Exploring Dart

AAIT, ITSC, 2020

Learning outcomes

After completing this lab session you should be able to

Install Dart and configure an IDE to use Dart

Explore Dart features such as

`Objects, Type Inference, Generics, Functions, Variables, Visibility, Identifiers, Expression vs Statements, Final, Const, Integer, Double, Strings, Booleans, List, Set, Map, Anonymous Functions, Closure, Type Cast, Type Test, Conditional Expression, Using Dart Packages`

https://dart.dev/guides/language/language-tour

Installing Dart SDK: Windows

Install using Chocolatey

```
C:\>choco install dart-sdk
```

Install using a setup wizard

Go to the following link to download the setup file and install it using the setup wizard

https://github.com/GeKorm/dart-windows/releases/download/v1.6.0/Dart_x64.stable.setup.exe

Installing Dart SDK: Debian/Ubuntu

Setup

```
$ sudo apt-get update
$ sudo apt-get install apt-transport-https
$ sudo sh -c 'wget -qO-
https://dl-ssl.google.com/linux/linux_signing_key.pub |
apt-key add -'
$ sudo sh -c 'wget -qO-
https://storage.googleapis.com/download.dartlang.org/linux/deb
ian/dart_stable.list >
/etc/apt/sources.list.d/dart_stable.list'
```

Installing Dart SDK: Debian/Ubuntu

Installation

```
$ sudo apt-get update
$ sudo apt-get install dart
```

https://dart.dev/get-dart

Installing Dart SDK: Debian/Ubuntu

Modify PATH for access to all Dart binaries

```
$ export PATH="$PATH:/usr/lib/dart/bin"
```

To change the PATH for future terminal sessions, use a command like this:

```
$ echo 'export PATH="$PATH:/usr/lib/dart/bin"' >> ~/.profile
```

https://dart.dev/get-dart

Installing Dart SDK: Mac

Install Homebrew, and then run:

```
$ brew tap dart-lang/dart
$ brew install dart
```

https://dart.dev/get-dart

A basic Dart program

```dart
// Define a function.
printInteger(int aNumber) {
  print('The number is $aNumber.'); // Print to console.
}

// This is where the app starts executing.
main() {
  var number = 42; // Declare and initialize a variable.
  printInteger(number); // Call a function.
}
```

Variable and Data Type

```
printInteger(int aNumber) {
  print('The number is $aNumber.');
}

main() {
  var number = 42;
  printInteger(number);
}
```

Specifying a type to a variable

Declare a variable without specifying its type

String Literal

```
printInteger(int aNumber) {
  print('The number is $aNumber.');
}
```

A string literal can be specified using single quotes as shown or using double quotes as well

String interpolation

```
printInteger(int aNumber) {
  print('The number is $aNumber.');
}
```

Allows you to include a variable or expression's string equivalent inside of a string literal

```
printInteger(int aNumber) {
  print('The number is ${aNumber}.');
}
```

main()

The special, **required**, **top-level function** where **app execution starts**

```
main() {
  var number = 42;
  printInteger(number);
}
```

Key points about Dart: Objects

Everything you can place in a variable is an object, and every object is an instance of a class

Even numbers, functions, and null are objects

All objects inherit from the Object class

Key points about Dart: Type Inference

Although Dart is strongly typed, type annotations are optional because **Dart can infer types**

```
main() {
  var number = 42;
  printInteger(number);
}
```

The type of number is inferred from its value

Key points about Dart: Generics

Dart supports generic types, such as

`List<int>` (a list of integers)

`List<dynamic>` (a list of objects of any type)

Key points about Dart: Functions

Dart supports

- **Top-level** functions
- Functions tied to a class (**static methods**)
- Functions tied to an object (**instance methods**)

Top-level functions

```dart
// file: print_number.dart

printInteger(int aNumber) {
  print('The number is $aNumber.');
}

main() {
  var number = 42;
  printInteger(number);
}
```

Both printInteger and main functions are top-level functions inside the print_number.dart file

Exercise

Create a dart file named **`example.dart`**

Create a top-level function named **`sum`**, which accepts two numeric arguments and returns the sum of the two numbers

Create a top-level **`main`** function inside it

- Declare two variables with values **`3.0`** and **`7.0`**
- Call the sum function by passing the two variables
- Print the sum of the two numbers as follows

  `"The sum of 3.0 and 7.0 is 10.0"`

Exercise

Use the following command to compile and run the file

```
    dart path/to/example.dart
```

Key points about Dart: Variables

Dart supports

- **Top-level** variables,
- Variables tied to a class (**static variables**)
- Variables tied to an object (**instance variables**)

Instance variables are sometimes known as **fields** or **properties**.

