rating (3.98), and a list of features.

**Plugins**

Search: dart

Search Results (59) Sort By: Relevance

Plugin	Download Size	Star Rating	Status
<b>Dart</b>	10.5M	3.98	Installed
<b>JsonToDart (JSON To Dart)</b>	89.4K	4.70	Install
<b>dart.extensions</b>	55K		Install
<b>JSON To Dart Class (JsonTo...</b>	40.9K	4.59	Install
<b>Json2Dart</b>	79.4K	4.46	Install
<b>Dart Data Class</b>	49.9K	4.76	Install

**Dart**

10.5M 3.98 JetBrains

Languages 202,853

Plugin homepage

Support for Dart

Features

- Smart coding assistance for Dart that includes formatting, navigation, intentions, refactoring, and more.
- Integration with pub and the Dart Analyzer.
- The IDE detects the problems with your code and suggests ways to automatically fix them.
- Run and debug Dart command-line applications and in the IDE using the built-in debugger.
- Run and debug tests.
- Create new Dart projects from the IDE.

Find more information on [getting started with Dart](#).

Size: 1.5M

# Initializing Dart Project

---

# Initializing Dart Project

---

- Instead of creating a single file we can initialize complete dart project
- It will bring a complete folder structure.
- It will also create other useful files like Readme, pubspec.yaml, .gitignore etc
- It will also implement ts-lint rules to precheck best practices used in the code.
- The name should be all lowercase, with underscores to separate words, my\_new\_project.

```
E:\> dart create my_first_dart_project
```

Creating my\_first\_dart\_project using template console-simple...

```
.gitignore  
analysis_options.yaml  
CHANGELOG.md  
pubspec.yaml  
README.md  
bin\my_first_dart_project.dart
```

Running pub get...

Resolving dependencies...

Changed 1 dependency!

Created project my\_first\_dart\_project in my-first-dart-project! In order to get started, run the following commands:

```
cd my_first_dart_project  
dart run
```

```
E:\> cd my-first-dart-project
```

```
E:\my_first_dart_project> dart run
```



- There are tons of libraries that we can use in our projects
- These libraries help us focus on the core business logic of our project rather than inventing the wheel from scratch for functionalities that are already created by others
- To add any library: for example Colorize Library

```
dart pub add colorize
```

- Dart keep all these dependency name list in **pubspec.yaml** file and creates a **package\_config.json** in **.dart\_tool** folder that contains the details of libraries code.
- If you already all the library list and wanted to fetch library code of all of the dependencies, you use following commands

```
dart pub get
```

- Libraries can be get or publish from:

<https://pub.dev/packages>

# Variables

---

- A variable is “a named space in the memory” that stores values.
- In other words, it acts a container for values in a program.
- Variable names are called identifiers.
- Variables must be declared before they are used
- Following are the naming rules for an identifier –
  - Identifiers cannot be keywords.
  - Identifiers can contain alphabets and numbers.
  - Identifiers cannot contain spaces and special characters, except the underscore (\_) and the dollar (\$) sign.
  - Variable names cannot begin with a number.
  - Variable name are case sensitive.

# Variables syntax



- `var <variable_name>;`
  - `var <name> = <expression>;`
  - `<type> <variable_name>;`
  - `<type> <variable_name> = <expression>;`
- In dart, uninitialized variables are provided with an initial value of null.

```
void main() {  
  var name;  
  print(name);  
}
```

null

```
void main() {  
  String name;  
  print(name);  
}
```

```
Error: Non-nullable variable  
'name' must be assigned  
before it can be used.  
  print(name);  
    ^^^^^  
Error: Compilation failed.
```

- Numbers

- `int val = 10;    // 10`
- `double val = 10.50;    // 10.50`

- Strings

- `String val = "aamir";    // aamir`

Any wrong type assignment will through an  
error

- Boolean

- `bool val = true;    // true`

- Dynamic (any type - decides at runtime)

- `dynamic val = "aamir";`
- `print(val);    // aamir`
- `val = 10;`
- `print(val);    // 10`

# Is dart statically typed / type-safe language?

- Yes, Dart 2 is statically typed.
- Dart has combination of static and runtime checks.
- Dart has a sound type system, which guarantees that an expression of one type cannot produce a value of another type.
- **Does not compile if type mismatched, No surprises!**
- Even with type-safe Dart, you can annotate any variable with dynamic if you need the flexibility of a dynamic language.
- The dynamic type itself is static, but can contain any type at runtime.
- Of course, that removes many of the benefits of a type-safe language for that variable.

# Type inference with VAR



```
void main() {  
  var name = "aamir";  
  name = "pinger";  
  name = 10;  
  print(name);  
}
```

```
Error compiling to JavaScript:  
Warning: Interpreting this as package URI, 'package:dartpad_sample/main.dart'.  
lib/main.dart:4:8:  
Error: A value of type 'int' can't be assigned to a variable of type 'String'.  
name = 10;  
  ^  
Error: Compilation failed.
```



# Final and Const keywords

---

# final / const keywords

- Dart supports the assignment of constant value to a variable.
- These are done by the use of the following keyword:
  - **const** keyword ---> must be initialized at compile time with value.
  - **final** keyword ---> can be initialized at compile time with value.
- These keywords are used to make variable read only.
- Final and Const treat List and Maps differently, we will discuss it with Lists and Maps topic

```
void main() {  
    final name = "aamir";  
    final upperCaseName = name.toUpperCase();  
    print(upperCaseName);  
}  
  
// AAMIR
```

```
void main() {  
    const name = "aamir";  
    const upperCaseName = name.toUpperCase();  
    print(upperCaseName);  
}
```

```
Error: Method invocation is not a  
constant expression.  
    const upperCaseName =  
    name.toUpperCase();
```

# Strings

---

# String concatenation (+) & interpolation \${}



```
void main() {  
  String firstName = "Aamir";  
  String lastName = "Pinger";  
  int age = 40;
```

```
  // String concatenation
```

```
  print("My name is " + firstName + " " + lastName);
```

```
  // String interpolation
```

```
  print("My name is $firstName $lastName, and my age is $age");
```

```
  print("My name is ${firstName} ${lastName}, and my age is ${age}");
```

```
  print("I will be ${age + 1} next year!");
```

```
}
```

```
My name is Aamir Pinger  
My name is Aamir Pinger, and my age is 40  
My name is Aamir Pinger, and my age is 40  
I will be 41 next year!
```



# Exercise

---

# Exercise - String interpolation $\${}$



- Write a table of value provided in variable num till 10.
- If num is 2 then output should be
- When you change the value of num and re-run the main function output should also change.
- **Important:** as we haven't covered loop yet, do not use any loops simply print the table manually.

```
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
2 x 10 = 20
```

# Exercise - String interpolation `${}` - Solution



```
void main() {  
  int num = 2;  
  
  print("$num x 1 = ${num*1}");  
  print("$num x 2 = ${num*2}");  
  print("$num x 3 = ${num*3}");  
  print("$num x 4 = ${num*4}");  
  print("$num x 5 = ${num*5}");  
  print("$num x 6 = ${num*6}");  
  print("$num x 7 = ${num*7}");  
  print("$num x 8 = ${num*8}");  
  print("$num x 9 = ${num*9}");  
  print("$num x 10 = ${num*10}");  
}
```

```
2 x 1 = 2  
2 x 2 = 4  
2 x 3 = 6  
2 x 4 = 8  
2 x 5 = 10  
2 x 6 = 12  
2 x 7 = 14  
2 x 8 = 16  
2 x 9 = 18  
2 x 10 = 20
```

# String Escaping



```
void main(){  
  int amount = 20;  
  
  print('Amount is ${amount}$');  
  print('I'm a software engineer');  
}
```



```
Error compiling to JavaScript:  
Warning: Interpreting this as package URI,  
'package:dartpad_sample/main.dart'.  
lib/main.dart:4:28:  
Error: A '$' has special meaning inside a  
string, and must be followed by an  
identifier or an expression in curly braces  
({}).  
  print('Amout is ${amount}$');  
                        ^  
Error: Compilation failed.
```

```
void main(){  
  int amount = 20;  
  
  // three ways to do it  
  print('Amount is ${amount}\$');  
  print("I'm a software engineer");  
  print('I\'m a software engineer');  
}
```




```
Amount is 20$  
I'm a software engineer  
I'm a software engineer
```




# Multiline String

```
void main(){  
  print("My name is Aamir. I am a  
software engineer.");  
}
```




My name is Aamir. I am a software engineer.

```
void main(){  
  print("My name is Aamir.\n I am a  
software engineer.");  
}
```



My name is Aamir.  
I am a software engineer.

```
void main(){  
  print("""  
My name is Aamir.  
I am a software engineer.  
"""); }  
"""
```



My name is Aamir.  
I am a software engineer.

# Other common string functions



## Methods

- `variable.toUpperCase()`
- `variable.toLowerCase()`
- `variable.contains(value)`
- `variable.replaceFirst(oldValue, newValue, [int startIndex = 0]);`
- `variable.replaceAll(oldValue, newValue);`
- `variable.replaceRange(startIndex, endIndex, newValue);`
- `variable.substring(startIndex, endIndex);`
- `variable.split(pattern)`

// "aamir".toUpperCase(); -----> AAMIR

// "AAMIR".toLowerCase(); -----> aamir

// "aamir".contains('m'); -----> true

// "aamir".replaceFirst('m','x'); -----> aaxir

// "aamir".replaceAll('a','x'); -----> xxmir

// "aamir".replaceRange(1,4,'x'); -----> axxxr

// "aamir".substring(1,4); -----> ami

// "aamir-pinger".split('-'); -----> ['aamir','pinger']

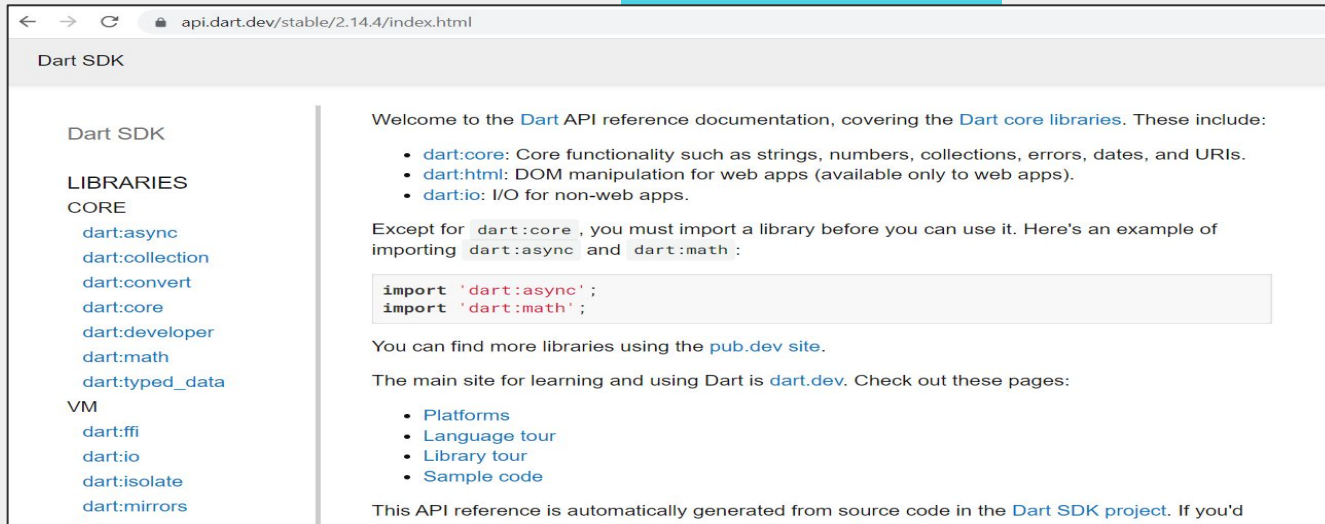
# Other common string functions

- `variable.trim()` // to remove white spaces from left and right     `// " aamir ".trim(); -----> aamir`
- `variable.trimLeft()` // to remove white spaces from left     `// " aamir ".trimLeft('-'); -----> aamir`
- `variable.trimRight()` // to remove white spaces from right     `// " aamir ".trimRight('-'); -----> aamir`
- `variable.codeUnitAt(index)` // to get ascii code     `// "Aamir".codeUnitAt(0); -----> 65`
- `variable.indexOf(pattern)`     `// "Aamir".indexOf('m'); -----> 2`
- `variable.lastIndexOf(patter)`     `// "aamir".lastIndexOf('m'); -----> 1`

## Properties

- `variable.isEmpty` // true if string is empty
- `variable.isNotEmpty` // true if string has some values
- `variable.length` // true if string has some values

- If you are using Android studio, ctrl+B will get you the implementation of anything if available, for example, when cursor is at String, press Ctrl+B.
- You can do the same with ctrl+mouse click in both android studio and vs code.
- Dart also offers rich documentation at <http://api.dart.dev>



# Numbers

---

# Arithmetic operations



```
void main(){  
  int a = 9;  
  int b = 5;  
  
  print('addition: ${a + b}');           // addition: 14  
  print('subtraction: ${a - b}');       // subtraction: 4  
  print('multiplication: ${a * b}');    // multiplication: 45  
  print('division: ${a / b}');          // division: 1.8  
  print('integer part of division: ${a ~/ b}'); // integer part of division: 1  
  print('remainder: ${a % b}');        // remainder: 4  
}
```

# Arithmetic operations

- We can use \* (multiplication sign) with string as well.

```
void main()  
{  
    String str = 'x';  
    print(str);      // x  
    print(str * 2);  // xx  
    print(str * 3);  // xxx  
    print(str * 4);  // xxxx  
    print(str * 5);  // xxxxx  
}
```

# Arithmetic operations



```
void main() {  
    int a = 9;  
    a = a + 1;  
    print(a); // 10  
    a += 1;  
    print(a); // 11  
    a++;  
    print(a); // 12  
    a--;  
    print(a); // 11  
    print(a++); // 11  
    print(++a); // 13  
    print(--a); // 12  
}
```



# Arithmetic operations

```
void main() {  
  int a = 9;  
  
  a = a / 5;  
  print(a);  
}
```



```
Error: A value of type 'double' can't be  
assigned to a variable of type 'int'.  
  a = a / 5;  
      ^
```

```
void main() {  
  int a = 9;  
  
  a = a ~/ 5;  
  print(a);    // 1  
}
```



```
void main() {  
  double a = 9;  
  
  a = a / 5;  
  print(a);    // 1.8  
}
```



# Exercise

---

# Exercise - Arithmetic operations



## Guess the output!

```
void main() {  
    int a = 9;  
    int b = 5;  
  
    print('addition: ${--a + b}');           // addition: 13  
    a += a;  
    a++;  
    ++a;  
    print('addition: ${a++ + b}');           // addition: 23  
}
```

# Other common number functions



## Methods

- `variable.toString()` `// converts number to string`
- `variable.toStringAsFixed(number)` `// converts int/string to double`  
`with fixed decimal points`
- `variable.toDouble()`

# Further read



- Further read on variables

<https://www.geeksforgeeks.org/variables-and-keywords-in-dart/>

# Operators

---

# Relational Operators



OPERATOR	DESCRIPTION	EXAMPLE
>	greater than	<code>5&gt;5 // false</code>
<	Less than	<code>5&lt;5 // false</code>
>=	greater than or equal to	<code>5&gt;=5 // true</code>
<=	less than or equal to	<code>5&lt;=5 // false</code>
==	is equal to	<code>5==2 // false</code>
!=	not equal to	<code>5!=2 // true</code>

# Logical Operators



OPERATOR	NAME	DESCRIPTION	EXAMPLE
&&	Logical AND	return true if all expression are true	<code>(5&gt;2 &amp;&amp; 2&gt;3) // false</code>
	Logical OR	return true if any expression is true	<code>(5&gt;2    2&gt;3) // true</code>
!	Logical NOT	return complement of expression	<code>!(5&gt;2) // false</code>



# Type test operators

OPERATOR	DESCRIPTION	EXAMPLE
<code>is</code>	return true if the object has the specified type	<pre>void main() {     int num = 5;     print(num is int); } // true</pre>
<code>is!</code>	return true if the object has not the specified type	<pre>void main() {     String name = "aamir";     print(name is! int); } // true</pre>

# Ternary Operator



```
condition ? expOnTrue : expOnFalse
```

If condition matches return expOnTrue else return expOnFalse.

```
void main()  
{  
  var res = 10 > 15 ? "Greater":"Smaller";  
  print(res);  
}
```

If / else if / else

---

# if / else if / else

- We can conditionally perform code using if / else if / else.

```
void main()
{
    int myAge = 40;
    int yourAge = 30;

    if(myAge > yourAge){
        print("I am elder than you.");
    } else if(myAge == yourAge){
        print("We are same age.");
    } else {
        print("you are older than me.");
    }
    print("Good luck!");
}
```

```
// I am elder than you.
// Good luck!
```

# Conditionally assigning value



```
void main()  
{  
  int myAge = 40;  
  int yourAge = 30;  
  String msg;  
  
  if(myAge > yourAge){  
    msg = "I am elder than you.";  
  } else if(myAge == yourAge){  
    msg = "We are same age.";  
  } else {  
    msg = "you are older than me.";  
  }  
  
  print(msg);  
  print("Good luck!");  
}
```

```
// I am elder than you.  
// Good luck!
```

# Exercise

---

# Exercise - if / else if /else

Write a function to print grades with following criteria.

**marks = 0.70**

- Percentage **greater than or equal to** 90% print Grade A
- Percentage **greater than or equal to** 80% print Grade B
- Percentage **greater than or equal to** 70% print Grade C
- Percentage **greater than or equal to** 60% print Grade D
- Percentage **greater than or equal to** 40% print Grade E
- Percentage **less than** 40% print Grade F

```
void main()
{
    const marks = 0.70;
    String grade;

    if(marks >= 0.90) {
        grade = "A";
    } else if(marks >= 0.80) {
        grade = "B";
    } else if(marks >= 0.70) {
        grade = "C";
    } else if(marks >= 0.60) {
        grade = "D";
    } else if(marks >= 0.40) {
        grade = "E";
    } else {
        grade = "F";
    }

    print("Grade $grade");
}
```

// Grade C



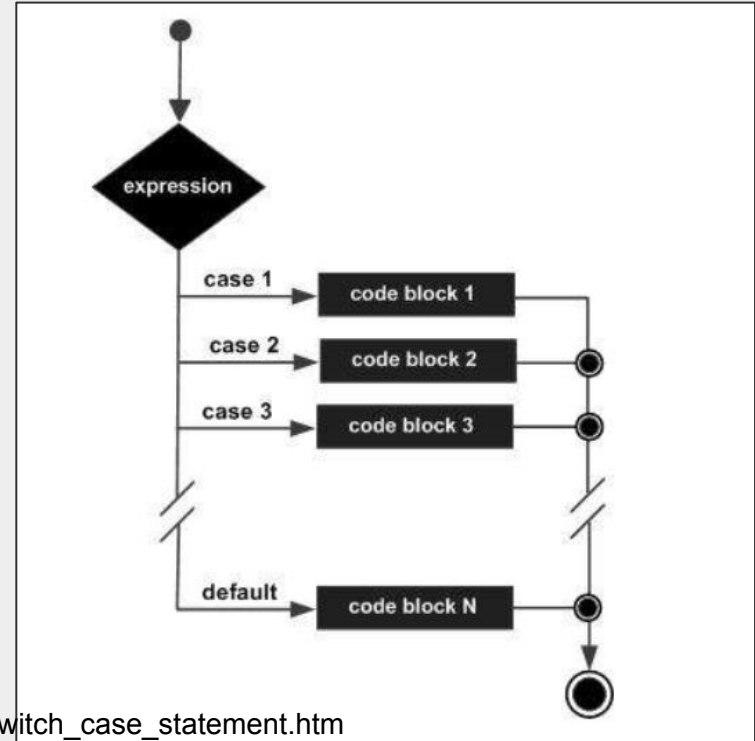
# Switch .. Case

---

# Switch Case

- The switch statement evaluates an expression, matches the expression's value to a case clause and executes the statements associated with that case.
- More clean and readable than **if / else**.

```
switch(variable_expression) {  
    case constant_expr1:  
        // statements;  
        break;  
    case constant_expr2:  
        //statements;  
        break;  
  
    default:  
        //statements;  
}
```



# Switch Case



A very important note:

- A switch statement can only be used with constant expressions like enums, int, String, double...

```
void main() {  
    int day = 5;  
  
    switch (day) {  
        case 1:  
            print('Monday');  
            break;  
        case 2:  
            print('Tuesday');  
            break;  
        case 3:  
            print('Wednesday');  
            break;  
        case 4:  
            print('Thursday');  
            break;  
        case 5:  
            print('Friday');  
            break;  
        case 6:  
            print('Saturday');  
            break;
```

```
        case 7:  
            print('Sunday');  
            break;  
        default:  
            print('Invalid Week day');  
    }  
} // Friday
```

Tip:

- **Default** section is optional
- **Default** section does not need **break** statement
- You can also use { } in all cases/default section

```
case 7: {  
    print('Sunday');  
    break;  
}
```

# Loops

---

# While Loops

---

# While loop

- There will be many situation where we need to execute same piece of code as iterations.
- Basic example could, print all the names of in the List.

```
void main()  
{  
  List names = ["Aamir", "Pinger",  
"Ali"];  
  int i = 0;  
  while(i < names.length) {  
    print(names[i]);  
    i++;  
  }  
}
```

```
// Aamir  
// Pinger  
// Ali
```

← don't forget to increment the number, otherwise it will become an infinite loop.

# Exercise

---



# Exercise - while loop

- Write a table of value provided in variable num till 10 **using while loop**
- If num is 2 then output should be

```
2 x 1 = 2  
2 x 2 = 4  
2 x 3 = 6  
2 x 4 = 8  
2 x 5 = 10  
2 x 6 = 12  
2 x 7 = 14  
2 x 8 = 16  
2 x 9 = 18  
2 x 10 = 20
```

```
2 x 1 = 2  
2 x 2 = 4  
2 x 3 = 6  
2 x 4 = 8  
2 x 5 = 10  
2 x 6 = 12  
2 x 7 = 14  
2 x 8 = 16  
2 x 9 = 18  
2 x 10 = 20
```

- When you change the value of num and re-run the main function output should also change.

# Exercise - Solution using while loop



```
void main()  
{  
    int tableFor = 2;  
    int i = 1;  
    while(i <= 10) {  
        print("$tableFor x $i = ${tableFor * i}");  
        i++;  
    }  
}
```

```
2 x 1 = 2  
2 x 2 = 4  
2 x 3 = 6  
2 x 4 = 8  
2 x 5 = 10  
2 x 6 = 12  
2 x 7 = 14  
2 x 8 = 16  
2 x 9 = 18  
2 x 10 = 20
```

# For Loop

---

# for loop

- Similar to while loop we have a for loop that has initialization, condition check and update iteration variable as a part of syntax
- Easy to read and less risky to become infinite loop.

```
void main()  
{  
    List names = ["Aamir", "Pinger", "Ali"];  
    for(int i=0; i < names.length; i++){  
        print(names[i]);  
    }  
}
```

```
// Aamir  
// Pinger  
// Ali
```

# Exercise

---

# Exercise - for loop

- Write a table of value provided in variable num till 10 **using for loop**
- If num is 2 then output should be

```
2 x 1 = 2  
2 x 2 = 4  
2 x 3 = 6  
2 x 4 = 8  
2 x 5 = 10  
2 x 6 = 12  
2 x 7 = 14  
2 x 8 = 16  
2 x 9 = 18  
2 x 10 = 20
```

```
2 x 1 = 2  
2 x 2 = 4  
2 x 3 = 6  
2 x 4 = 8  
2 x 5 = 10  
2 x 6 = 12  
2 x 7 = 14  
2 x 8 = 16  
2 x 9 = 18  
2 x 10 = 20
```

- When you change the value of num and re-run the main function output should also change.

# Exercise - Solution using for loop



```
void main()  
{  
    int tableFor = 2;  
    for(int i=1; i <= 10; i++){  
        print("$tableFor x $i = ${tableFor * i}");  
    }  
}
```

```
2 x 1 = 2  
2 x 2 = 4  
2 x 3 = 6  
2 x 4 = 8  
2 x 5 = 10  
2 x 6 = 12  
2 x 7 = 14  
2 x 8 = 16  
2 x 9 = 18  
2 x 10 = 20
```

break / continue

---



# Break / Continue with for loop

- To exit loop even it has not completed all iteration, we use **break**.
- To skip rest of the code within loop and starting a new iteration, we use **continue**.

```
void main()
{
    for (int i=1; i<=10; i++) {
        if(i==5){
            print("Exiting loop");
            break;
        }
        if(i % 3 == 0){
            print("$i is an odd no.");
            continue;
        }

        print("$i is an even no.");
    }
}
```

```
1 is an even no.
2 is an even no.
3 is an odd no.
4 is an even no.
Exiting loop as i = 5
```

# Break / Continue with While loop



```
void main()
{
  List thief = ["suspect 1", "suspect 2", "suspect 3", "suspect 4", "culprit", "suspect 5"];

  int i=0;
  while(i<=thief.length) {
    if(thief[i]=="culprit"){
      print("Main ${thief[i]} is caught, no need to catch further suspects.");
      break;
    }

    print("${thief[i]} is caught");

    if(i % 2 == 0){
      print("${thief[i]} locked up for further interrogation.");
      i++;
      continue;
    }

    print("${thief[i]} is released after initial interrogation");
    i++;
  }
}
```

```
suspect 1 is caught
suspect 1 locked up for further interrogation.
suspect 2 is caught
suspect 2 is released after initial interrogation
suspect 3 is caught
suspect 3 locked up for further interrogation.
suspect 4 is caught
suspect 4 is released after initial interrogation
Main culprit is caught, no need to catch further suspects.
```

# Best practice - loops

- Also mostly it is best practice to use an for instead of a while loop, if only one variable needs to be manipulated.
- If there are multiple conditions/variables which can break the loop, then it is best practice to use a while loop.

**Easy, readable approach with multiple values to handle**

```
void main(){  
  var a = 0, b = 0, c = 0, result;  
  
  while(true){  
    a+=1;  
    b-=2;  
    c++;  
  
    result = a+b+c;  
    if(result<=10){  
      break;  
    }  
  }  
  print(result); // 0  
}
```

# Enum

---

# Enum



- Hard coding the values in the code is bad practice and lead to bugs mostly.
- Also a small mistake can bypass any condition for example

```
void main()  
{  
    String today = 'sunday';  
  
    if(today == 'sunday'){  
        print('Today is holiday!');  
    } else {  
        print('Its a working day');  
    }  
} // Today is holiday!
```

```
void main()  
{  
    String today = 'Sunday';  
  
    if(today == 'sunday'){  
        print('Today is holiday!');  
    } else {  
        print('Its a working day');  
    }  
} // Its a working day
```

- It is always recommended to create a Enums for these kind of values.
- This will never lead to error no matter how many developers are working together.

```
enum DaysOfWeek {  
    sunday,  
    monday,  
    tuesday,  
    wednesday,  
    thursday,  
    friday,  
}
```

```
void main()  
{  
    DaysOfWeek today = DaysOfWeek.sunday;  
  
    if(today == DaysOfWeek.sunday){  
        print('Today is holiday!');  
    } else {  
        print('Its a working day');  
    }  
} // Today is holiday!
```

# Exercise

---

# Exercise - Switch / Case / Enums



## Greet the person.

- **Create enums**
  - First with values morning, afternoon, and evening.
  - Second with values Mr., and Miss.

```
void main()  
{  
  GreetFor greet = GreetFor.afternoon;  
  Titles title = Titles.mr;  
  String name = "Aamir";  
  String message = '';  
  
  // use switch case  
  // print the greeting msg. Example: Good afternoon Mr. Aamir.  
  
  print(message) ;  
}
```



<pre>enum Titles {     mr,     miss }  enum GreetFor {     morning,     afternoon,     evening, }  void main() {     GreetFor greet = GreetFor.afternoon;     Titles title = Titles.mr;     String name = "Aamir";      String message = '';      switch (greet) {         case GreetFor.morning:             message += "Good morning ";             break; </pre>	<pre>         case GreetFor.afternoon:             message += "Good afternoon ";             break;         case GreetFor.evening:             message += "Good evening ";             break;     }      switch(title) {         case Titles.mr:             message += "Mr \$name.";             break;         case Titles.miss:             message += "Miss \$name.";             break;     }      print(message); }  // Good afternoon Mr Aamir.</pre>
---	--

# Exercise

---

Guess the number

---

# Guess the number

- We will be creating a small game
- This app will take a guess from **1 to 5 from user from command line**.
- Application will Generate a **random number** and match it with user input.
- If matched will print **“You win”**, otherwise **“Try again!”**.
- App will also maintain the score by adding 1 on every win.
- User can always type **“exit”** to end the application.

```
import 'dart:io';  
import 'dart:math';  
  
void main() {  
  int score = 0;  
  while (true) {  
    stdout.write("Enter your guess: ");  
    final userInput = stdin.readLineSync();  
    int randomNo = Random().nextInt(5) + 1;  
  
    if (userInput == null) {  
      print('Please enter some input.');      break;  
    } else if (userInput.toLowerCase() == 'exit') {  
      break;  
    } else if (userInput == randomNo.toString()) {  
      score++;  
      print('You win!, your score now is $score');    } else {  
      print('Try again. your input: $userInput, system no. $randomNo');    }  
  }  
}
```

To take input from the console you need to import a library, named dart:io from built-in libraries of Dart.

To generate random number you need to import this built-in library.

The difference between print() and stdout.write() is that print adds a carriage return by default and next statement on the CLI would be on new line whereas stdout.write() show next statement on the same line

# Collections

---

List

---

# Collections - Lists

- It is normally called as an Array in other programming language
- A List is simply an ordered group of objects.

```
void main() {  
    var valArray = ["aamir","ali","fazal"];  
    print( valArray[ 0 ] );    // aamir  
}
```



- Add an element in the list

```
var val_array = ["aamir","ali","fazal"];  
print( val_array[ 0 ] );      // aamir
```

---

```
List a = [];  
a.add("aamir");  
a.add("pinger");  
print(a[1]); // pinger
```

# Collections - Lists with **for loop**

- We can use loops to iterate through lists

```
void main() {  
    var valArray = ["aamir", "ali", "fazal"];  
    for (int i = 0; i < valArray.length; i++) {  
        print(valArray[i]);  
    }  
}
```

```
aamir  
ali  
fazal
```

# Collections - Lists with **for in loop**

- For in loop also helps and easy to use with Lists

```
void main() {  
    var valArray = ["aamir", "ali", "fazal"];  
    for (String element in valArray) {  
        print(element);  
    }  
}
```

```
aamir  
ali  
fazal
```

# Exercise

---

# Exercise Collections - Lists



- Sum the numbers

```
void main() {  
    var valArray = [5, 10, 15, 20, 25];  
    // write the code  
}
```

75

# Exercise Collections - Lists - Solution



- Sum the numbers

```
void main() {  
    var valArray = [5, 10, 15, 20, 25];  
    int total = 0;  
    for (int element in valArray) {  
        total += element;  
    }  
    print(total);  
}
```

75

# Collections - Lists with **.map**

- .map function takes a List as argument and returns a new lazy iterable.
- Lazy iterable means that they will iterate the original list every time when iteration is performed on it but not before.
- This function is very helpful when we need to perform some logic on all List elements and store back as List.

```
void main() {  
    List<int> grossSalaries = [10000, 12000, 15000];  
    List<double> netSalaries = grossSalaries.map((e) => e - (e * 0.10)).toList();  
  
    print(netSalaries);  
}
```

```
[9000.0, 10800.0, 13500.0]
```

# Collections - Lists with **.where**

- **.where()** is useful to filter a List
- **.where()** return a lazy Iterable with all elements that satisfy the given condition.

```
void main() {  
    List<int> grossSalaries = [10000, 12000, 15000];  
    List<int> bigSalaries = grossSalaries.where((e) => e > 10000).toList();  
  
    print(bigSalaries);  
}
```

```
[12000, 15000]
```



# Collections - Lists with **.firstWhere**

- **.firstWhere()** is useful to check if any of the values matches condition.
- **.firstWhere()** takes two parameter, the first one is use function that check for provided condition and returns first value that is found.
- The second parameter is **{int Function()? orElse}**, it will get invoke if first function fails to find any value.
- Do not iterate further if first list element matches the condition.

```
void main() {  
    List<int> grossSalaries = [10000, 12000, 15000];  
    print(grossSalaries.firstWhere((e) => e > 10000, orElse: () => -1));  
    print(grossSalaries.firstWhere((e) => e > 20000, orElse: () => -1));  
}
```

```
12000  
-1
```

Some basic list methods are as follows

- **add()** : The add() function is used to append a specified value to the end of the list and returns a modified list object.
- **addAll()** : The addAll() function is used to append multiple values to the given list object. It accepts multiple comma separated values enclosed within square brackets ([]) and appends it to the list.
- **insert()** : The insert() function is used to insert an element at specified position. It accepts a value and inserts it at the specified index.
- **insertAll()**: The insertAll() function is used to inserts a list of multiple values to a given list at specified position. It accepts index position and a list of multiple comma separated values enclosed within square brackets ([]) and insert the list values beginning from the index specified.

# Collection - List methods

---

- **contains()** : This method will take a value to match with all list elements and will return true if found or else false.
- **indexOf()** : This function will return the FIRST index of value found in the list. **-1** will be returned if no match found.
- **lastIndexOf()** : This function will return the LAST index of value found in the list. **-1** will be returned if no match found.

# Collection - List methods

- **remove()** : The remove() function is used to remove an elements from the list. It removes the first occurrence of a specified element in the list. It returns true if the specified element is removed from the list.
- **removeAt()** : The removeAt() function is used to remove an element from specified index position and returns it.
- **clear():** The clear() function removes all the elements from the list and returns empty list [ ]
- To update the list value we do **list\_name[index] = new\_value;**

# Collection - List Properties

- **first:** It returns the first element case.
- **isEmpty:** It returns true if the collection has no elements.
- **isNotEmpty:** It returns true if the collection has at least one element.
- **length:** It returns length/size of the list, can also be seen as number of elements in a given list.
- **last:** It returns the last element in the list.
- **reversed:** It returns an iterable object containing the lists values in the reverse order.
- **single:** It is used to checks if the list has only one element and returns it.

**A very good article on List methods**

**<https://medium.com/flutter-community/useful-list-methods-in-dart-6e173cac803d>**

# Collection - List - Type annotations

- When we define list with `List list_name = ["val1","val2", 100 ]` or `var list_name = ["val1","val2", 100 ]`, dart do not complaint and takes values of different type, string, integer, boolean or any other type.
- To restrict a list a specific type we can use type annotation **<type\_name>** and by using type annotation dart will throw an error at compile time for type mismatch

```
var valArray = <int>[5, 10, 15, 20, 5, "aamir"];
```

Or

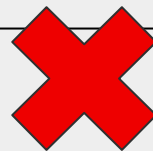
```
List<int> valArray = [5, 10, 15, 20, 5, "aamir"];
```

# Collection - List with final / const



- Final and Const makes variable immutable but they treat List differently.
- When we use **Const** keyword with List, it makes the list immutable but produce exception only at runtime, when using .add method to add a new element.

```
void main() {  
  const List myArray = ['aamir', 'pinger'];  
  myArray.add('irfan');  
  print(myArray); // Uncaught Error: Unsupported operation: add  
}
```





- When we use **final** keyword with List it make the List variable immutable but the values of List can be modified, **which could be quite dangerous!**

```
void main() {  
  final List myArray = ['aamir', 'pinger'];  
  myArray.add('irfan');  
  print(myArray); //[aamir, pinger, irfan]  
}
```




```
void main() {  
  final List myArray = ['aamir', 'pinger'];  
  myArray[0] = 'irfan';  
  print(myArray); // [irfan, pinger]  
}
```



# List.unmodifiable

- Dart offers **List.unmodifiable()** method to create an immutable array.
- By using this method the element of List will be freezed for any modification



```
void main() {  
    final List myArray = List.unmodifiable(['aamir', 'pinger']);  
    myArray.add('irfan');  
    print(myArray); // Uncaught Error: Unsupported operation: add  
}
```

- Only problem with List.unmodifiable is it throw exception at runtime and developer do not get intimation during development.
- To solve warning issue there is a **kt\_dart library** that is widely use to create the all kind of immutable Collections

Set

---

# Collections - Sets



- A set in Dart is an unordered collection of unique items.
- If same value added multiple time, set will ignore the new value.
- For lists we use `[]` and for set we use `{ }`

```
void main() {  
    Set<String> valSet = {"aamir", "ali", "fazal", "aamir"};  
    print( valSet );    // {aamir, ali, fazal}  
}
```

- Add an element in the list

```
var val_set = {"aamir","ali","fazal"};  
print( val_set.elementAt(0) );      // aamir
```

---

```
Set a = {};  
a.add("aamir");  
a.add("pinger");  
print(a[1]); // pinger
```

# Collections - Sets with **for loop**

- We can use loops to iterate through lists

```
void main() {  
    Set<String> uniqueValues = {"aamir", "ali", "fazal", "aamir"};  
    for (int i = 0; i < uniqueValues.length; i++) {  
        print(uniqueValues.elementAt(i));  
    }  
}
```

```
aamir  
ali  
fazal
```

# Collections - Sets with **for in loop**

- For in loop also helps and easy to use with Lists

```
void main() {  
    Set<String> uniqueValues = {"aamir", "ali", "fazal", "aamir"};  
    for (String element in uniqueValues) {  
        print(element);  
    }  
}
```

```
aamir  
ali  
fazal
```