Key points about Dart: Visibility

If an identifier starts with an **underscore** (_), it’s **private** to its library

Key points about Dart: Identifiers

Identifiers can start with a **letter** or **underscore** (_), **followed by any combination of those characters** plus **digits**

Key points about Dart: Expressions and Statements

Dart has both **expressions** (which have runtime values) and **statements** (which don’t have runtime values)

Example conditional expression

```
    var x = 2 > 3 ? true : false;
```

`if-else` statement has no value

A statement often contains one or more expressions, but an expression can’t directly contain a statement

Key points about Dart: Warnings and Errors

Warnings are just indications that your code might not work, but they don't prevent your program from executing

Errors can be either compile-time or run-time

> A **compile-time error** prevents the code from executing at all
>
> A **run-time error** results in an exception being raised while the code executes

Variables

Creating a variable and initializing it

```
    var name = 'Bob';
```

Variables store references

The variable called `name` contains a **reference to a `String` object** with a value of **“Bob”**

The type of the `name` variable is **inferred** to be `String`

Variables

If an object isn’t restricted to a single type, specify the `Object` or `dynamic` type

Experiment the difference between the following declarations

```
var name = 'Bob';

dynamic name = 'Bob';

Object name = 'Bob';

String name = 'Bob';
```

Variables: Default value

Uninitialized variables have an initial value of null

Even variables with numeric types are initially null, because numbers—like everything else in Dart—are objects

Check if this default value of uninitialized variables are null

Define a variable of any type and check if its initial value is null

Variables: `final` and `const`

A `final` variable can be set only once

A `const` variable is a **compile-time constant**

`const` variables are implicitly `final`

A final top-level or class variable is initialized the first time it’s used

Variables: `final` and `const`

Instance variables can be `final` but not `const`

`final` instance variables **must be initialized** before the constructor body starts — **at the variable declaration**, **by a constructor parameter**, or **in the constructor's initializer list**

If the `const` variable is at the **class level**, mark it `static const`

Variables: `final` and `const`

When you declare a `const` variable, you should set the value to **a compile-time constant** such as a **number** or **string literal**, a **const variable**, or **the result of an arithmetic operation on constant numbers**

```
const kibibyte = 1024;  // 1 Kibibyte(KiB) = 1024 bytes

const mebibyte = 1024 * kibibyte; // Mebibyte(MiB) = 1024
```

Variables: `final` and `const`

You can also create **constant values**

You can also declare constructors that create constant values

Any variable can have a constant value

```
var x  = const [];
final y = const [];
const z = []; // Equivalent to `const []
```

Variables: `final` and `const`

Experiment on the following cases

1. To which one of the following **array/list** you can add additional element

```
var x  = const [];
const x  = const [];
final y= [];
const z = [];
```

2. What are the difference among them?

`You can use the add() method of a list to add an element to it`

Variables: `final` and `const`

You can change the value of a non-final, non-const variable, **even if it used to have a const value**

You can't change the value of a const variable

Which one of the following is possible? The left or the right?

```
var x = const [1,2,3];
x.add(4);
```

```
var x = const [1,2,3];
 x = [1,2,3];
 x.add(4);
```

Variables: `const`

As of Dart 2.5, you can define constants that use type checks and casts (`is` and `as`), collection `if`, and spread operators (`...` and `...?`)

```
const Object i = 3;
const aList = [i as int, 4, 5, 6]; //typecast
const aMap = {if (i is int) i: "int"}; //is and collection if
const aSet = {if (aList is List<int>) ...aList}; //a spread
```

`Try to print the value of each of the variables i, aList, aMap, and aSet`

Built-in types: Numbers

`int`

integer values no larger than 64 bits

`double`

64-bit (double-precision) floating-point numbers

Both int and double are subtypes of num

Check if `i` in the following code is a `num` or not

```
int i = 2;
```

Built-in types: Numbers

Examples of defining integer literals

```
int x = 1;

int hex = 0xDEADBEEF;
```

Examples of defining double literals

```
var y = 1.1;

var exponents = 1.42e5;
```

Built-in types: Numbers

Converting **`String`** to **`int`/`double`**

```
var one = int.parse('1');
var onePointOne = double.parse('1.1');
```

Converting **`int`/`double`** to **`String`**

```
String oneAsString = 1.toString();

String piAsString = 3.14159.toStringAsFixed(2);
```

Built-in types: Strings

A Dart string is a sequence of UTF-16 code units

You can use either single or double quotes to create a string:

```
var s1 = 'Single quotes work well for string literals.';
var s2 = "Double quotes work just as well.";
var s3 = 'It\'s easy to escape the string delimiter.';
var s4 = "It's even easier to use the other delimiter.";
```

Built-in types: Strings

You can put the value of an expression inside a string by using **`${expression}`**

If the expression is an identifier, you can skip the **`{}`**

To get the string corresponding to an object, Dart calls the object’s **`toString()`** method

Two strings are equivalent if they contain the same sequence of code units

Exercise

Write a program which prints the contents of the variable `s` as shown, using the string interpolation feature. You should use the `toUpperCase()` method of the string object to convert the contents of `s` to all caps.

```
var s = "Interpolation";
```

Expected output:

```
Dart has string INTERPOLATION feature
```

Built-in types: Strings

You can concatenate strings using **adjacent string literals** or the **+** operator

```
var s1 = 'String '
         'concatenation'
         " works even over line breaks.";


var s2 = 'String ' + 'concatenation' +

         " using another approach.";
```

Built-in types: Strings

Using a **triple quote** with either **single** or **double quotation marks** to create **multi-line string**

```
var s1 = '''
You can create
multi-line strings like this one.
''';

var s2 = """This is also a
multi-line string.""";
```

Built-in types: Booleans

`bool`

Only two objects have type `bool`: the boolean literals `true` and `false`, which are both compile-time constants.