# Exercise

---



# Exercise Collections - Sets



- Sum the numbers

```
void main() {  
    var values = {5, 10, 15, 20, 25};  
    // write the code  
}
```

75

# Exercise Collections - Sets - Solution



- Sum the numbers

```
void main() {  
    var values = {5, 10, 15, 20, 25};  
    int total = 0;  
    for (int element in values) {  
        total += element;  
    }  
    print(total);  
}
```

75

# Collection - Sets methods

- **union()** : **first\_set.union(second\_set)** will merge both sets and returns a new set.
- **intersection()** : **first\_set.intersection(second\_set)** will return a new set with values that are same in both sets. Empty sets if no match found.
- **difference()**: **first\_set.difference(second\_set)** will return a new set with values that are unique in both sets. Empty sets if no match found.
- To update the set value we do not have any direct way but to first remove the element by using **.remove(value)** and add a new value using **.add(value)** method

# Maps

---

# Collections - Maps



- A map is an object that can have keys and values.
- Both keys and values can be any type of object.
- Each key occurs only once, but you can use the same value multiple times.
- If you assign same key once again it will overwrite the previous value of the the same key with the new value

```
void main() {  
  var info = {  
    "firstName": "aamir",  
    "lastName": "pinger",  
    "age": 40,  
  };  
  print(info);  
}
```

```
{firstName: aamir, lastName: pinger, age: 40}
```

```
var val_obj = { "city": "Karachi", "country": "Pakistan"};  
    print( val_obj[ "city" ] );      // Karachi
```

---

```
Map val = { "first": 100, "second": 200};  
    print(val["second"]);           // 200
```

---

```
Map<String, int> val = { "first": 100, "second": 200};  
    print(val["second"]);           // 200
```

# Collections - Maps



```
void main() {  
  var val_obj = {"city": "Karachi", "country": "Pakistan"};  
  print(val_obj.keys); // (city, country)  
}
```

---

```
void main() {  
  var val_obj = {"city": "Karachi", "country": "Pakistan"};  
  print(val_obj.values); // (Karachi, Pakistan)  
}
```

---

```
void main() {  
  var val_obj = {"city": "Karachi", "country": "Pakistan"};  
  print(val_obj.entries); // (MapEntry(city: Karachi), MapEntry(country: Pakistan))  
}
```

# Collections - Maps



```
void main() {  
  var info = Map<String, String>();  
  info['firstName'] = 'Aamir';  
  info['lastName'] = 'Pinger';  
  info['age'] = "40";
```

```
  var items = Map<int, dynamic>();  
  items[1] = 'glass';  
  items[5] = 'cup';  
  items[3] = 1000;
```

```
  print(info);
```

```
  print(items);
```

```
  print(info['full_name']);
```

```
}
```

{firstName: Aamir, lastName: Pinger, age: 40}

{1: glass, 5: cup, 3: 1000}

null



# Collections - Maps



```
void main() {  
  var val_obj = {"city": "Karachi", "country": "Pakistan"};  
  val_obj.forEach((key, value) {  
    print("$key: $value");  
  });  
}
```

city: Karachi  
country: Pakistan


```
void main() {  
  var val_obj = {"city": "Karachi", "country": "Pakistan"};  
  for (var values in val_obj.values) {  
    print(values);  
  }  
}
```

Karachi  
Pakistan

# Collection - Maps with final / const

- Similar to list **final** and **const** treat Maps differently too.
- When we use **Const** keyword with Map, it makes the Map variable and its elements immutable but produce exception only at runtime.

```
void main() {  
  Map myObj = {  
    "name": "aamir",  
  };  
  
  myObj["name"] = "irfan"; // Uncaught Error: Unsupported operation: Cannot modify  
unmodifiable Map  
  
  print(myObj);  
}
```



# Collection - Maps with final / const



- When we use **final** keyword with Map, it makes the Map variable as immutable but values can be modified


```
void main() {  
  final Map myObj = {  
    "name": "aamir",  
  };  
  
  myObj["name"] = "irfan";  
  myObj["gender"] = "male";  
  
  print(myObj); // {name: irfan, gender: male}  
}
```



# Map.unmodifiable

- Dart offers **Map.unmodifiable()** method to create an immutable map.
- Similar to List by using this method the element of Map will be freezed for any modification.

```
void main() {  
  final Map myObj = Map.unmodifiable({  
    "name": "aamir",  
  });  
  
  myObj["name"] = "irfan"; // Uncaught Error: Unsupported operation: Cannot modify  
unmodifiable Map  
  myObj["gender"] = "male";  
  
  print(myObj);  
}
```



# Map.unmodifiable

- Cons of Map.unmodifiable
  - throw exception at runtime and developer do not get intimation during development.
- Solution:
  - **kt\_dart library** that is widely use to create the all kind of immutable Collections

# Exercise

---

# Exercise - Collections

- Declare a map for a school classes with the name **allClasses**.
- Add a **schoolName** key with value **City School**.
- There should be two **shifts** as map keys, **morning** and **afternoon**.
- The **morning shift** as three periods, **maths**, **english**, **science**.
- The **afternoon shift** as two periods, **science**, **arts**.
- The **fee** key to store a school fee with the value **10000**.
- Print school name.
- Print School fee.
- Print 2nd period of morning and 1st period of afternoon shift.



```
City School  
10000  
english  
science
```

# Exercise - Collections - Solution



```
void main(){  
  Map allClasses = {  
    "schoolName": "City School",  
    "morningShift": ["maths", "english", "science"],  
    "afternoonShift": ["science", "arts"],  
    "fee": 10000  
  };  
  
  print(allClasses["schoolName"]);  
  print(allClasses["fee"]);  
  print(allClasses["morningShift"][1]);  
  print(allClasses["afternoonShift"][0]);  
}
```

```
City School  
10000  
english  
science
```



# Exercise - Collections



- Given the following data, create a map and then print all the unique student ids who have scored below 40 in any one subject.

```
{
  "science": {
    "aamir": 40,
    "fazal": 70,
    "ali": 30,
  },
  "maths": {
    "aamir": 60,
    "fazal": 39,
    "ali": 30,
  },
  "english": {
    "aamir": 50,
    "fazal": 70,
    "ali": 60,
  },
}
```

// {ali, fazal}

```
void main() {  
    Map<String, Map<String, int>>  
    studentResults = {  
        "science": {  
            "aamir": 40,  
            "fazal": 70,  
            "ali": 30,  
        },  
        "maths": {  
            "aamir": 60,  
            "fazal": 39,  
            "ali": 30,  
        },  
        "english": {  
            "aamir": 50,  
            "fazal": 70,  
            "ali": 60,  
        },  
    };  
};
```

```
Set<String> names = {};  
    studentResults.forEach((subjects  
scores) {  
        scores.forEach((name, score) {  
            if (score < 40) {  
                names.add(name);  
            }  
        });  
    });  
  
    print(names);  
}  
  
// {ali, fazal}
```

# Collection if, for and spread

---

# Collection if



- When creating a list, you can use a collection if to determine whether an element is included based on some condition.

```
void main() {  
    String teacher = "aamir";  
    var info = [];  
    info.add(teacher);  
    if (teacher == "aamir") {  
        info.add("dart");  
    } else if (teacher == "ali") {  
        info.add("java");  
    }  
    print(info);  
} // [aamir, dart]
```

```
void main() {  
    String teacher = "aamir";  
    var info = [  
        teacher,  
        if (teacher == "aamir")  
            "dart"  
        else if (teacher == "ali")  
            "java"  
    ];  
    print(info);  
} // [aamir, dart]
```

# Collection for



```
void main() {  
  List new_cities = ["karachi", "hyderabad", "lahore"];  
  var cities = ["PESHAWAR"];  
  for (var city in new_cities) {  
    cities.add( city.toUpperCase() );  
  }  
  print(cities);  
} // [PESHAWAR, KARACHI, HYDERABAD, LAHORE]
```

```
void main() {  
  List new_cities = ["karachi", "hyderabad", "lahore"];  
  var cities = [  
    "PESHAWAR",  
    for (var city in new_cities) city.toUpperCase(),  
  ];  
  print(cities);  
} // [PESHAWAR, KARACHI, HYDERABAD, LAHORE]
```

# Spread operator



```
void main() {  
  List new_cities = ["karachi", "hyderabad", "lahore"];  
  var cities = ["peshawar"];  
  for (var city in new_cities) {  
    cities.add( city );  
  }  
  print(cities);  
} // [peshawar, karachi, hyderabad, lahore]
```

```
void main() {  
  Map person = {"name": "aamir", "age": 40};  
  bool isAdult = true;  
  var persons = {  
    ...person,  
    if (person["age"] >= 21) ...{"isAdult": isAdult},  
  };  
  print(persons);  
} // {name: aamir, age: 40, isAdult: true}
```

# By Ref and By value

- Whenever we assign a non-collection variable to another variable, dart creates a new copy of variable value in the memory.

```
void main() {  
  String person1 = "aamir";  
  String person2 = person1;  
  person2 = "ali";  
  
  print("Person 1: $person1");  
  print("Person 2: $person2");  
}
```

```
Person 1: aamir  
Person 2: ali
```

# By Ref and By value

- But when we assign a collection variable to another variable, dart creates a reference that points to the same value in the memory for both variables.

```
void main() {  
  List person1 = ["aamir"];  
  List person2 = person1;  
  person2[0] = "ali";  
  
  print("Person 1: $person1");  
  print("Person 2: $person2");  
}
```

```
Person 1: [ ali ]  
Person 2: [ ali ]
```



# Command line arguments

---

# Command line arguments

- When we create a command line application, we usually need to pass some values to our app, this could be easily done in Dart.

```
void main(List<String> arguments) {  
    print("Arguments: $arguments");  
}
```

```
E:\> dart app.dart "my name is Aamir"  
Arguments: [my name is Aamir]
```

```
E:\> dart app.dart my name is Aamir  
Arguments: [my, name, is, Aamir]
```

# Command line arguments



- When we create a command line application, we usually need to pass some values to our app, this could be easily done in Dart.

```
import 'dart:io';

void main(List<String> arguments) {
  if (arguments.isEmpty) {
    print("Missing arguments. Example: dart <FILE_NAME> <ARGUMENT_LIST>");
    exit(1);
  }
  print("Arguments: $arguments");
}
```

```
E:\> dart app.dart
missing arguments. Example: dart <FILE_NAME> <ARGUMENT_LIST>
```

Read the file

---

# Read the file



- Let's create a msg.txt file and read it using dart script.

My name is Aamir Pinger.  
This is file read example with Dart.

msg.txt

```
import 'dart:io';
```

```
void main(List<String> arguments) {
```

```
    List<String> fileContent = File('./msg.txt').readAsLinesSync();
```

```
    print("File content: $fileContent");
```

```
}
```

app.dart

```
E:\> dart tmp.dart
```

```
File content: [My name is Aamir Pinger., This is file read example with Dart.]
```

# Exercise

---

# Exercise - Reading file



- Using the CSV file we will create a summary of health related activities.
- Link to CSV file <https://gist.github.com/aamirpinger/0cdc25b5ebacc1517255d3b718826676>
- Create a script that takes the file name as command line argument.
- Read the daily activity stats and sum them.
- Show the following summary.
  - No of days for sports activities
  - Average Calories intake with number of days
  - Average sleep hours with number of days

## Expected output

```
Total sports activity for 30 days was 33.05h  
Average daily KCal intake for 31 days was 1835.48 KCal/s  
Average daily Sleep hours for 31 days was 7.35h
```

# Exercise - Reading file - Solution



```
import 'dart:io';

void main(List<String> arguments) {
  if (arguments.isEmpty) {
    print("Missing arguments. Example: dart <FILE_NAME> <ARGUMENT_LIST>");
    exit(1);
  } else if (!arguments.first.endsWith('.csv')) {
    print("Invalid file: File must be CSV format.");
    exit(1);
  }

  String fileName = arguments.first;
  List<String> fileContent = File(fileName).readAsLinesSync();
  fileContent.removeAt(0);
}
```



# Exercise - Reading file - Solution



```
double sportDurationTotal = 0;
int sportDays = 0;
int kcalIntakeTotal = 0;
int kcalDays = 0;
double sleepHoursTotal = 0;
int sleepDays = 0;

for (String line in fileContent) {
    List<dynamic> lineValues = line.split(',');

    // Sports summary calculations
    double todaysSportHours = double.parse(lineValues[1]);
    if (todaysSportHours != 0) {
        sportDurationTotal += todaysSportHours;
        sportDays++;
    }
}
```

# Exercise - Reading file - Solution



```
int todaysKcalIntake = int.parse(lineValues[2]);
if (todaysKcalIntake != 0) {
    kcalIntakeTotal += todaysKcalIntake;
    kcalDays++;
}

double todaysSleepHours = double.parse(lineValues[3]);
if (todaysSleepHours != 0) {
    sleepHoursTotal += todaysSleepHours;
    sleepDays++;
}
}
```

# Exercise - Reading file - Solution



```
print(  
    "Total sports activity for $sportDays days was  
    ${sportDurationTotal.toStringAsFixed(2)}h");  
  
print(  
    "Average daily KCal intake for $kcalDays days was ${ (kcalIntakeTotal /  
kcalDays).toStringAsFixed(2)} KCals");  
  
print(  
    "Average daily Sleep hours for $sleepDays days was ${ (sleepHoursTotal  
/ sleepDays).toStringAsFixed(2)}h");  
  
}
```

# Null Safety

---

- The value **null** represents the intentional absence of any object value.
- This means a variable points to no object.
- Dart by default assign **null** to all variable when **declared** without **initialization**.
- **Declaration** is not to declare "value" to a variable; it's to declare the type of the variable.

```
String name;  
int age;
```

- **Initialization** is the storing a value to a variable at the time of declaration.

```
String name = "aamir";  
int age = 40;
```

# Null Safety



- As **null** is not a value that can be used for any arithmetic operations or even can cause error at runtime if we expect a List or Map but its null.
- Dart after version 2.0 support null safety.
- Dart now will throw an error at compile time if it finds any variable that might not get initialized with any value.

```
void main() {  
    int salary = 10000;  
    int bonus;  
    print("${salary + bonus}");  
}
```

```
E:\> dart app.dart  
app.dart:4:21: Error: Non-nullable variable 'bonus' must be assigned before it can be used.  
    print("${salary + bonus}");  
                  ^^^^^
```

# Null Safety



```
void main() {  
  int salary = 10000;  
  int bonus = 500;  
  print("${salary} + ${bonus}");  
}
```

```
E:\> dart app.dart  
10500
```

```
void main() {  
  int salary = 10000;  
  int bonus;  
  if (salary < 5000) {  
    bonus = 1000;  
  } else {  
    bonus = 500;  
  }  
  print("${salary} + ${bonus}");  
}
```

```
E:\> dart app.dart  
10500
```

- Sometimes we need a variable to accept a null value. For example in case of salary more than 10000 there will be no bonus and app should print a message salary with no bonus.

```
void main() {  
  int salary = 10000;  
  int bonus;  
  if (salary < 5000) {  
    bonus = 1000;  
  }  
  
  if (bonus == null) {  
    print("${salary} with no bonus.");  
  } else {  
    print("${salary + bonus} including bonus.");  
  }  
}
```

```
E:\> dart app.dart
```

```
app.dart:8:21: Error: Non-nullable variable 'bonus' must be assigned before it can be used.
```

```
  print("${salary + bonus}");
```

```
      ^^^^^
```



# Null Safety

- Dart provides a way to tell the compiler that *hey look, this variable might get a null value, so relax!*
- This can be done using **?** sign with variable type,

```
void main() {  
  int salary = 10000;  
  int? bonus;  
  if (salary < 5000) {  
    bonus = 1000;  
  }  
  
  if (bonus == null) {  
    print("${salary} with no bonus.");  
  } else {  
    print("${salary + bonus} including bonus.");  
  }  
}
```

```
E:\> dart app.dart  
10000 with no bonus.
```

# Null Safety



- The beauty of null safety is, if Dart compile senses that the app might not assign any value to nullable variable before using it, Dart will throw error at compile time for this too.
- It's a **sound null safety!**

```
void main() {  
  int salary = 10000;  
  int? bonus;  
  if (salary < 5000) {  
    bonus = 1000;  
  }  
  
  print("${salary + bonus} including bonus.");  
}
```

```
E:\> dart app.dart
```

```
app.dart:4:21: Error: A value of type 'int?' can't be assigned to a variable of type 'num' because 'int?'  
is nullable and 'num' isn't.
```

```
print("${salary + bonus} including bonus.");  
          ^
```

if-null operator

---

# if-null operator

- To deal with the problems like following dart has if-null operator i.e. ??

```
void main() {  
  int salary = 10000;  
  int? bonus;  
  if (salary < 5000) {  
    bonus = 1000;  
  }  
  
  print("${salary + (bonus ?? 0)} including bonus.");  
}
```

```
E:\> dart app.dart  
10000 including bonus
```

```
void main() {  
  int salary = 10000;  
  int? bonus;  
  if (salary < 5000) {  
    bonus = 1000;  
  }  
  
  bonus ??= 0;  
  
  print("${salary + bonus} including bonus.");  
}
```

```
E:\> dart app.dart  
10000 including bonus
```

null safety on Collections

---

# null safety on Collections

- There will be lot of situation where we need to store a null value into the collections.
- For example you records a daily sporting hours in a List but on sunday user do not adds any sports hours. So technically for that you will place a null value to the List element.

```
void main() {  
  List<String> dayNames = [  
    'Monday',  
    'Tuesday',  
    'Wednesday',  
    'Thursday',  
    'Friday',  
    'Saturday',  
    'Sunday'  
  ];  
  List<double> dailySportsHours = [1, 1.5, 1.25, 1.3, 2, 1.5, 2];  
  for (int i = 0; i < dailySportsHours.length; i++) {  
    print('${dayNames[i]}: ${dailySportsHours[i]}');  
  }  
}
```

```
E:\> dart app.dart  
Monday: 1.0  
Tuesday: 1.5  
Wednesday: 1.25  
Thursday: 1.3  
Friday: 2.0  
Saturday: 1.5  
Sunday: 2.0
```

**List without any null value**



# null safety on Collections

- But when you add a null value in List<double>, the dart compiler will throw an error.

```
void main() {  
  List<String> dayNames = [  
    'Monday',  
    'Tuesday',  
    'Wednesday',  
    'Thursday',  
    'Friday',  
    'Saturday',  
    'Sunday'  
  ];  
  List<double> dailySportsHours = [1, 1.5, 1.25, 1.3, 2, 1.5, null];  
  for (int i = 0; i < dailySportsHours.length; i++) {  
    print('${dayNames[i]}: ${dailySportsHours[i]}');  
  }  
}
```

List WITH a null value



```
E:\> dart app.dart
```

```
app.dart:11:63: Error: The value 'null' can't be assigned to a variable of type 'double' because 'double' is not nullable.
```

```
List<double> dailySportsHours = [1, 1.5, 1.25, 1.3, 2, 1.5, null];
```

# null safety on Collections

- But when you add a null value in List<double>, the dart compiler will throw an error.

```
void main() {  
  List<String> dayNames = [  
    'Monday',  
    'Tuesday',  
    'Wednesday',  
    'Thursday',  
    'Friday',  
    'Saturday',  
    'Sunday'  
  ];  
}
```

```
E:\> dart app.dart  
Monday: 1.0  
Tuesday: 1.5  
Wednesday: 1.25  
Thursday: 1.3  
Friday: 2.0  
Saturday: 1.5  
Sunday: null
```

List WITH a null value

```
List<double?> dailySportsHours = [1, 1.5, 1.25, 1.3, 2, 1.5, null];  
for (int i = 0; i < dailySportsHours.length; i++) {  
  print('${dayNames[i]}: ${dailySportsHours[i]}');  
}
```



# null safety on Collections

- Null safety could be further used on the functions of any object.
- For example you want to round all the numbers and you use `.round()` method, but because list could have null, dart compiler will throw an Error.

```
void main() {  
    List<String> dayNames = [ 'Monday', 'Tuesday', 'Wednesday',  
    'Thursday', 'Friday', 'Saturday', 'Sunday' ];  
    List<double?> dailySportsHours = [1, 1.5, 1.25, 1.3, 2, 1.5,  
    null];  
    for (int i = 0; i < dailySportsHours.length; i++) {  
        print('${dayNames[i]}: ${dailySportsHours[i].round()}');  
    }  
}
```

```
E:\> dart app.dart
```

```
app.dart:5:50: Error: Method 'round' cannot be called on 'double?' because it is potentially null.  
Try calling using ?. instead.   print('${dayNames[i]}: ${dailySportsHours[i].round()}');
```

# null safety on Collections



- To avoid this we use ? before any methods i.e. `obj_name?.round()`

```
void main() {  
  List<String> dayNames = [  
    'Monday',  
    'Tuesday',  
    'Wednesday',  
    'Thursday',  
    'Friday',  
    'Saturday',  
    'Sunday'  
  ];  
  
  List<double>? dailySportsHours = [1, 1.5, 1.25, 1.3, 2, 1.5, null];  
  for (int i = 0; i < dailySportsHours.length; i++) {  
    print('${dayNames[i]}: ${dailySportsHours[i]}.round()');  
  }  
}
```

```
E:\> dart app.dart  
Monday: 1.0  
Tuesday: 1.5  
Wednesday: 1.25  
Thursday: 1.3  
Friday: 2.0  
Saturday: 1.5  
Sunday: null
```

List WITH a null value

# Functions

---

- Functions are the building blocks of readable, maintainable, and reusable code.
- A function is a set of statements to perform a specific task.
- Functions organize the program into logical blocks of code.
- Once defined, functions may be called to access code.
- This makes the code reusable.
- For example you have a piece of code or some complex logic that you might
  - Less code if you want to use the same line of code multiple times,
  - Makes it easy to maintain. You change at once place and change reflects everywhere that function is called.
  - Separate it from the long code for better readability.

# Function syntax

---

# Function syntax



Here we write the type of function's return value or void incase function does not returns any value

Function name by which we will call it for example **any\_function\_name()**

The comma separated list of arguments which we will pass when calling a function, these values then can be used in function's body.

```
void function_name(arguments) {
```

```
// function body
```

```
}
```

Function body will contain the piece of code to be executed when invoking a function.

# Functions



- Simple function calling

```
void main() {  
  printName();  
}
```

```
void printName() {  
  print("The name is Aamir.");  
}
```

```
E:\> dart app.dart  
The name is Aamir.
```

# Functions



- If any function has only one statement, we can add `=>` and remove `{ }` and even return statement.

```
void main() => printName();
```

```
void printName() => print("The name is Aamir.");
```

```
E:\> dart app.dart  
The name is Aamir.
```



# Functions



- Passing an argument to function.

```
void main() {  
    printName("Aamir");  
}  
  
void printName(String name) {  
    print("The name is $name.");  
}
```

```
E:\> dart app.dart  
The name is Aamir.
```

# Functions



- Return a value from a function.

```
void main() {  
    String message = getMessage("Aamir");  
    print(message);  
}  
  
String getMessage(String name) {  
    return "The name is $name.";  
}
```

```
E:\> dart app.dart  
The name is Aamir.
```

# Functions



- Return a value from a function.

```
void main() {  
    printName("Aamir", 40);  
}
```

```
void printName(String name, int age) {  
    print("My name is $name. I am $age years old.");  
}
```

```
E:\> dart app.dart  
My name is Aamir. I am 40 years old.
```

# Functions



- Making a function parameter **nullable**.

```
void main() {  
  printName("Aamir");  
}
```

```
void printName(String name, [int? age]) {  
  print("My name is $name. I am $age years old.");  
}
```

```
E:\> dart app.dart  
My name is Aamir. I am null years old.
```

# Functions



- Assigning default value in case of no value passed

```
void main() {  
    printName(null, "Aamir");  
    printName(40);  
}  
  
void printName([int? age, String name = 'unknown']) {  
    print("My name is $name.");  
    if (age != null) {  
        print("I am $age years old.");  
    }  
}
```

```
E:\> dart app.dart  
My name is Aamir.  
My name is unknown.  
I am 40 years old.
```

# Exercise

---

# Exercise - break the following function into functions



```
void main() {  
  int yourNumber = Random().nextInt(6) + 1;  
  print("Your number $yourNumber");  
  int systemNumber = Random().nextInt(6) + 1;  
  print("System number $systemNumber");  
  if (yourNumber == systemNumber) {  
    print("its a tie!");  
  } else if (yourNumber > systemNumber) {  
    print("You win!");  
  } else {  
    print("You lost!");  
  }  
}
```

```
E:\> dart app.dart  
Your number 6  
System number 5  
You win!
```

# Exercise - Solution



```
import 'dart:math';

void main() {
  int yourNumber = rollTheDice('Your');
  int systemNumber = rollTheDice('System');
  showTheResult(yourNumber, systemNumber);
}
```

```
void showTheResult(int yourNumber, int
systemNumber) {
  if (yourNumber == systemNumber) {
    print("its a tie!");
  } else if (yourNumber > systemNumber) {
    print("You win!");
  } else {
    print("You lost!");
  }
}
```

```
int rollTheDice(String title) {
  int num = Random().nextInt(6) + 1;
  print("$title number $num");
  return num;
}
```

```
E:\> dart app.dart
Your Number number 6
System Number number 2
You win!
```



# Named parameters

---

# Named Parameters

- It is mandatory to pass all function arguments in a correct order as described in the function parameters list

```
void printInfo(String name, int age) {  
    ...  
}
```

```
printInfo("aamir", 40);
```



```
printInfo(40, "aamir");
```



# Named Parameters

- Dart also provide us option to pass named parameters that helps in better readability of the code and allows passing parameters in any order.

```
void main() {  
    printInfo(age: 40, name: "aamir");  
}
```

```
void printInfo({required String name, required int age}) {  
    print("$name $age");  
}
```

Required / nullable / Default value

---

# Required / nullable / Default value

---

- When using named parameter in function's argument list we need tell compiler following things
  - Is parameter is mandatory?
  - Can parameter be null?
  - Any default value if no value is being passed to the function.

# Is parameter is mandatory?

- This could be done using **required** keyword before parameter type

```
void main() {  
    printInfo(age: 40, name: "Aamir");  
}
```

```
void printInfo({required String name, required int age}) {  
    print("$name $age");  
}
```

```
E:\> dart app.dart  
Aamir 40
```

# Can parameter be null?

- This could be done using **nullable ? operator** after a parameter type.

```
void main() {  
    printInfo( name: "Aamir" );  
}
```

```
void printInfo( {String? name, int? age} ) {  
    print("$name $age");  
}
```

```
E:\> dart app.dart  
Aamir null
```

# Default value for function parameters

- This could be done by assigning a default value at function level.

```
void main() {  
    printInfo(age: 40, name: "Aamir");  
    printInfo();  
}
```

```
void printInfo({String name = "unknown", int age = 0}) {  
    print("$name $age");  
}
```

```
E:\> dart app.dart  
Aamir 40  
unknown 0
```



# Exercise

---

# Exercise - Functions



- Find the average students age of different programs. Must use functions!

```
[  
  { "name": 'BBA Morning', studentAgeList": [20, 21, 22, 20, 23, 19] },  
  { "name": 'MBA Morning', "studentAgeList": [23, 23, 24, 23, 22.5] },  
  {"name": 'MBA Evening', "studentAgeList": [] }  
];
```

**Output should be:**

The average age in program BBA Morning is 20.83  
The average age in program MBA Morning is 23.1  
The average age in program MBA Evening is 0.0

# Exercise - Functions - Solution



```
void main() {  
  List<Map> studentRecords = [  
    { "name": 'BBA Morning', "studentAgeList": [20, 21, 22, 20, 23, 19] },  
    { "name": 'MBA Morning', "studentAgeList": [23, 23, 24, 23, 22.5] },  
    { "name": 'MBA Evening', "studentAgeList": [] }  
  ];  
  getAllAverages(studentRecords);  
}  
  
void getAllAverages(List<Map> studentRecords) {  
  studentRecords.forEach((element) {  
    print(  
      "The average age in program ${element['name']} is  
      ${getAverage(element['studentAgeList'] ?? [])}");  
    });  
}
```

# Exercise - Functions - Solution



```
double getAverage(List<dynamic> ageList) {  
    if (ageList.length == 0) {  
        return 0;  
    }  
  
    double totalAge = 0;  
    ageList.forEach((element) {  
        totalAge += element;  
    });  
  
    return double.parse((totalAge / ageList.length).toStringAsFixed(2));  
}
```

# Lexical scope

---

# Lexical scope

- Dart is a lexically scoped language
- The scope of variables is determined statically, simply by the layout of the code.
- You can “**follow the curly braces outwards**” to see if a variable is in scope.

## Example

```
String topLevel = "This is top level variable";

void main() {
  String insideMain = "This variable is define in main function";
  ;

  void myFunction() {
    String insideFunction = "This variable is define in inside myFunction";
    ;
  }
}
```

# Lexical scope - Example (cont...)



```
void nestedFunction() {  
    String insideNestedFunction =  
        "This variable is define in inside nestedFunction";  
    ;  
  
    print(topLevel);  
    print(insideMain);  
    print(insideFunction);  
    print(insideNestedFunction);  
}  
  
nestedFunction();  
}  
  
myFunction();  
}
```

This is top level variable  
This variable is define in main function  
This variable is define in inside myFunction  
This variable is define in inside nestedFunction

# Anonymous functions

---



# Anonymous functions

- Most functions are named, such as `main()` or `printElement()`.
- You can also create a nameless function called an anonymous function, or sometimes a lambda or closure.
- You might assign an anonymous function to a variable so that, for example, you can add or remove it from a collection.
- An anonymous function looks similar to a named function— zero or more parameters, separated by commas and optional type annotations, between parentheses.

```
void main() {  
    final printMsg = (String name) {  
        print('Hello $name!');  
    };  
  
    printMsg('Aamir');  
}
```

Hello Aamir

# Functions as first-class objects

---

# Functions as first-class objects

- You can pass a function as a parameter to another function.
- One of the example is `forEach` method of `Collection`, that takes a function as input.

```
void main() {  
  List names = ["Aamir", "Pinger"];  
  
  names.forEach((element) {  
    print(element);  
  });  
}
```

```
Aamir  
Pinger
```

# Functions as first-class objects



- You can also create a custom function that takes a function as an argument
- The `String Function(String) getMessage` say from left to right.
  - Returns a **String** value.
  - Type is **Function** and it takes **String** parameter.
  - Function name is **getMessage**.

```
void main() {  
  final getMsg = (String name) => 'Hello $name';  
  
  final printMsg = (String Function(String) getMessage, String name) {  
    String msg = getMessage(name);  
    print(msg);  
  };  
  
  printMsg(getMsg, 'Aamir');  
  printMsg(getMsg, 'Pinger');  
}
```

Hello Aamir  
Hello Pinger

# Function as Types

---

# Function as Types

- The following printMsg function argument list look a bit messy.
- We can define a type as function and use it as we use String, int, List etc.

```
void main() {  
  final getMsg = (String name) => 'Hello $name';  
  
  final printMsg = (String Function(String) getMessage, String name) {  
    String msg = getMessage(name);  
    print(msg);  
  };  
  
  printMsg(getMsg, 'Aamir');  
  printMsg(getMsg, 'Pinger');  
}
```

```
Hello Aamir  
Hello Pinger
```

# Function as Types

- The following printMsg function argument list look a bit messy.
- We can define a type as function and use it as we use String, int, List etc.

```
typedef Message = String Function(String);
```

```
void main() {
```

```
    final getMsg = (String name) => 'Hello $name';
```

```
    final printMsg = (Message getMessage, String name) {
```

```
        String msg = getMessage(name);
```

```
        print(msg);
```

```
    };
```

```
    printMsg(getMsg, 'Aamir');
```

```
    printMsg(getMsg, 'Pinger');
```

```
}
```

```
Hello Aamir  
Hello Pinger
```

# Classes

---



- Think of class as any object that have some properties and functional abilities.
- In the real world, you are an object of a class called an human being
- Every human share some similar abilities and some unique properties or functional abilities unique to them.
- Every human have bones, skin etc but height, weight, skin color could be different to others.

- Another example could be your car or bike are also an object from vehicle class.
- The Car or Bike, share some same characteristics or functionalities along with additional distinguished characteristics.
- For example they take you from one place to another place, both have horn but car have 4 wheels and bike have usually 2 wheels that make them different.

- Dart is an object-oriented language with classes and mixin-based inheritance.
- When you think of object oriented programming, you use objects to represent things same like the real world.
- Those object can have different characteristics, properties and ability to perform some functionality (object methods).

- They may extends some of the characteristics and functional abilities from parent object.
- These are very helpful to confine the related properties and functionality into a wrapper i.e. a class.
- You reuse the same wrapper by creating instances and every instance then hold their own properties.
- For example you create instances for toyota, honda, and suzuki from Car Class

# Classes - Instances

---

# Classes example



```
class Car {  
  String? color;  
  int wheels = 4;  
}
```

This is the **class** that have **2 properties**, **color** and **wheels**

```
void main() {  
  Car myCar = Car();  
  print(myCar.wheels);  
}
```

**myCar** is the **instance** of the **Car Class**

4

This is how you **access instance property**

# Classes example



```
class Car {  
  String? color;  
  int wheels = 4;  
}  
  
void main() {  
  Car myCar = Car();  
  myCar.color = 'White';  
  print(myCar.color);  
  print(myCar.wheels);  
}
```

White  
4

```
class Car {  
  String? color;  
  int wheels = 4;  
}  
  
void main() {  
  Car myCar = Car();  
  myCar.color = 'White';  
  Car yourCar = Car();  
  yourCar.color = 'Red';  
  print(myCar.color);  
  print(myCar.wheels);  
  print(yourCar.color);  
  print(yourCar.wheels);  
}
```

White  
4  
Red  
4

# Classes - Methods

---



# Classes - methods



```
class Car {  
  String? color;  
  int wheels = 4;  
  int mileage = 0;  
  
  int getMileage() {  
    return mileage;  
  }  
  
  bool increaseMileage(int distance) {  
    mileage += distance;  
    return true;  
  }  
}
```

# Classes - methods



```
void main() {  
    Car myCar = Car();  
    myCar.color = 'White';  
    print('The car current mileage is ${myCar.getMileage()}');  
    myCar.increaseMileage(50);  
    print('The car current mileage is ${myCar.getMileage()}');  
    myCar.increaseMileage(100);  
    print('The car current mileage is ${myCar.getMileage()}');  
}
```

The car current mileage is 0

The car current mileage is 50

The car current mileage is 150

# Classes - Constructors

---

- This of the previous example where initial mileage of car instance is Zero, but think if that car is second hand and has some mileage already on instance initialization.
- This type of initial value assignment can be done using class constructor.
- Class constructor is a simple function with the same name as class name that we automatically invoke on instance creation.
- Using this function we can set the initial values, for our example, initial car mileage.

# Classes - constructor



```
class Car {  
  Car(int mileage) {  
    mileage = mileage;  
  }  
  
  String? color;  
  int wheels = 4;  
  int mileage = 0;  
  
  int getMileage() {  
    return mileage;  
  }  
  
  bool increaseMileage(int distance) {  
    mileage += distance;  
    return true;  
  }  
}
```

# Classes - constructor



```
void main() {  
  Car myCar = Car(10);  
  myCar.color = 'White';  
  print('The car current mileage is ${myCar.getMileage()}');  
  myCar.increaseMileage(50);  
  print('The car current mileage is ${myCar.getMileage()}');  
  myCar.increaseMileage(100);  
  print('The car current mileage is ${myCar.getMileage()}');  
}
```

The car current mileage is 0  
The car current mileage is 50  
The car current mileage is 150

**Opps, it is still showing 0 on first mileage print.**

# Classes - This Keyword

---

# Classes - this keyword



```
class Car {  
  Car(int mileage) {  
    this.mileage = mileage;  
  }  
  
  String? color;  
  int wheels = 4;  
  int mileage = 0;  
  
  int getMileage() {  
    return mileage;  
  }  
  
  bool increaseMileage(int distance) {  
    mileage += distance;  
    return true;  
  }  
}
```

**this keyword refers to the instance variable.**



# Classes - constructor



```
void main() {  
    Car myCar = Car(10);  
    myCar.color = 'White';  
    print('The car current mileage is ${myCar.getMileage()}');  
    myCar.increaseMileage(50);  
    print('The car current mileage is ${myCar.getMileage()}');  
    myCar.increaseMileage(100);  
    print('The car current mileage is ${myCar.getMileage()}');  
}
```

```
The car current mileage is 10  
The car current mileage is 60  
The car current mileage is 150
```

# Classes - Late initializer

---

# Classes - late initializer

```
class Car {  
  Car(int mileage) {  
    this.mileage = mileage;  
  }  
}
```

```
String? color;  
int wheels = 4;  
int mileage = 0;
```

```
int getMileage() {  
  return mileage;  
}
```

```
bool increaseMileage(int distance) {  
  mileage += distance;  
  return true;  
}  
}
```

**This color variable is nullable, means it can have null value, but what if we want non-nullable value**

# Classes - late initializer



```
void main() {  
  Car myCar = Car(10);  
  myCar.color = 'red';  
  print('The car color is ${myCar.color}');  
}
```

The car color is null

# Classes - late initializer



```
class Car {  
  Car(int mileage) {  
    this.mileage = mileage;  
  }  
  
  String color;  
  int wheels = 4;  
  int mileage = 0;  
  
  int getMileage() {  
    return mileage;  
  }  
  
  bool increaseMileage(int distance) {  
    mileage += distance;  
    return true;  
  }  
}
```

When you remove ?, compiler will through an error

```
void main() {  
  Car myCar = Car(10);  
  myCar.color = 'red';  
  print('The car color is ${myCar.color}');  
}
```

Error: Field 'color' should be initialized because its type 'String' doesn't allow null.

```
String color;  
  ^^^^^
```

# Classes - late initializer



```
class Car {  
  Car(int mileage) {  
    this.mileage = mileage;  
  }  
  
  late String color;  
  int wheels = 4;  
  int mileage = 0;  
  
  int getMileage() {  
    return mileage;  
  }  
  
  bool increaseMileage(int distance) {  
    mileage += distance;  
    return true;  
  }  
}
```

Using late keyword tells compiler that this variable will be initialized later but before accessing it anywhere.