Built-in types: Lists

In Dart, arrays are `List` objects, ordered group of objects

```
    var list = [1, 2, 3];
```

Lists use **zero-based indexing**, where `0` is the index of the first element and `list.length - 1` is the index of the last element

Write a program which declare a list of 10 integers and then print the first and the last elements.

Built-in types: Lists

To create a list that’s a compile-time constant, add **const** before the list literal

```
    var constantList = const [1, 2, 3];
```

What will happen if you try to do as follows?

```
        constantList.add(7);
```

Built-in types: Lists

Spread operator (`...`)

you can use the spread operator (`...`) to insert all the elements of a list into another list

```
var list = [1, 2, 3];

var list2 = [0, ...list];
```

Try to print the length or content of `list2`

Built-in types: Lists

Null-aware spread operator (`...?`)

If the expression to the right of the spread operator might be null, you can avoid exceptions by using a null-aware spread operator

```
var list;

var list2 = [0, ...list];
```

What will happen if you try to execute the above code? Check it

Built-in types: Lists

Null-aware spread operator (`...?`)

If the expression to the right of the spread operator might be null, you can avoid exceptions by using a null-aware spread operator

```
var list;

var list2 = [0, ...?list];
```

Add the null-aware spread operator as shown above and check if the exception go away

Built-in types: Lists

collection `if`

you can use it to build collections using conditionals

```
var nav = [
  'Home',
  'Furniture',
  'Plants',
  if (promoActive) 'Outlet'
];
```

What do you think is the length of the `nav` list?

Built-in types: Lists

collection for

you can use it to build collections using iteration

```
var listOfInts = [1, 2, 3];
var listOfStrings = [
    '#0', for (var i in listOfInts) '#$i'
];
```

Print `listOfStrings` and check the output

Built-in types: Sets

A set in Dart is an unordered collection of unique items

Dart support for sets is provided by set literals and the `Set` type

```
var weekDays = {'Monday', 'Tuesday', 'Wednesday',
'Thursday', 'Friday'};
```

To create an empty set, use `{}` preceded by a type argument, or assign `{}` to a variable of type `Set`

```
var firstNames = <String>{};
```

Built-in types: Sets

To create an empty set, use `{}` preceded by a type argument, or assign `{}` to a variable of type `Set`

```
var departments = <String>{};
Set<String> courses = {};
var names = {}; // Creates a map, not a set.
```

Check in code if the last line creates a map not a set

You can use `is` or `runtimeType` field of the types to check their type

Built-in types: Sets

You can add items to an existing set using the `add()` or `addAll()` methods

Write a program which define one set with four elements and another one with two elements. Add the first set element to the second using `add()` or `addAll()` methods

Use `.length` to get the number of items in the set

Print the length of your sets using `.length`

Built-in types: Sets

To create a set that's a **compile-time constant**, add `const` before the set literal

```
var weekDays = const {'Monday', 'Tuesday', 'Wednesday',
'Thursday', 'Friday'};
```

Like lists sets support

spread operators (`...` and `...?`) and

collection `ifs` and `fors`

Built-in types: Maps

A map is an object that **associates 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

Dart support for maps is provided by **map literals** and the **`Map` type**

Built-in types: Maps

```
var weekDays1 = {
    'first': 'Monday',
    'second': 'Tuesday',
    'third': 'Wednesday',
    'fourth': 'Thursday',
    'fifth': 'Friday'
 };
```

```
var weekDays2 = {
    1: 'Monday',
    2: 'Tuesday',
    3: 'Wednesday',
    4: 'Thursday',
    5: 'Friday'
 };
```

Write a program that print the second element from each map

Built-in types: Maps

```
var weekDays = {};
weekDays['first'] = 'Monday';
weekDays[2] = 'Tuesday';
weekDays[3] = 3;
```

Do you think this code is correct? If yes, check its `runtimeType` and justify your answer and restrict both the keys and values only to `String` type

Built-in types: Maps

Use `.length` to get the number of key-value pairs in the map

To create a map that’s a **compile-time constant**, add `const` before the map literal

```
var weekDays = const {
    1: 'Monday',
    2: 'Tuesday',
    3: 'Wednesday',
    4: 'Thursday',
    5: 'Friday'
 };
```

Built-in types: Maps

Like lists maps support

- **spread operators** (`...` and `...?`) and
- **collection** `if`s and `for`s

Functions

Dart is a true object-oriented language, so even **functions are objects** and have a type, **`Function`**

This means that **functions can be assigned to variables** or **passed as arguments to other functions**

You can also call an instance of a Dart class as if it were a function

Functions

Example

```
bool isEven(int i) {
 return i % 2 == 0;
}
```

Though not recommended, you can omit the types

```
isEven(i) {
 return i % 2 == 0;
}
```

Functions

The `=> expr` syntax is a shorthand for `{ return expr; }`

The `=>` notation is sometimes referred to as **arrow syntax**

The previous example can be re-written as follows

```
    bool isEven(int i) => i%2 == 0;
```

Only an **expression—not a statement**—can appear between the arrow (`=>`) and the semicolon (`;`).

Functions

A function can have two types of parameters: **required** and **optional**

The required parameters are listed first, followed by any optional parameters

Optional parameters can be **named** or **positional** but not both

Flutter widget constructors use only named parameters, even for parameters that are mandatory

Functions

Defining named parameters

When defining a function, use `{param1, param2, …}` to specify named parameters

```
bool boldFlag = false;
bool hiddenFlag = false;

void enableFlags({bool bold, bool hidden}) {
    boldFlag = bold;
    hiddenFlag = hidden;
}
```

Functions

Using named parameters

When calling a function, you can specify named parameters using `paramName: value`

```
main() {
 enableFlags(bold:true);
}
```

```
main() {
 enableFlags(hidden: false, bold:true);
}
```

Functions

Named parameters

Although named parameters are a kind of optional parameter, you can annotate them with `@required` to indicate that the parameter is mandatory — **that users must provide a value for the parameter**

```
void enableFlags({bool bold, @required bool hidden}) {
    boldFlag = bold;
    hiddenFlag = hidden;
}
```

Functions

Named parameters

To use the `@required` annotation, you need to import the **`meta`** package as shown below

```
import ‘package:meta/meta.dart’;
```

To know how to import packages check the **Import Dart Packages** slide

Functions

Positional parameters

Wrapping a set of function parameters in `[]` marks them as optional positional parameters

```
String say(String from, String msg, [String device]) {
  var result = '$from says $msg';
  if (device != null) {
    result = '$result with a $device';
  }
  return result;
}
```

Functions

Positional parameters

Here’s an example of calling this function without the optional parameter

```
say('Mr. X', 'Hello')
```

And here’s an example of calling this function with the third parameter

```
say('Mr. X', 'Hello', 'Telephone')
```

Functions

Default parameter values

Your function can use `=` to define default values for both named and positional parameters

The **default values must be compile-time constants**

If no default value is provided, the default value is null

Functions

Default parameter values

Your function can use = to define default values for both named and positional parameters

```
bool boldFlag;
bool hiddenFlag;
void enableFlags({bool bold = false, @required bool hidden})
{
     boldFlag = bold;
     hiddenFlag = hidden;
}
```

Functions

Default parameter values

```
String say(String from, String msg, [String device = 'radio'])
{
    var result = '$from says $msg';
    if (device != null) {
      result = '$result with a $device';
    }
    return result;
}
```

The `main()` function

Every app must have a top-level **`main()`** function, which serves as the entrypoint to the app

The **`main()`** function returns **`void`** and has an optional **`List<String>`** parameter for arguments

Write a program which accept two numbers as command line argument and returns their sum

Functions as first-class objects

You can pass a function as a parameter to another function

```
void printElement(int element) {

  print(element*element);

}

var list = [1, 2, 3];

// Pass printElement as a parameter
list.forEach(printElement);
```

Functions as first-class objects

Write a function which accept any function and execute/call the accepted/passed function

Try the function you have created with another function which print the word `‘Hello’`

Functions as first-class objects

You can also assign a function to a variable

```
var makeAllCaps = (word) => word.toUpperCase();

void main() {
 print(makeAllCaps('hello'));
}
```

`makeAllCaps` variable is assigned anonymous function

Anonymous functions

You can create a nameless function called an **anonymous function**, or sometimes a **lambda** or **closure**

You might assign an anonymous function to a variable

An anonymous function looks similar to a named function— zero or more parameters, separated by commas and optional type annotations, between parentheses

```
([[Type] param1[, …]]) {
    codeBlock;
};
```

Anonymous functions

```
var list = ['apples', 'bananas', 'oranges'];

list.forEach((item) {

  print('${list.indexOf(item)}: $item');

});
```

Rewrite the above code using the arrow (=>) syntax

Functions: Lexical scope

Dart is a lexically scoped language, which means that 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

Functions: Lexical scope

```
bool topLevel = true;
void main() {
 var insideMain = true;
 void myFunction() {
   var insideFunction = true;
   void nestedFunction() {
     var insideNestedFunction = true;
     print(topLevel);
     print(insideMain);
     print(insideFunction);
     print(insideNestedFunction);
   }
 }
}
```

Functions: Lexical closures

A **closure** is a function object that has access to variables in its lexical scope, even when the function is used outside of its original scope