# Classes - late initializer



```
void main() {  
  Car myCar = Car(10);  
  myCar.color = 'red';  
  print('The car color is ${myCar.color}');  
}
```

The car color is red



# Class - Constructor - initializer list

---

# Classes - Constructor - initializer list



```
class Car {  
  Car(int mileage) : mileage = mileage;
```

```
  String? color;  
  int wheels = 4;  
  int mileage = 0;
```

**this is called initializer list**

```
  int getMileage() {  
    return mileage;  
  }
```

```
  bool increaseMileage(int distance) {  
    mileage += distance;  
    return true;  
  }  
}
```

# Classes - Constructor - initializer list



```
void main() {  
    Car myCar = Car(10);  
    myCar.color = 'White';  
    print('The car current mileage is ${myCar.getMileage()}');  
    myCar.increaseMileage(50);  
    print('The car current mileage is ${myCar.getMileage()}');  
    myCar.increaseMileage(100);  
    print('The car current mileage is ${myCar.getMileage()}');  
}
```

```
The car current mileage is 10  
The car current mileage is 50  
The car current mileage is 150
```

# Classes - Constructor - initializer list



```
class Car {  
    Car(String color, int mileage)  
        : mileage = mileage,  
          color = color;  
  
    String? color;  
    int wheels = 4;  
    int mileage = 0;  
  
    int getMileage() {  
        return mileage;  
    }  
  
    bool increaseMileage(int distance) {  
        mileage += distance;  
        return true;  
    }  
}
```

# Classes - Constructor - initializer list



```
void main() {  
    Car myCar = Car('red', 10);  
    print('The car current mileage is ${myCar.getMileage()}');  
    print('The car color is ${myCar.color}');  
}
```

The car current mileage is 10  
The car color is red

Constructor - initializer list with default value

---

# Classes - Constructor - initializer list



```
class Car {  
  Car({required String color, int mileage = 0})  
    : mileage = mileage,  
      color = color;  
  
  String? color;  
  int wheels = 4;  
  int mileage = 0;  
  
  int getMileage() {  
    return mileage;  
  }  
  
  bool increaseMileage(int distance) {  
    mileage += distance;  
    return true;  
  }  
}
```

# Classes - Constructor - initializer list



```
void main() {  
    Car myCar = Car(color: 'red', mileage: 10);  
    print('The car current mileage is ${myCar.getMileage()}');  
    print('The car color is ${myCar.color}');  
}
```

The car current mileage is 10  
The car color is red



# Constructor - initializer list - shorthand

---

# Classes - Constructor - initializer list - shorthand



```
class Car {  
  Car({required this.color, this.mileage = 0});  
  
  String? color;  
  int wheels = 4;  
  int mileage;  
  
  int getMileage() {  
    return mileage;  
  }  
  
  bool increaseMileage(int distance) {  
    mileage += distance;  
    return true;  
  }  
}
```

# Classes - Constructor - initializer list



```
void main() {  
    Car myCar = Car(color: 'red', mileage: 10);  
    print('The car current mileage is ${myCar.getMileage()}');  
    print('The car color is ${myCar.color}');  
}
```

The car current mileage is 10  
The car color is red

# Exercise

---

- Create a class for your bank account.
- Required properties are
  - Current Balance
  - Account title
  - Transaction list (deposit should be positive numbers, and withdrawal amounts should be in negatives)
- Required methods are
  - Make a deposit
  - Withdraw from bank (if balance is available)
  - Print the transactions (if no transactions then it should **print 'No Transactions found.'**)
  - Get the account title
  - Get Current Balance
- You need to decide yourself on where to use final, late or with what variables should create a constructor initializer list.

# Class - Exercise - Expected output



The title of my account is Aamir

The initial bank balance is 0

No transaction found.

The initial bank balance is 140

The transactions are:

50

-10

100

End of transactions.

# Class - Exercise - Solution



```
class Bank {  
  Bank({required this.title, this.balance = 0});  
  
  final String title;  
  int balance;  
  List<int> transactions = [];  
  
  bool deposit(int amount) {  
    balance += amount;  
    transactions.add(amount);  
    return true;  
  }  
  
  String getTitle() {  
    return title;  
  }  
}
```

# Class - Exercise - Solution



```
bool withdraw(int amount) {
    if (balance > amount) {
        balance -= amount;
        transactions.add(-amount);
        return true;
    }
    return false;
}

void printTransactions() {
    if (transactions.isEmpty) {
        print('No transaction found.');
```

```
    } else {
        print('The transactions are:');
        transactions.forEach((element) {
            print("$element");
        });
        print('End of transactions.');
```

```
    }
}
```



# Class - Exercise - Solution



```
int getBalance() {  
    return balance;  
}  
  
void main() {  
    final myAccount = Bank(title: "Aamir");  
    print('The title of my account is ${myAccount.getTitle()}');  
    print('The initial bank balance is ${myAccount.getBalance()}');  
    myAccount.printTransactions();  
    myAccount.withdraw(10);  
    myAccount.deposit(50);  
    myAccount.withdraw(10);  
    myAccount.deposit(100);  
    print('The current bank balance is ${myAccount.getBalance()}');  
    myAccount.printTransactions();  
}
```

# Named Constructor

---

# Named Constructor

- You may have a need of creating multiple constructor for a class.
- Using multiple constructor creates ease and also gives better readability of the code.

```
class Car {  
  Car({required this.color, required this.wheels, required this.doors});  
  
  Car.twoDoors(String color) {  
    this.color = color;  
    doors = 2;  
    wheels = 4;  
  }  
  
  Car.fourDoor(String color) {  
    this.color = color;  
    doors = 4;  
    wheels = 4;  
  }  
  
  String? color;  
  int wheels;  
  int doors;  
}
```

# Named Constructor



```
void main() {  
  final myCar = Car.twoDoors('red');  
  print('The car color is ${myCar.color}');  
  print('The car is ${myCar.wheels} wheeler.');
```

```
  print('The car is ${myCar.doors} door.');
```

```
  final myCar2 = Car.fourDoor('blue');  
  print('The car color is ${myCar2.color}');
```

```
  print('The car is ${myCar2.wheels} wheeler.');
```

```
  print('The car is ${myCar2.doors} door.');
```

```
  final myCar3 = Car(color: 'white', wheels: 6, doors: 5);  
  print('The car color is ${myCar3.color}');
```

```
  print('The car is ${myCar3.wheels} wheeler.');
```

```
  print('The car is ${myCar3.doors} door.');
```

```
}
```

The car color is red  
The car is 4 wheeler.  
The car is 2 door.  
The car color is blue  
The car is 4 wheeler.  
The car is 4 door.  
The car color is white  
The car is 6 wheeler.  
The car is 5 door.

# Getters and Setters

---

# Getters and Setters

- Getters and setters are special class methods that is used to initialize and retrieve the values of class fields respectively.
- The setter method is used to set or initialize respective class field,
- The getter method is used to retrieve respective class field value.
- All classes have default getter and setter method associated with it.
- However, you are free to override the default ones by implementing the getter and setter method explicitly.
- It is helpful when you wanted to manipulate the value before getting assigned to field or manipulate the field before retrieving it.

# Getters and Setters



```
class Person {  
  Person({this.firstName = '', this.LastName = ''});  
  
  String firstName;  
  String LastName;  
}  
  
void main() {  
  Person me = Person(firstName: 'Aamir', LastName: 'Pinger');  
  
  print('${me.firstName} ${me.LastName}');  
}
```

- Let say you want to get the full name but you have only first name and the last name field in the class.

# Getters and Setters



- One way could be to add a class method like following.

```
class Person {  
  Person({this.firstName = '', this.LastName = ''});  
  
  String firstName;  
  String LastName;  
  
  void setFullName(String fullName) {  
    final splittedName = fullName.trim().split(' ');  
    this.firstName = splittedName.removeAt(0);  
    this.LastName = splittedName.join(' ');  
  }  
  
  String getFullName() => '${firstName} ${LastName}';  
}
```

```
void main() {  
  Person me = Person(  
    firstName: 'Aamir',  
    LastName: 'Pinger'  
  );  
  
  print(me.getFullName());  
  me.setFullName('Asif Ali');  
  print(me.firstName);  
  print(me.LastName);  
  print(me.getFullName());  
}
```

Aamir Pinger  
Asif  
Ali  
Asif Ali



# Getters and Setters



- Another clean way of doing the same to have a setter and getter function.

```
class Person {  
  Person({this.firstName = '', this.LastName = ''});  
  
  String firstName;  
  String LastName;  
  
  void set fullName(String fullName) {  
    final splittedName = fullName.trim().split(' ');  
    this.firstName = splittedName.removeAt(0);  
    this.LastName = splittedName.join(' ');  
  }  
  
  String get fullName => '${firstName} ${LastName}';  
}
```

```
void main() {  
  Person me = Person(  
    firstName: 'Aamir',  
    LastName: 'Pinger'  
  );  
  
  print(me.fullName);  
  me.fullName = 'Asif Ali';  
  print(me.firstName);  
  print(me.LastName);  
  print(me.fullName);  
}
```

Aamir Pinger  
Asif  
Ali  
Asif Ali

# Exercise

---

# Exercise - Getters and Setters

- Create a BankAccount class.
- Class will have title and balance property.
- Title should be mandatory field.
- Deposit and withdraw method for Pak rupees amount.
- Create Getter / Setter:
  - Deposit in dollars and **setter** should convert it into Pak rupees @ 160 rs before adding to balance.
  - Withdraw in dollars and **setter** should convert it to pak rupees before deducting it from balance.
  - **Getter** to return the Balance after converting it to Dollars @ 160.

# Exercise - Getters and Setters - Solution



```
class Bank {  
  Bank({required this.title, this.balance = 0});  
  
  final String title;  
  int balance;  
  int dollarRate = 160;  
  
  bool deposit(int amount) {  
    balance += amount;  
    return true;  
  }  
  
  bool withdraw(int amount) {  
    if (balance > amount) {  
      balance -= amount;  
      return true;  
    }  
    return false;  
  }  
}
```

# Exercise - Getters and Setters - Solution



```
void set dollarWithdraw(int dollarAmount) {  
    final pakRs = dollarAmount * dollarRate;  
    if (balance > pakRs) {  
        balance -= pakRs;  
    } else {  
        print('Insufficient funds.');    }  
}  
  
double get balanceInDollar => balance / dollarRate;
```

# Exercise - Getters and Setters - Solution



```
void set dollarDeposit(int dollarAmount) {  
    balance += dollarAmount * dollarRate;  
}  
  
}  
  
void main() {  
    final myAccount = Bank(title: "Aamir");  
    print('The title of my account is ${myAccount.title}');  
    print('The initial bank balance is ${myAccount.balance}');  
    myAccount.dollarWithdraw = 10;  
    myAccount.dollarDeposit = 50;  
    myAccount.dollarWithdraw = 10;  
    myAccount.dollarDeposit = 100;  
    print('The bank balance in PKR ${myAccount.balance}');  
    print('The bank balance in dollar is ${myAccount.balanceInDollar}');  
}
```

The title of my account is Aamir  
The initial bank balance is 0  
Insufficient funds.  
The bank balance in PKR 22400  
The bank balance in dollar is 140.0

# Static variables and methods

---

- Till now to access every method or variable we've first initialize the class instance and then used them.
- As there could be multiple class instances of a single class, we can assign different values for every instance variables.
- It is quite right to say class variables actually belongs to the class instances.
- Static method are different. **They do not belong to instance but to the class itself**
- To use static method or variable, we **do not have to create class instance**, just **className.variableName**
- When they are belong to class that means that will not have multiple values as class is always one.



# Static variables and methods

- They are useful when we need to define enums or want some serialization of the data. For example.

```
class BankNames {  
    static const String Meezan = 'Meezan Bank';  
    static const String UBL = 'United Bank';  
    static const String HBL = 'Habib Bank';  
  
    static void greetMessage(String name, String bankName) {  
        print('Welcome ${name} at the ${bankName}.');  
    }  
}  
  
void main() {  
    print(BankNames.Meezan);  
    print(BankNames.HBL);  
    BankNames.greetMessage("Aamir", BankNames.UBL);  
}
```

Meezan Bank  
Habib Bank  
Welcome Aamir at the United Bank

# Private members

---

# Classes - Private members

- When you create class methods and variable, by default we can access all of them by default by **className.methodOrPropertyName**
- What if we want to restrict direct access to some of them, what if you wanted to store a password to class property?
- We just did the bank class example, where we accessed or even can change the balance property directly, but practicality we shouldn't allow to directly access or manipulate it.

# Private members



```
class Bank {  
  Bank({required this.title, this.balance = 0});  
  
  final String title;  
  int balance;  
  
  void deposit(int amount) {  
    balance += amount;  
  }  
  
  void withdraw(int amount) {  
    if (balance > amount) {  
      balance -= amount;  
    }  
  }  
}  
  
void main() {  
  final myAccount = Bank(title: "Aamir");  
  print('The initial bank balance is ${myAccount.balance}');  
  myAccount.deposit(100);  
  myAccount.balance = 50;  
  print('The current bank balance is ${myAccount.balance}');  
}
```

The initial bank balance is 0  
The current bank balance is 50

- To solve this problem, Dart supports Private members as other object oriented languages.
- In dart you can make any property or method private by adding **\_ (underscore)** before its name. For example **\_balance**
- This way we can tell dart that this variable will not be accessible from outside the class scope and max that same dart source code file where this class is written.
- If the class is written in index.dart and class has some private members, then anyone from index.dart file can also access private member of that class
- When you import one dart file to another, your private members will not be accessible from outside of the class and we need to use some class functions or getter / setters to access them.

# Private members



```
class Bank {  
  Bank({required this.title});  
  
  final String title;  
  int _balance = 0;  
  
  void deposit(int amount) { _balance += amount; }  
  
  void withdraw(int amount) {  
    if (_balance > amount) { _balance -= amount; }  
  }  
  
  int get currentBalance => _balance;  
}
```

bankClass.dart

```
import 'bankClass.dart';  
  
void main() {  
  final myAccount = Bank(title: "Aamir");  
  print('The initial bank balance is ${myAccount.currentBalance}');  
  myAccount.deposit(100);  
  myAccount._balance = 50;  
  print('The current bank balance is ${myAccount.currentBalance}');  
}
```

app.dart

Error: The setter '\_balance' isn't defined for the class 'Bank'.

- 'Bank' is from 'bankClass.dart'.

Try correcting the name to the name of an existing setter, or defining a setter or field named '\_balance'.

```
myAccount._balance = 50;
```

^^^^^^^^

# Private members



```
class Bank {  
  Bank({required this.title});  
  
  final String title;  
  int _balance = 0;  
  
  void deposit(int amount) { _balance += amount; }  
  
  void withdraw(int amount) {  
    if (_balance > amount) { _balance -= amount; }  
  }  
  
  int currentBalance => _balance;  
}
```

bankClass.dart

```
import 'bankClass.dart';
```

```
void main() {  
  final myAccount = Bank(title: "Aamir");  
  print('The initial bank balance is ${myAccount.currentBalance}');  
  myAccount.deposit(100);  
  myAccount.withdraw(50);  
  print('The current bank balance is ${myAccount.currentBalance}');  
}
```

app.dart

The initial bank balance is 0  
The current bank balance is 50

# OOP Principles

---



## Abstraction

A prototype representation of important properties & functionality.

## Encapsulation

A virtual shell that holds data and operational features and hide the implementation from the user.

## OOP Principles

## Inheritance

Create a subclass that inherits features of parent class.

## Polymorphism

Same function name but possibility of a different implementation in subclass

extends - Subclass (Inheritance)

---

# extends - Subclass (Inheritance)

- Inheritance is a common concept in human, Usually we inherit many characteristics from our parent. For example: skin color, the way we walk or talk.
- A child can have many characteristics that are same as parent but some of them can also be different. For example: Reading paper book vs reading online book, sleeping early vs sleeping late.
- Sometimes a child has unique characteristic than parent. For example: child can swim but parent cannot.
- This inheritance may go from parent to child and child to grandchild and so on.



# Sub-class (Inheritance)

---

- The same way we can do inheritance with classes in any object oriented language.
- This is also known as **subclassing**.
- We **extends** the parent class and then we can have any method or property of parent class along with the new properties or methods of the child class using child class object instance.

# Sub-class (Inheritance)



```
class Person {  
  String? name;  
  String? id;  
  
  greet() {  
    print('Hello ${name}.');  
  }  
}
```

```
void main() {  
  final person = Person();  
  Person.name = "Aamir";  
  print(person.name);  
  person.greet();  
}
```

**A normal class WITHOUT any inheritance.**

```
Aamir  
Hello Aamir.
```

# Sub-class (Inheritance)



```
class Person {  
  String? name;  
  String? id;  
  
  greet() {  
    print('Hello ${name}.');  
  }  
}
```

Aamir  
Hello Aamir.  
ABC-123456

```
class Student extends Person {  
  String? studentId;  
}
```

```
void main() {  
  final student = Student();  
  student.name = "Aamir";  
  print(student.name);  
  student.greet();  
  student.id = 'ABC-123456';  
  print(student.id);  
}
```

# Sub-class (Inheritance)



```
class Person {  
    String? name;  
    String? id;  
  
    greet() {  
        print('Hello ${name}.');  
    }  
}
```

```
class Student extends Person {  
    double? fee;  
}
```

```
class Teacher extends Person {  
    double? salary;  
}
```

```
void main() {  
    final student = Student();  
    student.name = "Aamir";  
    print(student.name);  
    student.greet();  
    student.id = 'ABC-123456';  
    print(student.id);  
  
    final teacher = Teacher();  
    teacher.name = "Ali";  
    print(teacher.name);  
    teacher.greet();  
    teacher.id = 'XYZ-987654';  
    teacher.salary = 50000;  
    print(teacher.salary);  
}
```

```
Aamir  
Hello Aamir.  
ABC-123456  
Ali  
Hello Ali.  
50000.0
```

# Super constructor

---



# Super constructor



- Let's create a constructor in Person class to assign value to name property.

```
class Person {  
  Person({required this.name});  
  String name;  
  String? id;  
  
  greet() {  
    print('Hello ${name}.');  
  }  
}
```

```
class Student extends Person {  
  String? studentId;  
}
```

```
void main() {  
  final student = Student();  
  student.name = "Aamir";  
  print(student.name);  
  student.greet();  
  student.id = 'ABC-123456';  
  print(student.id);  
}
```

Unhandled exception:

NoSuchMethodError: No constructor 'Person.'  
with matching arguments declared in class  
'Person'.