Functions can close over variables defined in surrounding scopes

Functions: Lexical closures

```
/// Returns a function that adds n to the function's argument
Function makeAdder(num n) {
 return (num i) => n + i;
}

void main() {
 var add2 = makeAdder(2);  // Create a function that adds 2
 var add4 = makeAdder(4);  // Create a function that adds 4
 print(add2(3));
 print(add4(3));
}
```

Functions: Return values

All functions return a value

If no return value is specified, the statement `return null;` is implicitly appended to the function body

Type test operators

Operator	Meaning
`as`	Typecast
`is`	True if the object has the specified type
`is!`	False if the object has the specified type

The result of `obj is T` is true if `obj` implements the interface specified by `T`

For example, `obj is Object` is always true

Type test operators

Use the `as` operator to cast an object to a particular type **if and only if** you are sure that the object is of that type

```
(emp as Person).firstName = 'Mr X';
```

If you aren’t sure that the object is of type `T`, then use `is T` to check the type before using the object

```
if (emp is Person) {
  // Type check
  emp.firstName = 'Mr X';
}
```

Assignment operators

You can assign values using the `=` operator

To assign only if the assigned-to variable is `null`, use the `??=` operator

```
// Assign value to a

a = value;

// Assign value to b if b is null;
// otherwise, b stays the same
b ??= value;
```

Conditional expressions

If ***`condition`*** is true, evaluates ***`expr1`*** (and returns its value); otherwise, evaluates and returns the value of ***`expr2`***

```
condition ? expr1 : expr2
```

If ***`expr1`*** is non-null, returns its value; otherwise, evaluates and returns the value of ***`expr2`***

```
expr1 ?? expr2
```

Conditional expressions

When you need to assign a value based on a boolean expression, consider using **`?:`**

If the boolean expression tests for null, consider using **`??`**

Control flow statements

`if` and `else`

`for` loops

`while` and `do-while` loops

`break` and `continue`

`switch` and `case`

Control flow statements: `Exercise`

Write a program which

- Declare list of 10 integers
- Separate even integers from odd integers and place them in separate lists

Write different version of the above program using `for`, `for-in`, `forEach` **method** of a list, `while`, `do-while` loops for iterating

Control flow statements: `Exercise`

Write a program which

- Declare list of 10 integers
- Print only the even numbers

Use `continue` statement to skip the odd numbers

Control flow statements: `Exercise`

Write a program which

Declare a list with the first 7 positive numbers

Print week days corresponding to each entry of the list

```
1 -> Monday, 2 -> Tuesday, ... , 7 -> Sunday
```

Use **`switch/case`** statement to complete this exercise

Assert

During development, use an assert statement — `assert(condition, optionalMessage);` — to disrupt normal execution if a boolean condition is false

```
// Make sure the variable has a non-null value.
assert(text != null);

// Make sure the value is less than 100.
assert(number < 100);
```

Assert

To attach a message to an assertion, add a string as the second argument to assert

```
assert(urlString.startsWith('https'),
      'URL ($urlString) should start with "https".');
```

You should supply a command-line flag: **`--enable-asserts`** to **`dart`** command when you execute your file

Exceptions

All of Dart’s exceptions are unchecked exceptions

Methods do not declare which exceptions they might throw, and you are not required to catch any exceptions

Dart provides `Exception` and `Error` types, as well as numerous predefined subtypes

You can define your own exceptions

Dart programs can throw any non-null object—not just `Exception` and `Error` objects—as an exception

Exceptions: Throw

Here’s an example of throwing, or raising, an exception

```
    throw FormatException('Expected at least 1 section');
```

You can also throw arbitrary objects

```
    throw 'Out of llamas!';
```

Production-quality code usually throws types that implement **`Error`** or **`Exception`**

Exceptions: Throw

Because throwing an exception is an expression, you can throw exceptions in => statements, as well as anywhere else that allows expressions

```
void distanceTo(Point other) => throw UnimplementedError();
```

Exceptions: Catch

Catching, or capturing, an exception stops the exception from propagating (unless you rethrow the exception)

```
void main() {
 try {
   throw StateError('Thrown exception');
 } on StateError {
   print("Bad State exception has been thrown");
 }
}
```

Exceptions: Catch

To handle code that can throw more than one type of exception, you can specify multiple catch clauses

The first catch clause that matches the thrown object’s type handles the exception

If the catch clause does not specify a type, that clause can handle any type of thrown object

Exceptions: Catch

```
void main() {
 int x, y, z;
 try {
   z = x + y;
 } on NullThrownError { // A specific exception
   print('Null exception');
 } on Exception catch (e) { // Anything else that is an exception
   print('Unknown exception: $e');
 } catch (e) { // No specified type, handles all
   print('Something really unknown: $e');
 }
}
```

Exceptions: Catch

As the preceding code shows, you can use either on or catch or both

Use on when you need to specify the exception type

Use catch when your exception handler needs the exception object

```
try {
  // ···
} on Exception catch (e) {
  print('Exception details:\n $e');
} catch (e, s) {
  print('Exception details:\n $e');
  print('Stack trace:\n $s');
}
```