# Super constructor

- The error occurred in previous solution because student that extends person do not pass any value to parent object (Person) for name property.
- **Super constructor** will help us **pass a value** from **child class** to **parent class**.

```
class Person {  
    Person({required this.name});  
    String name;  
    String? id;  
  
    greet() {  
        print('Hello ${name}.');  
    }  
}
```

```
class Student extends Person {  
    Student({required name}) : super(name: name);  
    String? studentId;  
}
```

```
void main() {  
    final student = Student(name: "Aamir");  
    print(student.name);  
    student.greet();  
    student.id = 'ABC-123456';  
    print(student.id);  
}
```

```
Aamir  
Hello Aamir.  
ABC-123456
```

# Method override

---

# Method override

- If the parent object has any method and we want to completely override or add more functionality to any of the child class.
- we can do by defining the method in child class with same name as it is in parent class.
- We can even call a parent class function with **super.methodName()** from the child class method.

```
class Person {  
  Person({required this.name});  
  String name;  
  String? id;  
  
  greet() {  
    print('Hello ${name}.');  
  }  
}
```

# Method override

- If the parent object has any method and we want to completely override or add more functionality to any of the child class.
- we can do by defining the method in child class with same name as it is in parent class.
- We can even call a parent class function with **super.methodName()** from the child class method.

# Method override



```
class Person {  
  Person({required this.name});  
  String name;  
  String? id;  
  
  greet() {  
    print('Hello ${name}.');  
  }  
}
```

Aamir  
Hello Aamir.  
This was from Student Class.  
ABC-123456

```
class Student extends Person {  
  Student({required name}) : super(name: name);  
  String? studentId;  
  
  @override  
  greet() {  
    super.greet();  
    print('This was from Student Class.');  }  
}  
  
void main() {  
  final student = Student(name: "Aamir");  
  student.greet();  
}
```

# Abstract Class

---

# Abstract Class

- Any class that contains one or more abstract methods is known as abstract class.
- An abstract class is a **template definition** of methods and variables of a class (category of objects) that contains one or more abstracted methods.
- To make a class abstract, add an abstract keyword before class definition.
- Abstract class may or may not include abstract methods.
- An abstract class can have abstract methods (methods without implementation) as well as concrete methods (methods with implementation).
- A normal class (non abstract class) is not allowed to have abstract methods.

**abstract**  
**keyword** to make  
the class abstract.

```
abstract class Person {  
    String name;  
  
    greet();  
}
```

**abstract method** without  
any method body. This will  
be implemented by  
subclass.



# Abstract Class

- An abstract class can not be instantiated, which means you are not allowed to create an object of it.
- Abstract classes can only be extended; and the subclass must provide implementations for all of the abstract methods in its parent class.
- If a subclass does not implements abstract methods, then the subclass must also be declared abstract.

# Abstract Class - Example



```
abstract class Person {  
  Person({required this.name});  
  
  String name;  
  String? id;  
  
  getId();  
}  
  
class Student extends Person {  
  Student({required name, required this.studentId}) : super(name: name);  
  
  String? studentId;  
  
  @override  
  getId() {  
    print('Hello ${name} - Student Id: ${studentId}');  
  }  
}
```

# Abstract Class - Example



```
class Trainer extends Person {
  Trainer({required name, required this.employeeId}) : super(name: name);

  String? employeeId;

  @override
  getId() {
    print('Hello ${name} - Employee Id: ${employeeId}');
  }
}

void main() {
  final student1 = Student(name: 'Aamir', studentId: 'ABC-123456');
  student1.getId();
  final trainer1 = Trainer(name: 'Ali', employeeId: 'EMP-987654');
  trainer1.getId();
}
```

Hello Aamir - Student Id: ABC-123456  
Hello Ali - Employee Id: EMP-987654

# Exercise

---

- Create an abstract class Animal

- Create private properties
  - Id
  - currentCost
  - Age
- Create abstract addCost method that takes amount as double value
- Create getDetails method
- Create getter and setter functions for rest above

## Expected output

```
Id: ABC-123456, Age: 0.8, Current Cost: 1010.0, canTalk: false  
Id: COW-123456, Age: 2.2, Current Cost: 5100.0, canMilk: true
```

- Create two abstract subclasses of above cattle and bird

- Add a canTalk as private property in birds class
- Add a canMilk private property in cattle class
- Create a getter and setter function for above
- Create a getDetails function for bird to print a message Id: xxxx, Age: xxxx, currentCost: xxxx, canFly: xxxx
- Create a getDetails function for cattle to print a message Id: xxxx, Age: xxxx, currentCost: xxxx, canMilk: xxxx

- Create two subclasses for Cow and Parrot

- addCost implementation for cow with 2% and bird with 1% tax increase in amount
- Implement the rest of the above methods

# Classes - Exercise - Solution



**animal.dart**

```
abstract class Animal {  
  Animal({required this.id});  
  
  final String id;  
  double _currentCost = 0;  
  double _age = 0;  
  
  getDetails();  
  
  addCost(double amount) => _currentCost += amount;  
  double get currentCost => _currentCost;  
  
  void set age(double age) => _age = age;  
  double get age => _age;  
}
```

# Classes - Exercise - Solution



bird.dart

```
import './animal.dart';

abstract class Bird extends Animal {
  Bird({required id}) : super(id: id);

  bool _canTalk = false;

  void set isTalkingBird(bool value) => _canTalk = value;
  bool get isTalkingBird => _canTalk;

  @override
  getDetails() {
    print(
      'Id: ${id}, Age: ${age}, Current Cost: ${currentCost}, canTalk:
${isTalkingBird}');
  }
}
```

# Classes - Exercise - Solution



cattle.dart

```
import './animal.dart';

abstract class Cattle extends Animal {
  Cattle({required id}) : super(id: id);

  bool _canMilk = false;

  void set isMilkingCattle(bool value) => _canMilk = value;
  bool get isMilkingCattle => _canMilk;

  @override
  getDetails() {
    print(
      'Id: ${id}, Age: ${age}, Current Cost: ${currentCost}, canMilk: ${isMilkingCattle}');
  }
}
```



# Classes - Exercise - Solution



cow.dart

```
import 'cattle.dart';

class Cow extends Cattle {
  Cow({required id}) : super(id: id);

  @override
  addCost(double amount) {
    super.addCost(amount * 1.02);
  }
}
```

# Classes - Exercise - Solution



parrot.dart

```
import 'bird.dart';

class Parrot extends Bird {
  Parrot({required id}) : super(id: id);

  @override
  addCost(double amount) {
    super.addCost(amount * 1.01);
  }
}
```

# Classes - Exercise - Solution



**my\_app.dart**

```
import 'cow.dart';
import 'parrot.dart';

void main() {
  final myPoly = Parrot(id: 'ABC-123456');
  myPoly.age = 0.8;
  myPoly.addCost(1000);
  myPoly.getDetails();

  final myMoo = Cow(id: 'COW-123456');
  myMoo.isMilkingCattle = true;
  myMoo.age = 2.2;
  myMoo.addCost(5000);
  myMoo.getDetails();
}
```

**output**

Id: ABC-123456, Age: 0.8, Current Cost: 1010.0, canTalk: false  
Id: COW-123456, Age: 2.2, Current Cost: 5100.0, canMilk: true

# Interfaces

---

- Think of an interface as a blueprint of a class
- When we have one parent class we use `extend` keyword but when we want to extend multiple classes we use **`implements`** instead of **`extends`**.
- In `extend` class we saw parent may or may not have abstract method and can have implemented methods
- When **`extend keyword`** is used, we have to only implement parent abstract method and we can use parent implemented method with or without overriding them
- When **`implements keyword`** is used, it is mandatory to override every property / method of parent classes even they have concrete implementation already in the parent class.

# Sub-class (Inheritance)



```
abstract class Add {  
  double a = 0, b = 0;  
  
  double add() {  
    return a + b;  
  }  
}
```

```
abstract class Divide {  
  double a = 0, b = 0;  
  
  double divide() {  
    return a / b;  
  }  
}
```

```
abstract class Subtract {  
  double a = 0, b = 0;  
  
  double sub();  
}
```

```
abstract class Multiply {  
  double a = 0, b = 0;  
  
  double multiply();  
}
```

# Sub-class (Inheritance)



```
class Arithmetic implements Add,
Subtract, Multiply, Divide {
    double a, b;

    Arithmetic(this.a, this.b);

    @override
    double add() {
        return a + b;
    }

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

```
@override
double multiply() {
    return a * b;
}

@override
double divide() {
    return a / b;
}

void main() {
    final myMaths = Arithmetic(10, 5);
    print('addition is: ${myMaths.add()}');
    print('subtraction is: ${myMaths.sub()}');
    print('multiplication is:
    ${myMaths.multiply()}');
    print('division is: ${myMaths.divide()}');
}
```

addition is: 15.0  
subtraction is: 5.0  
multiplication is: 50.0  
division is: 2.0

# Mixins

---



- A mixin is a sort of class that can be “associated” to another class in order to reuse pieces of code without using inheritance.
- As it is not inheritance then technically the mixins are not the parents class.
- It is far more efficient to reuse code from classes that share common behaviors.
- Mixin classes cannot be instantiated.
- Let's say We have classes, **falcon**, **lion**, and **Aeroplane**.
- Falcon and Lion can inherit from Animal class but lion cannot fly, falcon can.
- Falcon and Aeroplane can fly but Falcon and lion eats meat and Aeroplane is non living thing.
- Here we can use **mixin fly** and **mixin eat meat** and use them with the classes that requires these functionalities.

# Mixins - Example



```
mixin Fly {  
  void flying() {  
    print('Flying High. - ${this.toString()}');  
  }  
}
```

```
mixin EatMeat {  
  void eat() {  
    print('Eating Meat. - ${this.toString()}');  
  }  
}
```

```
class LivingThings {  
  void breathe() {  
    print('Breathing. - ${this.toString()}');  
  }  
}
```

# Mixins - Example

```
class Aeroplan with Fly {}  
  
class Lion extends LivingThings with EatMeat {}  
  
class Eagle extends LivingThings with EatMeat, Fly {}
```

```
void main() {  
    final aeroplane = Aeroplan();  
    aeroplane.flying();  
  
    final lion = Lion();  
    lion.breathe();  
    lion.eat();  
  
    final eagle = Eagle();  
    eagle.breathe();  
    eagle.flying();  
    eagle.eat();  
}
```

Flying High. - Instance of 'Aeroplan'  
Breathing. - Instance of 'Lion'  
Eating Meat. - Instance of 'Lion'  
Breathing. - Instance of 'Eagle'  
Flying High. - Instance of 'Eagle'  
Eating Meat. - Instance of 'Eagle'

- Mixins can sometime lead to confusion in case if different mixin have property/method with same names as other mixin.
- Compiler won't throw error and method or property from last mixin will be used, which may lead to unexpected output at run time.
- It cannot have constructor or accessed with super as it cannot be instantiated, that mean any assignment to properties will have be manually.

# Mixins - Example



```
mixin EatVeg {  
  void eat() {  
    print('Eating Veg. - ${this.toString()}');  
  }  
}
```

```
mixin EatMeat {  
  late int meatKcal;  
  void eat() {  
    print('Eating Meat. - ${this.toString()}');  
  }  
}
```

```
class LivingThings {  
  void breathe() {  
    print('Breathing. - ${this.toString()}');  
  }  
}
```

# Mixins - Example

```
class Lion extends LivingThings with EatMeat {}
```

```
class Person extends LivingThings with EatMeat, EatVeg{}
```

```
void main() {  
    final lion = Lion();  
    lion.breathe();  
    lion.eat();  
  
    final person = Person();  
    person.kcalMeat = 500;  
    person.breathe();  
    person.eat();  
    print(person.kcalMeat);  
}
```

Breathing. - Instance of 'Lion'  
Eating Meat. - Instance of 'Lion'  
Breathing. - Instance of 'Person'  
Eating Veg. - Instance of 'Person'  
500

toString()

---

# toString()

- Everything you can place in a variable is an object
- Every object is an instance of a class.
- Even numbers, functions, and null are objects.
- With the exception of null, all objects inherit from the Object class.
- The object class comes with few default properties and method that we get with every every object we create.
- One of the method is **toString()** that will be used quite a lot working on projects.
- We can override this method in our classes to get some useful information for the class.



# toString()

```
class Cattle {  
  Cattle({required this.type});  
  String type;  
}  
  
class Bird {  
  Bird({required this.type});  
  String type;  
  
  @override  
  String toString() => 'The bird type is $type';  
}  
  
void main(List<String> args) {  
  var myCattle = Cattle(type: 'Sheep');  
  print(myCattle.toString());  
  
  var myBird = Bird(type: 'Parrot');  
  print(myBird.toString());  
}
```

Instance of 'Cattle'  
The bird type is Parrot

# Extension methods

---

# Extension methods

- Extension methods, are a way to add functionality to existing library's classes.
- It is basically adding you own method in dart's or someone else class.
- For example, we used **double.parse('10.50')** or **int.parse('10')**, we can create better short syntax function for both of these.
- Something like **'10.50'.parseDouble()** and **'42'.parseInt()**

# Extension methods - Example



```
extension NumberParsing on String {  
  int parseInt() {  
    return int.parse(this);  
  }  
  
  double parseDouble() {  
    return double.parse(this);  
  }  
}
```

```
void main() {  
  print('10.50'.parseDouble());  
  print('10'.parseInt());  
}
```

10.5  
10

# Generic Extensions

---

# Extension methods

- Dart extensions can be used with generic types.
- For example, let's create an extension on list to have a sum of values, like,  
**[1, 2, 3, 4].sum()**
- You can create a normal extension like following

```
extension IterableNum on Iterable<int> {  
  int sum() => reduce((value, element) => value + element);  
}
```

```
void main() {  
  int intVal = [1, 2, 3, 4].sum();  
  print(intVal);  
}
```

10

# Extension methods

- But what if we pass double instead of int values

`[1.5, 2.5, 3, 4].sum()`

- With the same extension, compiler will throw an error.

```
extension IterableNum on Iterable<int> {  
  int sum() => reduce((value, element) => value + element);  
}
```

```
void main() {  
  int intVal = [1, 2, 3, 4].sum();  
  double doubleVal = [1.5, 2.5, 3.5, 4.5].sum();  
  print(intVal);  
  print(doubleVal);  
}
```

The method 'sum' isn't defined for the class 'List<double>'.

- 'List' is from 'dart:core'.

Try correcting the name to the name of an existing method, or defining a method named 'sum'.

```
print([1.5, 2.5, 3, 4].sum());
```

^^^

# Extension methods



- To solve this issue we can again have a num instead of int
- This will work but as mentioned earlier num typed cannot be assigned to int or double

```
extension IterableNum on Iterable<num> {  
  num sum() => reduce((value, element) => value + element);  
}
```

```
void main() {  
  int intVal = [1, 2, 3, 4].sum();  
  double doubleVal = [1.5, 2.5, 3.5, 4.5].sum();  
  print(intVal);  
  print(doubleVal);  
}
```

Error: A value of type 'num' can't be returned from a function with return type 'int'.

```
return this.reduce((value, element) => value + element);  
                  ^
```



# Extension methods

- The solution for this problem is generic type, usually denote as `<T>`
- Dart compiler will be then flexible enough to receive **int or double** and process it

```
extension IterableNum<T extends num> on Iterable<T> {  
  T sum() => reduce((value, element) => (value + element) as T);  
}
```

```
void main() {  
  int intVal = [1, 2, 3, 4].sum();  
  double doubleVal = [1.5, 2.5, 3.5, 4.5].sum();  
  print(intVal);  
  print(doubleVal);  
}
```

```
10  
12
```

# Exercise

---

# Generic Extensions - Exercise

- Create a generic extension for `.unique()` method on list and returns a list with unique value.
- Values can be int or double or it could be string as well

```
void main() {  
    List<int> intVal = [1, 2, 1, 4].unique();  
    List<double> doubleVal = [1.5, 4.5, 3.5, 4.5].unique();  
    List<String> stringVal = ["aamir", "pinger", "zahid", "aamir"].unique();  
    print(intVal);  
    print(doubleVal);  
    print(stringVal);  
}
```

[1, 2, 4]

[1.5, 4.5, 3.5]

[aamir, pinger, zahid]

**Expected output**

# Generic Extensions - Exercise - Solution



```
extension IterableNum<T extends Object> on Iterable<T> {  
    List<T> unique() => this.toSet().toList();  
}
```

```
void main() {  
    List<int> intVal = [1, 2, 1, 4].unique();  
    List<double> doubleVal = [1.5, 4.5, 3.5, 4.5].unique();  
    List<String> stringVal = ["aamir", "pinger", "zahid", "aamir"].unique();  
    print(intVal);  
    print(doubleVal);  
    print(stringVal);  
}
```

```
[1, 2, 4]  
[1.5, 4.5, 3.5]  
[aamir, pinger, zahid]
```

# Composition

---

- Composition is to gather two or more objects and wrap them into one.
- Composition is core of any application we build today.
- Composition is a way to combine simple objects to create complex ones.
- For example, when making a computer you put together a motherboard, CPU, GPU, RAM, and a hard drive. This is composition.

- In Composition, on the other hand, a class contains instances of other classes which bring their own properties and behavior to the containing class.
- In Flutter the composition is widely use to create an UI, For example you create a independent search-box class (widget) and a table class (widget) and then you combine them to composite a searchable table, and reuse that composite class (widget) wherever you want.

# Composition



```
abstract class Widget {}
```

```
class Image extends Widget {  
  Image({required this.url});  
  String url;  
  
  @override  
  String toString() => url;  
}
```

```
void printMessage() =>  
  print('This button is  
  pressed.');
```

```
class Text extends Widget {  
  Text({required this.text});  
  String text;  
  
  @override  
  String toString() => text;  
}
```

```
class Button {  
  Button({  
    required this.id,  
    required this.child,  
    required this.onPress,  
  });  
  final id;  
  final Widget child;  
  final void Function() onPress;  
}
```



# Composition



```
void main() {  
  final saveButton = Button(  
    id: 'btx-1',  
    child: Image(url: 'http://example.com/car.jpg'),  
    onPress: printMessage,  
  );  
  
  print(saveButton.child.toString());  
  saveButton.onPress();  
  
  final cancelButton = Button(  
    id: 'btx-2',  
    child: Text(text: 'Cancel & go back'),  
    onPress: printMessage,  
  );  
  
  print(cancelButton.child.toString());  
  cancelButton.onPress();  
}
```

http://example.com/car.jpg  
This button is pressed.  
Cancel & go back  
This button is pressed.

# Factory constructors

---

# Factory constructors

- When we write `final myMoo = Person(name: 'Aamir');` we call a constructor the the Person class.
- This type of normal constructors always returns a new instance of the current class (except when the constructor throws an exception).

```
class Person {  
  Person({required this.name});  
  String name;  
}  
  
void main() {  
  final me = Person(name: 'Aamir');  
}
```

**Example of a normal constructor**

# Factory constructors

- A factory constructor is quite similar to a static method with the differences that
  - factory keyword is added before constructor name.
  - it can only return an instance of the current class OR one of its subclasses
  - has no initializer list (no : super())
  - A factory can use control flow to determine what object to return, and must utilize the return keyword.
  - In order for a factory to return a new class instance, it must first call a normal constructor of the returning class.

# Factory constructors - Example



```
class Trainer extends Person {  
  Trainer({required name,  
    required this.salary}) :  
    super(name: name);  
  int salary;  
  
  void getDetails() =>  
    print('Trainer name: $name,  
    salary: $salary');  
}
```

```
class Student extends Person {  
  Student({required name,  
    required this.fee}) :  
    super(name: name);  
  int fee;  
  
  void getDetails() =>  
    print('Student name: $name, fee:  
    $fee');  
}
```

# Factory constructors - Example



```
abstract class Person {  
  Person({required this.name});  
  String name;  
  
  void getDetails();  
  
  factory Person.fromType({required typeName, required name, salary,  
fee}) {  
    if (typeName == 'Trainer') return Trainer(salary: salary, name:  
name);  
    if (typeName == 'Student') return Student(fee: fee, name: name);  
  
    throw UnimplementedError('Unable to recognize $typeName');  
  }  
}
```

# Factory constructors - Example



```
void main() {  
    final dartTrainer =  
        Person.fromType(typeName: 'Trainer', name: 'Aamir', salary: 1000);  
    final student1 = Person.fromType(typeName: 'Student', name: 'Ali', fee: 500);  
  
    dartTrainer.getDetails();  
    student1.getDetails();  
  
    final unknowType =  
        Person.fromType(typeName: 'director', name: 'John', salary: 1500);  
    unknowType.getDetails();  
}
```

Trainer name: Aamir, salary: 1000

Student name: Ali, fee: 500

Unhandled exception:

UnimplementedError: Unable to recognized director

# Exercise

---



# Factory constructors - Exercise

- Create the same Person, Trainer, and student class.
- The abstract class should have **fromJson** factory constructor that accepts a json.
- Use Switch case instead of if conditions.

The main function should be as following

```
void main() {  
  
    final dartTrainer =  
        Person.fromJson(json: {'name': 'Aamir', 'salary': 1000, 'type': 'Trainer'});  
    final student1 =  
        Person.fromJson(json: {'name': 'Ali', 'fee': 500, 'type': 'Student'});  
  
    dartTrainer.getDetails();  
    student1.getDetails();  
  
    final unknowType =  
        Person.fromJson(json: {'name': 'John', 'fee': 1500, 'type': 'Director'});  
    unknowType.getDetails();  
}
```

# Factory constructors - Exercise - Solution



```
class Trainer extends Person {  
  Trainer({required name,  
    required this.salary}) :  
    super(name: name);  
  int salary;  
  
  void getDetails() =>  
    print('Trainer name: $name,  
    salary: $salary');  
}
```

```
class Student extends Person {  
  Student({required name,  
    required this.fee}) :  
    super(name: name);  
  int fee;  
  
  void getDetails() =>  
    print('Student name: $name, fee:  
    $fee');  
}
```

# Factory constructors - Exercise - Solution



```
abstract class Person {  
    Person({required this.name});  
    String name;  
  
    void getDetails();  
  
    factory Person.fromJson({required Map<String, Object> json}) {  
        final salary = json['salary'];  
        final type = json['type'];  
        final fee = json['fee'];  
        final name = json['name'];  
  
        final obj;
```

# Factory constructors - Exercise - Solution



```
switch (type) {
  case 'Trainer':
    if (salary is int) {
      obj = Trainer(salary: salary, name: name);
      break;
    }
    throw UnimplementedError('Invalid salary.');
```

```
  case 'Student':
    if (fee is int) {
      obj = Student(fee: fee, name: name);
      break;
    }
    throw UnimplementedError('Invalid salary.');
```

```
  default:
    throw UnimplementedError('Unable to recognized $type');
```

```
}
```

```
return obj;
```

```
}
```

```
}
```

# Factory constructors - Exercise - Solution



```
void main() {  
  
  final dartTrainer =  
    Person.fromJson(json: {'name': 'Aamir', 'salary': 1000, 'type': 'Trainer'});  
  final student1 =  
    Person.fromJson(json: {'name': 'Ali', 'fee': 500, 'type': 'Student'});  
  
  dartTrainer.getDetails();  
  student1.getDetails();  
  
  final unknowType =  
    Person.fromJson(json: {'name': 'John', 'fee': 1500, 'type': 'Director'});  
  unknowType.getDetails();  
}
```

**Trainer name: Aamir, salary: 1000**

**Student name: Ali, fee: 500**

**Unhandled exception:**

**UnimplementedError: Unable to recognized director**

# Cascade notation

---

# Cascade notation



- Let's suppose we have a student class and we wanted to update the course, how we can do that?

```
class Student {  
  final int id;  
  final String name;  
  String? course;  
  String? shift;  
  
  Student(  
    {required this.id,  
    required this.name});  
  
  @override  
  String toString() {  
    return 'id: ${id}, name: $name, course:  
$course, shift: $shift';  
  }  
}
```

```
void main() {  
  final student1 =  
    Student(id: 1, name: "Ali");  
  
  student1.course = "Dart";  
  student1.shift = "Morning";  
  
  print(student1);  
}
```

id: 1, name: Ali, course: Dart, shift: Morning

# Cascade notation



- A better way to chain the steps by using Cascade notation

```
class Student {  
  final int id;  
  final String name;  
  String? course;  
  String? shift;  
  
  Student(  
    {required this.id,  
    required this.name});  
  
  @override  
  String toString() {  
    return 'id: ${id}, name: $name, course:  
$course, shift: $shift';  
  }  
}
```

```
void main() {  
  final student1 =  
    Student(id: 1, name: "Ali")  
    ..course = "Dart"  
    ..shift = "Morning";  
  
  print(student1);  
}
```

id: 1, name: Ali, course: Dart, shift: Morning



copyWith

---

# copyWith method

- copyWith method is a handy function that will return an object instance similar to a source object.
- This method not only copies the object but does provides a way to pass all or some new properties that needed to be updated.
- Think of a list of student object of any particular class, the student id and name would be different but the class timings and course would be the same for all students.
- In above we can create all object from scratch by providing values to all properties everytime OR create one object and keep making clone with updated id and name of the student in each object.

# copyWith method - Example



```
class Student {  
  final int id;  
  final String name;  
  final String course;  
  final String shift;  
  
  Student(  
    {required this.id,  
    required this.name,  
    required this.course,  
    required this.shift});  
  
  @override  
  String toString() {  
    return 'id: ${id}, name: $name, course: $course, shift: $shift';  
  }  
}
```

# copyWith method - Example



```
Student copyWith(  
  {required int id, required String name, String? course, String? shift}) {  
  return Student(  
    id: id,  
    name: name,  
    course: course ?? this.course,  
    shift: shift ?? this.shift,  
  );  
}
```

# copyWith method - Example

```
void main() {  
    final student1 =  
        Student(id: 1, name: "Ali", course: "Dart", shift: "morning");  
  
    final student2 = student1.copyWith(id: 2, name: 'Atif');  
    final student3 = student1.copyWith(id: 3, name: "Jim");  
    final student4 = student1.copyWith(id: 4, name: "Jim");  
    print(student1);  
    print(student2);  
    print(student3);  
    print(student4);  
}
```

```
id: 1, name: Ali, course: Dart, shift: morning  
id: 2, name: Atif, course: Dart, shift: morning  
id: 3, name: Jim, course: Dart, shift: morning  
id: 4, name: Jim, course: Dart, shift: morning
```

# Exercise

---

# A book shop app

---

# Exercise - A book shop app



## Create an Dart command line app for a book shop

- The shop will have following
  - Shop name
  - Available Books
  - Supporting methods
- The book object will have
  - id
  - Book title
  - Author
  - Price
  - Supporting method
- Item class will have following
  - book
  - Qty purchased

- Bill class will have following
  - Id (random generated)
  - Selected books
  - Total bill
  - Supporting methods
- User will be asked for  
**Please select: (v)iew items, (s)elect book, (c)heckout, (e)xit:**
- On **V**, list all selected books with qty and amount
- On **S**, show available book to purchase
- The User will input the book add to add to the bill.
- On **C**, Print **Bill # 831 - Total Bill Amount: 72.0**
- On **E**, Exit the app.



# Exercise - A book shop app



```
Please select: (v)view items, (s)elect book, (c)heckout, (e)xit: s
Available books:
(1) Rich Dad Poor Dad (Robert T. Kiyosaki) - (Rs.35.0)
(2) A short history of the world (Christopher Lascelles) - (Rs.30.0)
(3) How to win friends and influence people (Dale Carnegie) - (Rs.25.0)
(4) Ikigai (Francesc Miralles and Hector Garcia) - (Rs.37.0)
(5) The essential Rumi (Coleman Barks) - (Rs.50.0)
Your choice: 1
Please select: (v)view items, (s)elect book, (c)heckout, (e)xit: s
Available books:
(1) Rich Dad Poor Dad (Robert T. Kiyosaki) - (Rs.35.0)
(2) A short history of the world (Christopher Lascelles) - (Rs.30.0)
(3) How to win friends and influence people (Dale Carnegie) - (Rs.25.0)
(4) Ikigai (Francesc Miralles and Hector Garcia) - (Rs.37.0)
(5) The essential Rumi (Coleman Barks) - (Rs.50.0)
Your choice: 1
Please select: (v)view items, (s)elect book, (c)heckout, (e)xit: s
Available books:
(1) Rich Dad Poor Dad (Robert T. Kiyosaki) - (Rs.35.0)
(2) A short history of the world (Christopher Lascelles) - (Rs.30.0)
(3) How to win friends and influence people (Dale Carnegie) - (Rs.25.0)
(4) Ikigai (Francesc Miralles and Hector Garcia) - (Rs.37.0)
(5) The essential Rumi (Coleman Barks) - (Rs.50.0)
Your choice: 2
Please select: (v)view items, (s)elect book, (c)heckout, (e)xit: v
(1) Rich Dad Poor Dad (Robert T. Kiyosaki) - (Rs.35.0 x 2 = 70.0)
(2) A short history of the world (Christopher Lascelles) - (Rs.30.0 x 1 = 30.0)
Please select: (v)view items, (s)elect book, (c)heckout, (e)xit: c
Bill # 528 - Total Bill Amount: 100.0
Please select: (v)view items, (s)elect book, (c)heckout, (e)xit: e
```

# Exercise - A book shop app - Solution



```
E:\> dart create book-shop
```

```
E:\> cd book-shop
```

```
E:\book-shop> pub add collection
```

```
E:\book-shop> pub add colorize
```

```
E:\book-shop> code .
```

# Exercise - A book shop app - Solution



bin/shop.dart

```
import 'package:collection/collection.dart';
import 'book.dart';

class Shop {
  Shop({required this.name});

  final name;
  final List<Book> _books = [];

  List<String> get availableBooks {
    return _books.map((book) => book.getDetails()).toList();
  }

  set addBooks(List<Book> newStock) => _books.addAll(newStock);

  Book? getBookById(String id) =>
    _books.firstWhereOrNull((book) => book.id == (id));
}
```

# Exercise - A book shop app - Solution



bin/book.dart

```
class Book {  
  Book({  
    required this.id,  
    required this.title,  
    required this.author,  
    required this.price,  
  });  
  
  final String id;  
  final String title;  
  final String author;  
  double price;  
  
  String getDetails() => '(${id}) ${title} (${author}) - (Rs.${price})';  
}
```

# Exercise - A book shop app - Solution



bin/item.dart

```
import 'dart:math';  
import 'book.dart';  
  
class Item {  
  Item({required this.book, required this.qty});  
  
  int qty;  
  Book book;  
}
```

# Exercise - A book shop app - Solution



bin/bill.dart

```
import 'dart:math';
import 'book.dart';
import 'item.dart';

class Bill {
  Bill();

  final id = Random().nextInt(1000);
  double _bill = 0;
  Map<String, Item> _books = {};

  void addBook(Book book) {
    if (_books[book.id] == null) {
      _books[book.id] = Item(book: book, qty: 1);
    } else {
      _books[book.id] = Item(book: book, qty: _books[book.id]!.qty + 1);
    }

    _bill += book.price;
  }
}
```

# Exercise - A book shop app - Solution



bin/bill.dart

```
String get totalBill => 'Bill # $id - Total Bill Amount: $_bill';

List<String>? get viewBooks {
  final List<String> details = [];
  _books.forEach((key, value) {
    final book = value.book;
    details.add(
      '(${book.id}) ${book.title} (${book.author}) - (Rs.${book.price} x
${value.qty} = ${book.price * value.qty})');
  });
  return details;
}
```

# Exercise - A book shop app - Solution



bin/bookshop.dart

```
import 'package:colorize/colorize.dart';
import 'dart:io';

import 'bill.dart';
import 'book.dart';
import 'shop.dart';

void main(List<String> arguments) {
  // Setting up shop with initial book stock
  final shop = Shop(name: "The book shop");
  shop.addBooks = [
    Book(
      id: '1',
      title: "Rich Dad Poor Dad",
      author: 'Robert T. Kiyosaki',
      price: 35.0,
    ),
  ],
```



# Exercise - A book shop app - Solution



bin/bookshop.dart

```
Book(  
  id: '2',  
  title: "A short history of the world",  
  author: 'Christopher Lascelles',  
  price: 30.0,  
) ,  
Book(  
  id: '3',  
  title: "How to win friends and influence people",  
  author: 'Dale Carnegie',  
  price: 25.0,  
) ,  
Book(  
  id: '4',  
  title: "Ikigai",  
  author: 'Francesc Miralles and Hector Garcia',  
  price: 37.0,  
) ,
```

# Exercise - A book shop app - Solution



bin/bookshop.dart

```
Book(  
  id: '5',  
  title: "The essential Rumi",  
  author: 'Coleman Barks',  
  price: 50.0,  
)  
];  
openShop(shop);  
}  
  
void openShop(Shop shop) {  
  final bill = Bill();  
  while (true) {  
    stdout.write(Colorize(  
      'Please select: (v)iew items, (s)elect book, (c)heckout, (e)xit: '  
      .lightBlue()));  
    final line = stdin.readLineSync().toString();  
    performAction(shop, bill, line);  
  }  
}
```

# Exercise - A book shop app - Solution



```
void performAction(Shop shop, Bill bill, String line) {  
  switch (line) {  
    case 's':  
      final book = showBookList(shop);  
      if (book != null) {  
        bill.addBook(book);  
      }  
      break;  
    case 'v':  
      bill.viewBooks?.forEach((element) {  
        print(Colorize('$element \n').lightGreen());  
      });  
      break;  
    case 'c':  
      print(Colorize(bill.totalBill).black().bgLightCyan());  
      break;  
    case 'e':  
      exit(0);  
  }  
}
```

bin/bookshop.dart

# Exercise - A book shop app - Solution



bin/bookshop.dart

```
Book? showBookList(Shop shop) {
  print(Colorize('Available books:')..bgLightMagenta().black()),

  shop.availableBooks.forEach((book) => print(book));

  String? userInput;
  while (true) {
    stdout.write(Colorize('Your choice: ')..bgLightMagenta().black());

    userInput = stdin.readLineSync();
    if (userInput!.trim().isEmpty) { break; }
  }

  Book? bookFound = shop.getBookById(userInput);
  if (bookFound is Book) { return bookFound; }

  print('Book not found. ');
  return null;
}
```

# Exceptions

---

# Exceptions

- An exception is a errors that occurs during the execution of a program.
- When an Exception occurs the normal flow of the program is disrupted and two possible things could be done
  - Application terminates abnormally.
  - You catch the exceptions, do something to handle the error and move on.