Exceptions: rethrow

To partially handle an exception, while allowing it to propagate, use the `rethrow` keyword

```
void misbehave() {
 try {
   dynamic x = true;
   print(x++); // Runtime error
 } catch (e) {
   print('misbehave() partially handled ${e.runtimeType}.');
   rethrow; // Allow callers to see the exception.
 }
}
```

Exceptions: Catch

Call the **`misbehave`** function in the previous slide and print the rethrown exception

Exceptions: Finally

To ensure that some code runs whether or not an exception is thrown, use a finally clause

If no catch clause matches the exception, the exception is propagated after the finally clause runs

```
try {
    // some code
} finally {
    // Always clean up, even if an exception is thrown.
}
```

Exceptions: Finally

The **`finally`** clause runs after any matching **`catch`** clauses

```
try {
   // some code
} catch (e) {
   print('Error: $e');
} finally {
   // Always clean up, even if an exception is thrown.
}
```

Classes

Dart is an object-oriented language with `classes` and `mixin-based` inheritance

Every object is an instance of a `class`, and all classes descend from `Object`

`Mixin-based` inheritance means that although every class (except for Object) has exactly one superclass, **a class body can be reused in multiple class hierarchies**

Extension methods are a way to add functionality to a class without changing the class or creating a subclass

Classes: Using class members

Objects have members consisting of functions (methods) and data (instance variables)

When you call a method, you invoke it on an object: the method has access to that object’s functions and data

Use a dot (`.`) to refer to an instance variable or method

All instance variables generate an **implicit getter method**

Non-final instance variables also generate an **implicit setter method**

Classes: Example

```
class Point {
 num x;
 num y;

 Point(this.x, this.y);

 num distanceTo(Point p) {
   return sqrt(pow((x - p.x), 2) + pow((y - p.y), 2));
 }
 @override
 String toString() {
   return "(${this.x}, ${this.y})";
 }
}
```

Classes: Example

```
class Point {
 num x;
 num y;

 Point(this.x, this.y);

 num distanceTo(Point p) {
   return sqrt(pow((x - p.x), 2) + pow((y - p.y), 2));
 }
 @override
 String toString() {
   return "(${this.x}, ${this.y})";
 }
}
```

Instance variables/fields/properties

Constructor

Instance Methods

Classes: Using class members

```
void main() {
 var p1 = Point(2, 2);
 var p2 = Point(4, 4);
 num distance = p1.distanceTo(p2);
 print("Distance b/n P1 = ${p1} & P2 = ${p2} is $distance");
 p1.x = -2;
 p1.y = -3;
 distance = p1.distanceTo(p2);
 print("Distance b/n P1 = ${p1} & P2 = ${p2} is $distance");
}
```

Classes: Using class members

Use `?.` instead of `.` to avoid an exception when the leftmost operand is null

What will happen if you run the following code?

```
void main() {
 Point p;
 p.x = 3;
 p.y = 4;
 print(p);
}
```

Classes: Using class members

Use `?.` instead of `.` to avoid an exception when the leftmost operand is null

What will happen if you run the following code?

```
void main() {
 Point p;
 p.x = 3;   | Exeption will be thrown here
 p.y = 4;   |
 print(p);
}
```

Classes: Using class members

Use `?.` instead of `.` to avoid an exception when the leftmost operand is null

What will happen if you run the following code?

```
void main() {
 Point p;
 p?.x = 3;|
          | No Exeption if you use ?.
 p?.y = 4;|
 print(p);
}
```

Classes: Using constructors

You can create an object using a `constructor`

Constructor names can be of the form `ClassName` or `ClassName.identifier`

Classes: Using constructors

```
class Point {
 num x;
 num y;

 Point(this.x, this.y);
 Point.fromMap(Map<String, num> m){
   this.x = m['x'];
   this.y = m['y'];
 }
}
```

The `this` keyword refers to the current instance

Classes: Using constructors

```
class Point {
 num x;
 num y;

 Point(this.x, this.y);
 Point.fromMap(Map<String, num> m){
   this.x = m['x'];
   this.y = m['y'];
 }                       Named constructor
}
```

Classes: Using constructors

```
void main() {
 Point p0 = Point(0,0);
 Point p1 = Point.fromMap({'x': 7, 'y': 2});
 print(p0);
 print(p1);
}
```

Classes: Using constructors

Some classes provide constant constructors

```
class Point {
 final num x;
 final num y;
 const Point(this.x, this.y);
}
```

Note: The fields are final

Classes: Using constructors

To create a compile-time constant using a constant constructor, put the `const` keyword before the constructor name

```
Point p = const Point(0,0);
```

Classes: Getting an object’s type

To get an object’s type at runtime, you can use Object’s `runtimeType` property, which returns a `Type` object

```
void main() {
 var p = const Point(0,0);
 print(p.runtimeType);
}
```

Classes: Default constructors

If you don’t declare a constructor, a default constructor is provided for you

The default constructor has no arguments and invokes the no-argument constructor in the superclass

Classes: Constructors aren’t inherited

Subclasses don’t inherit constructors from their superclass

A subclass that declares no constructors has only the default (no argument, no name) constructor

Classes: Invoking a superclass constructor

By default, a constructor in a subclass calls the superclass's unnamed, no-argument constructor

If the superclass doesn't have an unnamed, no-argument constructor, then you must manually call one of the constructors in the superclass

Specify the superclass constructor after a colon (`:`), just before the constructor body (if any)

```
class Employee extends Person {
  Employee() : super.fromJson(defaultData);
  // ···
}
```

Classes: Initializer list

Besides invoking a superclass constructor, you can also initialize instance variables before the constructor body runs

Separate initializers with commas

Classes: Initializer list

```
class Point {
 num x;
 num y;
 Point.fromMap(Map<String, num> m)
      : x = m['x'],
        y = m['y'] {
   print('In Point.fromMap(): ($x, $y)');
 }
}
```

Classes: Redirecting constructors

Sometimes a constructor’s only purpose is to redirect to another constructor in the same class

A redirecting constructor’s body is empty, with the constructor call appearing after a colon (`:`)

Classes: Redirecting constructors

```
class Point {
 num x, y;
 // The main constructor for this class.
 Point(this.x, this.y);
 // Delegates to the main constructor.
 Point.alongXAxis(num x) : this(x, 0);
}
```

Classes: Factory constructors

Use the factory keyword when implementing a constructor that doesn’t always create a new instance of its class

For example, a factory constructor might return an instance from a cache, or it might return an instance of a subtype

Classes: Factory constructors

```
class Logger {
 final String name;
 bool mute = false;
 static final Map<String, Logger> _cache = <String, Logger>{};
 factory Logger(String name) {
   return _cache.putIfAbsent(name, () => Logger._internal(name));
 }
 Logger._internal(this.name);
 void log(String msg) {
   if (!mute) print(msg);
 }
}
```

Usage

```
void main() {
 var logger = Logger('UI');
 logger.log('Button clicked');
}
```

Classes: Getters and setters

Getters and setters are special methods that provide read and write access to an object's properties

Instance variable has an implicit getter, plus a setter if appropriate

You can create additional properties by implementing **getters** and **setters**, using the `get` and `set` keywords

Classes: Getters and setters

```
class Rectangle {
 num left, top, width, height;
 Rectangle(this.left, this.top, this.width, this.height);
 // Define two calculated properties: right and bottom.
 num get right => left + width;
 set right(num value) => left = value - width;
 num get bottom => top + height;
 set bottom(num value) => top = value - height;
}
```

Classes: Getters and setters

```
void main() {
  var rect = Rectangle(3, 4, 20, 15);
  assert(rect.left == 3);
  rect.right = 12;
  assert(rect.left == -8);
}
```

Classes: Getters and setters

With getters and setters, you can start with instance variables, later wrapping them with methods, all without changing client code

Write a class to represent a circle with a given radius value and its area. The area should be computed with a getter method. There should be a setter for the radius. The setter make sure that the radius is never less than 0.

Abstract classes

```
abstract class Shape {
 num area();
}

class Circle extends Shape {
 final num radius;
 Circle(this.radius);
 num area() {
   return pi * radius * radius;
 }
}
```

Usage

```
void main() {
 Circle c = Circle(2.3);
 print(c.area());
}
```

Abstract methods

Instance, getter, and setter methods can be abstract, defining an interface but leaving its implementation up to other classes

Abstract methods can only exist in abstract classes

Implicit interfaces

Every class implicitly defines an interface containing all the instance members of the class and of any interfaces it implements

If you want to create a class **`A`** that supports class **`B`**'s API without inheriting **`B`**'s implementation, class **`A`** should implement the **`B`** interface

A class implements one or more interfaces by declaring them in an **`implements`** clause and then providing the APIs required by the interfaces

Implicit interfaces

```
// A person. The implicit interface contains greet()
class Person {
 final _name; // In the interface
 Person(this._name); // Not in the interface
 String greet(String who) => 'Hello, $who. I am $_name.';
}

// An implementation of the Person interface
class Impostor implements Person {
 get _name => '';
 String greet(String who) => 'Hi $who. Do you know who I am?';
}

String greetBob(Person person) => person.greet('Bob');
```

Implicit interfaces

```
String greetBob(Person person) => person.greet('Bob');

void main() {
 print(greetBob(Person('Kathy')));
 print(greetBob(Impostor()));
}
```

Example of specifying that a class implements multiple interfaces

```
    class Point implements Comparable, Location {...}
```

Extending a class

Use **`extends`** to create a subclass, and **`super`** to refer to the superclass

```
class Television {
  void turnOn() {
    _illuminateDisplay();
    _activateIrSensor();
  }
  // ...
}
```

```
class SmartTelevision extends Television {
  void turnOn() {
    super.turnOn();
    _bootNetworkInterface();
    _initializeMemory();
    _upgradeApps();
  }
  // ...
}
```