```
void main(List<String> args) {  
  final arr = [1, 2, 3, 4, 5];  
  print(arr[10]);  
}
```

## Unhandled exception:

RangeError (index): Invalid value: Not in inclusive range 0..4: 10

#0 List[] (dart:core-patch/growable\_array.dart:254:60)

#1 main app.dart:8

#2 \_delayEntrypointInvocation.<anonymous closure> (dart:isolate-patch/isolate\_patch.dart:281:32)

#3 \_RawReceivePortImpl.\_handleMessage (dart:isolate-patch/isolate\_patch.dart:184:12)

Exited (255)

# Assert

---

- To pre-check some of the errors during development, **Assert** is right tool.
- An assert statement disrupt normal execution if a boolean condition is false.
- The execution ends with an `AssertionError`.
- If the boolean expression is true, then the code continues to execute normally.
- It is kind of a if condition for error checks, if all ok, carry on if not stop!
- Important to remember,
  - In production code, assertions are ignored, and the arguments to assert aren't evaluated.
  - Some tools, such as `dart run` and `dart2js` support assertions through a command-line flag: **`--enable-asserts`**.



# Assert - Example



```
void validateURL({required String url}) {  
    assert(url.startsWith('https'), 'The url should start with https');  
  
    print('URL is OK.');
```

```
}  
  
void main() {  
    validateURL(url: "http://example.com");  
}
```

```
E:\> dart --enable-asserts app.dart
```

Unhandled exception:

'file:///E:/app.dart': Failed assertion: line 2 pos 10: 'url.startsWith('https')': The url should start with https

```
#0    _AssertionError._doThrowNew (dart:core-patch/errors_patch.dart:46:39)  
#1    _AssertionError._throwNew (dart:core-patch/errors_patch.dart:36:5)  
#2    validateURL (file:///E:/app.dart:2:10)  
#3    main (file:///E:/app.dart:8:3)  
#4    _delayEntrypointInvocation.<anonymous closure> (dart:isolate-patch/isolate_patch.dart:281:32)  
#5    _RawReceivePortImpl._handleMessage (dart:isolate-patch/isolate_patch.dart:184:12)
```

# Assert - Example

```
class Person {  
  Person({required this.name, required this.age});  
  String name;  
  int age;  
}  
  
void main() {  
  final person1 = Person(name: "Aamir", age: -40);  
  
  print('Name: ${person1.name} Age: ${person1.age}');  
}
```

Age cannot be negative

```
E:\> dart --enable-asserts app.dart  
Name: Aamir Age: -40
```

# Assert - Example

```
class Person {  
  Person({required this.name, required this.age}) : assert(age > 0, 'Age must be  
greater than zero.');
```

```
  String name;  
  int age;  
}
```

```
void main() {  
  final person1 = Person(name: "Aamir", age: -40);  
  
  print('Name: ${person1.name} Age: ${person1.age}');
```

```
}
```

E:\> dart --enable-asserts app.dart

Unhandled exception:

'file:///E:/app.dart': Failed assertion: line 3 pos 16: 'age > 0': Age must be greater than zero.

# Try catch finally

---

# try ... catch finally

```
class Arithmetic {  
  Arithmetic({required this.values})  
    : assert(values.length == 2, 'Value list must contain 2 elements.');
```

List<int> values;

```
  int get divide => values[0] ~/ values[1];  
}
```

```
void doDivision() {  
  final myMaths = Arithmetic(values: [10,2]);  
  final answer = myMaths.divide;  
  print(answer);  
}
```

```
void main() {  
  doDivision();  
  print('All is well!');  
}
```

This works well

5

All is well!

# try ... catch finally

```
class Arithmetic {  
  Arithmetic({required this.values})  
    : assert(values.length == 2, 'Value list must contain 2 elements.');
```

List<int> values;

```
  int get divide => values[0] ~/ values[1];  
}
```

```
void doDivision() {  
  final myMaths = Arithmetic(values: [10]);  
  final answer = myMaths.divide;  
  print(answer);  
}
```

```
void main() {  
  doDivision();  
  print('All is well!');  
}
```

## Exception

Unhandled exception:

'file:///E:/app.dart': Failed assertion: line 3 pos 16: 'values.length == 2': Value list must contain 2 elements.

# try ... catch finally

```
class Arithmetic {  
  Arithmetic({required this.values})  
    : assert(values.length == 2, 'Value list must contain 2 elements.');
```

List<int> values;

```
  int get divide => values[0] ~/ values[1];  
}
```

```
void doDivision() {  
  final myMaths = Arithmetic(values: [10,0]);  
  final answer = myMaths.divide;  
  print(answer);  
}
```

```
void main() {  
  doDivision();  
  print('All is well!');  
}
```

Unhandled exception:  
IntegerDivisionByZeroException

Another exception

# try ... catch finally

```
class Arithmetic {  
  Arithmetic({required this.values})  
    : assert(values.length == 2, 'Value list must contain 2 elements.');
```

List<int> values;

```
  int get divide => values[0] ~/ values[1];  
}
```

```
void doDivision() {  
  try {  
    final myMaths = Arithmetic(values: [10,0]);  
    final answer = myMaths.divide;  
    print(answer);  
  } catch (e) {  
    print(e);  
  }  
}
```

```
void main() {  
  doDivision();  
  print('All is well!');  
}
```

Error but app didn't crashed.

IntegerDivisionByZeroException  
All is well!



# try ... catch finally

```
class Arithmetic {  
  Arithmetic({required this.values})  
    : assert(values.length == 2, 'Value list must contain 2 elements.');
```

List<int> values;

```
  int get divide => values[0] ~/ values[1];  
}
```

```
void doDivision() {  
  try {  
    final myMaths = Arithmetic(values: [10,0]);  
    final answer = myMaths.divide;  
    print(answer);  
  } catch (e) {  
    print(e);  
  }  
}
```

```
void main() {  
  doDivision();  
  print('All is well!');  
}
```

Error but app didn't crashed.

IntegerDivisionByZeroException  
All is well!

# try ... catch finally

```
void doDivision() {  
  try {  
    final myMaths = Arithmetic(values: [10, 0]);  
    final answer = myMaths.divide;  
    print(answer);  
  } on IntegerDivisionByZeroException catch (e) {  
    print('Cannot divide by zero.');  } catch (e) {  
    print('Other error.');  } finally {  
    print('this will be printed in any exception or no exception');  }  
}
```

Better error handling

Cannot divide by zero.  
this will be printed in any exception or no excpetion  
All is well!

rethrow

---

```
class Arithmetic {  
  Arithmetic({required this.values})  
    : assert(values.length == 2, 'Value list must contain 2 elements.');
```

List<int> values;

```
  int get divide => values[0] ~/ values[1];  
}
```

```
void doDivision() {  
  try {  
    final myMaths = Arithmetic(values: [10,0]);  
    final answer = myMaths.divide;  
    print(answer);  
  } catch (e) {  
    rethrow;  
  }  
}
```

```
void main() {  
  try {  
    doDivision();  
    print('All is well!');  
  } catch (e) {  
    print('this is main catch!');  
  }  
}
```

this is main catch!

throw

---

# throw



- You can generate your own exception using **throw Exception()**

```
main(List<String> args) {  
  try {  
    if (args.length != 1) {  
      throw Exception('Args length must be one.');    }  
  
    print(args[0]);  
  } catch (e) {  
    print('Oops! $e');  
  }  
}
```

Oops! Exception: Args length must be one.

# Asynchronous Dart

---

# Event loop

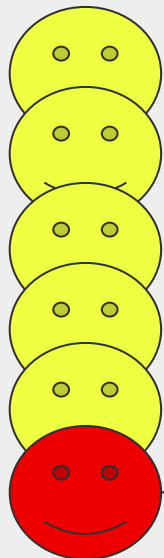
---



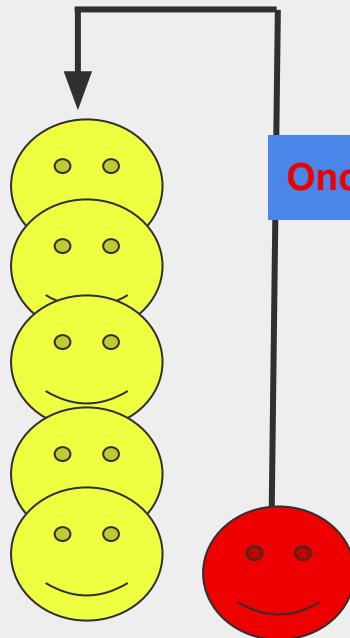
# Event loop

- Dart is the single threaded language
- Event loop handles the execution of multiple chunks of your program over time
- The event loop is what allows multiple operation in a non-blocking way
- The multi-threaded system kernel helps handle multiple operations executing in the background
- If there is a piece of code which may delay the response, we can tell the event loop to keep it a side until response is received.
- When one of these operations completes, the kernel tells event loop so that the appropriate callback may queued to eventually be executed

# Blocking vs Non-Blocking



**Will take time to receive the data**



**Once done, rejoin the queue**

Future

---

- Future in dart can be referred as a real life promise
- You keep the promise or you break it, in the middle it is uncertain or pending
- In dart, Future may give you some valid response (e.g. promise kept), or you may get error (e.g. promise broke)
- Calling APIs, reading file from harrdisk, or reading/writing to the database

`anyFuturisticMethod().then( callMethodA ).catchError( callMethodB ).whenComplete(callMethodC)`

The Future function that will return response/error once received

If response is not error, call this method

If response is error, call this method

In any response error or non-error, call this method in the end

# Future



```
void fetchUserProfile() {  
  Future.delayed(Duration(seconds: 2), () => print('Name: Aamir'));  
}  
  
void main() {  
  print('Fetching user profile...');  
  fetchUserProfile();  
  print('Done...');  
}
```

```
Fetching user profile...  
Done...  
Name: Aamir
```

# Future



```
Future<Map<String, String>> fetchUserProfile() {  
    return Future.delayed(const Duration(seconds: 2), () => Future.value({"name":  
    "Aamir"}));  
}  
  
void main() {  
    print('Fetching user profile...');  
    fetchUserProfile().then((value) => print('The name is ${value["name"]}'));  
    print('Done...');  
}
```

```
Fetching user profile...  
Done...  
The name is Aamir
```

# Future



```
Future<Map<String, String>> fetchUserProfile() {  
  return Future.delayed(  
    const Duration(seconds: 2), () => Future.error("Error"));  
}  
  
void main() {  
  print('Fetching user profile...');  
  fetchUserProfile()  
    .then((value) => print('The name is ${value["name"]}'))  
    .catchError((err) {  
      print('Some wrong, ${err.toString()}');  
    });  
  print('Done...');  
}
```

```
Fetching user profile...  
Done...  
Some wrong, Exception: Error
```

# Future



```
Future<Map<String, String>> fetchUserProfile() {  
  return Future.delayed(  
    const Duration(seconds: 2), () => throw Exception("Error");  
}
```

```
void main() {  
  print('Fetching user profile...');  
  fetchUserProfile()  
    .then((value) => print('The name is ${value["name"]}'))  
    .catchError((err) {  
      print('Some wrong, ${err.toString()}');  
    })  
    .whenComplete(() => print('Done...'));  
}
```

```
Fetching user profile...  
Some wrong, Exception: Error  
Done...
```



# Async / Await

---

- When we call Future functions, surely we wanted to perform some actions on the data that will be received from Future functions.
- Some actions could be calling another Future Function and calling Future function after the second one too.
- Soon the code become unmanageable and messy.
- For this purpose we use **async** and **await**
- By using these we tell Dart to wait to move further until the response is received from Future functions
- Technically, we make asynchronous function to synchronous function!

# Async / Await



```
Future<Map<String, String>> fetchUserName() {  
  return Future.delayed(const Duration(seconds: 2), () => {"name": "Aamir"});  
}
```

```
Future<String> fetchOtherData() {  
  return Future.delayed(  
    const Duration(seconds: 2), () => "This is from another API.");  
}
```

```
void main() {  
  print('Fetching user profile...');  
  fetchUserName()  
    .then((value) => print('The name is ${value["name"]}'))  
    .catchError((err) {  
      print('Some wrong, ${err.toString()}');  
    }).whenComplete(() =>  
      fetchOtherData().then((value) => print(value)).catchError((err) {  
        print('Some wrong, ${err.toString()}');  
      }).whenComplete(() => print('Done...')));  
}
```

# Async / Await



```
void main() async {  
  try {  
  
    print('Fetching user profile...');  
  
    final value = await fetchUserName();  
    print('The name is ${value["name"]}');  
  
    final value2 = await fetchOtherData();  
    print(value2);  
  
  } catch (err) {  
    print('Some wrong, ${err.toString()}');  
  
  } finally {  
    print('Done...');  
  }  
}
```

# Exercise

---

# Async / Await - Exercise

- Create a **future function** that **returns random integer (0 to 20)** after a **delay of 1 second**.
- **Call this future function** and **print asterisks \* x times the integer** returns from future function. E.g. assume function returns 5 then print \*\*\*\*\*
- **Use async and await**
- All the asterisks should be **printed in one horizontal line**.
- Call this future function **until the sum of values** received **does not exceed 100**.
- **Print “Download starting...” before** calling this future function
- **Print [ before and print ]** after all the asterisks. E.g. [\*\*\*\*\*]
- **Print “Download completed...”** in the next line only if download complete successfully
- Use **Try..catch**

# Async / Await - Exercise - Solution



```
import 'dart:io';  
import 'dart:math';  
  
Future<int> getChunk() {  
    return Future.delayed(const Duration(seconds: 1), () => Random().nextInt(20));  
}
```



# Async / Await - Exercise - Solution



```
void main() async {  
  try {  
    print('Download starting...');  
  
    int totalPercentage = 0;  
    stdout.write('[');  
    while (totalPercentage < 100) {  
      int value = await getChunk();  
      stdout.write('*' * value);  
      totalPercentage += value;  
    }  
    print(']');  
    print('Downloading completed...');  
  } catch (err) {  
    print('Some wrong, ${err.toString()}');  
  }  
}
```

Download starting...

[\*\*\*\*\*]

Download completed...

# Streams

---

- Asynchronous programming in Dart is characterized by the **Future** and **Stream** classes.
- A Future represents a computation that doesn't complete immediately.
- Normal function returns the result, an asynchronous function returns a Future, which will eventually contain the result.
- The future will tell you when the result is ready.

- A stream is a sequence of asynchronous events.
- It is like an asynchronous Iterable—where, instead of getting the next event when you ask for it, the stream tells you that there is an event when it is ready.
- You listen on a stream to get notified of the results (both data and errors) and of the stream shutting down.
- You can also pause while listening or stop listening to the stream before it is complete.
- For example, you are fetching today's news from database and you want get an updated recordset if any latest news added to the database. Stream helps here!

- There are multiple ways to generate the streams, one of which is streams generation from iterables.
- For example you generate a series of numbers as a stream. E.g. [5,4,3,2,1]
- These integers will be now stream of int i.e. **Stream<int>**
- We can now iterate over stream<int> using for loop with or without await, and use the value of each iteration as soon as it is received.

# Streams - Example 1



```
void main() async {  
  final stream = Stream.fromIterable([5, 4, 3, 2, 1]);  
  print(await stream.toList());  
}
```

[5, 4, 3, 2, 1]

# Streams - Example 2



```
void main() async {  
  final stream = Stream.fromIterable([5, 4, 3, 2, 1]);  
  stream.listen((s) => print(s));  
}
```

```
5  
4  
3  
2  
1
```

# Streams - Example 3



```
void main() async {  
  final stream = Stream.fromIterable([5, 4, 3, 2, 1]);  
  final sum = await sumStream(stream);  
  
  print('Total: $sum');  
  print('done');  
}  
  
Future<int> sumStream(Stream<int> stream) async {  
  var sum = 0;  
  
  await for (var value in stream) {  
    await Future.delayed(const Duration(milliseconds: 400));  
    print('value $value');  
    sum += value;  
  }  
  return sum;  
}
```

```
value 5  
value 4  
value 3  
value 2  
value 1  
Total: 15  
done
```



# Generators

---

- The previous example where we generated a int series using Stream Iterables, can be generated also by using Generators.
- By definition,  
**“Dart generator functions are used to generate a sequence of values on-demand lazily.”**
- Dart generator support async generation and sync generation.
- In async generation function returns a Stream object. This is exactly what we did in last example
- In sync generations function returns an Iterable object. That means generation process is already complete but values will be returned lazily on-demand.

# Generators - Sync - Example 1



```
void main() async {  
  final iterableValues = generateList(5);  
  print(iterableValues.toList());  
}  
  
Iterable<int> generateList(int value) sync* {  
  for (int i = value; i > 0; i--) {  
    yield i;  
  }  
}
```

[5, 4, 3, 2, 1]

# Generators - Async - Example 2



```
void main() async {  
  final stream = generateList(5);  
  print(await stream.toList());  
}
```

```
Stream<int> generateList(int value) async* {  
  for (int i = value; i > 0; i--) {  
    await Future.delayed(const Duration(milliseconds: 400));  
    print('i = $i');  
    yield i;  
  }  
}
```

```
i = 5  
i = 4  
i = 3  
i = 2  
i = 1  
[5, 4, 3, 2, 1]
```

# Streams - async - Example 3



```
void main() async {  
  final stream = generateList(5);  
  final sum = await sumStream(stream);  
  
  print('Total: $sum');  
  print('done');  
}  
  
Stream<int> generateList(int value) async* {  
  for (int i = value; i > 0; i--) {  
    await Future.delayed(const Duration(milliseconds: 400));  
    print('i = $i');  
    yield i;  
  }  
}
```

# Streams - async - Example 3



```
Future<int> sumStream(Stream<int> stream) async {  
  var sum = 0;  
  
  await for (var value in stream) {  
    await Future.delayed(const Duration(milliseconds: 400));  
    print('value $value');  
    sum += value;  
  }  
  return sum;  
}
```

```
i = 5  
value 5  
i = 4  
value 4  
i = 3  
value 3  
i = 2  
value 2  
i = 1  
value 1  
Total: 15  
done
```

# Stream constructors

---

# Stream constructors



- We now know how to generate stream using **Stream.fromIterable()**
- Following are other methods that can be used to generate Streams

PROPERTIES	METHODS			
first	any	elementAt	map	toList
hashCode	asBroadcastStream	every	<i>noSuchMethod</i>	toSet
isBroadcast	asyncExpand	expand	pipe	<i>toString</i>
isEmpty	asyncMap	firstWhere	reduce	transform
last	cast	fold	singleWhere	where
length	contains	forEach	skip	
<i>runtimeType</i>	distinct	handleError	skipWhile	
single	drain	join	take	
		lastWhere	takeWhile	
		listen	timeout	



# Stream constructors



<b>fromIterable</b>	Creates a single-subscription stream that gets its data from elements	<b>Stream.fromIterable</b> ([1,2,3,4,5]);
<b>empty</b>	Creates an empty broadcast stream.	<b>Stream.empty</b> ();
<b>error</b>	Creates a stream which emits a single error event before completing.	<b>Stream.error</b> (Error('Error occurred'));
<b>fromFuture</b>	Creates a new single-subscription stream from the future.	<b>Stream.fromFuture</b> ( Future.delayed(Duration(milliseconds: 400), () => 'Hi') );
<b>periodic</b>	Creates a stream that repeatedly emits events at period intervals.	<b>Stream.periodic</b> (milliseconds: 400), (index) => index);
<b>value</b>	Creates a stream which emits a single data event before completing.	<b>Stream.value</b> ('hi');

# Stream Methods / Properties

---

# Stream methods / properties



- Streams comes along various methods and properties

<b>first</b>	The first element of this stream.	<b>Stream.first</b>
<b>isBroadcast</b>	Whether this stream is a broadcast stream.	<b>Stream.isBroadcast</b>
<b>isEmpty</b>	Whether this stream contains any elements.	<b>Stream.isEmpty</b>
<b>last</b>	The last element of this stream.	<b>Stream.last</b>
<b>length</b>	The number of elements in this stream.	<b>Stream.length</b>

# Stream methods / properties



<b>forEach</b>	Executes action on each element of this stream.	<b>Stream.forEach</b> ((value) => print("Hi \$value") );
<b>map</b>	Transforms each element of this stream into a new stream event.	<b>Stream.map</b> ((value)=> value * 2) ;
<b>where</b>	Creates a new stream from this stream that discards some elements.	<b>Stream.where</b> ((value) => value > 5) ;
<b>firstWhere</b>	Finds the first element of this stream matching test.	<b>Stream.firstWhere</b> ((value) => value > 5) ;
<b>reduce</b>	Combines a sequence of values by repeatedly applying combine.	<b>Stream.reduce</b> ((val, elem) => val + elem ) ;

# API Calls

---

# API Calls - Dictionary App



- We will be creating a command line app that takes a word and return synonyms.
- We will use a free REST API from <https://dictionaryapi.dev/>

**`https://api.dictionaryapi.dev/api/v2/entries/en/<word>`**

- We will parse the JSON response from API and print in the following manner

```
> dart run good
```

```
Word: good
```

```
Meaning: to be desired or approved of.
```

```
Sentence: it's good that he's back to his old self
```

# Final Exercise

---

# Exercise - Weather App

- Create a command line app that takes a city name and return it's current weather.
- A Free REST API from **<https://openweathermap.org/current>**
- You will need to register first to get the API key.
- Parse the JSON and print in the following pattern

```
> dart run london
```

```
City: london
```

```
Temperature: 12 °C
```

```
Weather description: haze
```



# Thank you!

What next: Flutter Slides ([Link](#))



[Click for Dart Course Playlist](#)