Overriding members

Subclasses can override instance methods, getters, and setters

You can use the `@override` annotation to indicate that you are intentionally overriding a member

```
class SmartTelevision extends Television {
  @override
  void turnOn() {...}
  // ···
}
```

Overridable operators

You can override the operators shown below

```
<       +       |       [ ]     >       /       ^

[ ]     =       <=      ~/      &       ~       >>

>=      *       <<      ==      -       %
```

Overridable operators

For example, if you define a Vector class, you might define a + method to add two vectors

```
class Vector {
 final int x, y;
 Vector(this.x, this.y);
 Vector operator +(Vector v) => Vector(x + v.x, y + v.y);
 Vector operator -(Vector v) => Vector(x - v.x, y - v.y);
}
```

Overridable operators

For example, if you define a Vector class, you might define a + method to add two vectors

```dart
void main() {
 final v = Vector(2, 3);
 final w = Vector(2, 2);
 assert(v + w == Vector(4, 5));
 assert(v - w == Vector(0, 1));
}
```

Overridable operators

If you override `==`, you should also override Object's `hashCode` getter

Extension methods

Extension methods are a way to add functionality to existing libraries

Enumerated types

Often called enumerations or enums, are a special kind of class used to represent a fixed number of constant values

```
    enum Color { red, green, blue }
```

Each value in an enum has an index getter, which returns the zero-based position of the value in the enum declaration

```
    assert(Color.red.index == 0);

    assert(Color.green.index == 1);

    assert(Color.blue.index == 2);
```

Enumerated types

To get a list of all of the values in the **`enum`**, use the **`enum`**'s **`values`** constant

```
List<Color> colors = Color.values;

assert(colors[2] == Color.blue);
```

Enumerated types

You can use **`enums`** in **`switch`** statements, and you'll get a warning if you don't handle all of the enum's values

```
var aColor = Color.blue;

switch (aColor) {
  case Color.red:
    print('Red as roses!');
    break;
  case Color.green:
    print('Green as grass!');
    break;
  default: // Without this, you see a WARNING.
    print(aColor); // 'Color.blue'
}
```

Enumerated types

Enumerated types have the following limits

- You can’t subclass, mix in, or implement an enum
- You can’t explicitly instantiate an enum

mixins

Allows you to add features to a class

Mixins are a way of reusing a class's code in multiple class hierarchies

Use the `mixin` keyword to define mixins

If you want your mixin to be usable as a regular class, you can use the `class` keyword instead of `mixin`

To use a mixin, use the `with` keyword followed by one or more mixin names

mixins

```
mixin Sound {
 var loudness;
}


mixin Walk {
 var walkSpeed;
}
```

```
class Duck with Sound, Walk {
 @override
 String toString() {
    return "$loudness dB, $walkSpeed km/hr,";
 }
}
```

Class variables and methods

Use the `static` keyword to implement class-wide variables and methods

Static variables (class variables) are useful for class-wide state and constants

```
class Queue {
  static const capacity = 16;
  // ···
}
```

```
void main() {
  assert(Queue.capacity == 16);
}
```

Class variables and methods

Static methods (class methods) do not operate on an instance, and thus do not have access to this

```
class Point {
 num x, y;
 Point(this.x, this.y);
 static num distanceBetween(Point a, Point b) {
   var dx = a.x - b.x;
   var dy = a.y - b.y;
   return sqrt(dx * dx + dy * dy);
 }
}
```

Class variables and methods

Static methods (class methods) do not operate on an instance, and thus do not have access to this

```
void main() {
 var a = Point(2, 2);
 var b = Point(4, 4);
 var distance = Point.distanceBetween(a, b);
 assert(2.8 < distance && distance < 2.9);
 print(distance);
}
```

Generics

Example: **`List<E>`**

The **<…>** notation marks `List` as a generic (or parameterized) type—a type that has formal type parameters

By convention, most type variables have single-letter names, such as **`E`**, **`T`**, **`S`**, **`K`**, and **`V`**

`List`, **`set`**, and **`map`** literals can be parameterized

You add **`<type>`** for lists and sets or **`<keyType, valueType>`** for maps before the opening bracket

Generics

```
var names = <String>['Seth', 'Kathy', 'Lars'];
var uniqueNames = <String>{'Seth', 'Kathy', 'Lars'};
var pages = <String, String>{
  'index.html': 'Homepage',
  'robots.txt': 'Hints for web robots',
  'humans.txt': 'We are people, not machines'
};
```

Libraries and visibility

The **`import`** and **`library`** directives can help you create a modular and shareable code base

Libraries not only provide APIs, but are a unit of privacy: identifiers that start with an underscore (`_`) are visible only inside the library

Every Dart app is a library, even if it doesn't use a library directive

Libraries can be distributed using packages

Using libraries

Use **`import`** to specify how a namespace from one library is used in the scope of another library

```
import 'dart:math';
```

The only required argument to import is a URI specifying the library

For built-in libraries, the URI has the special **`dart:`** scheme

For other libraries, you can use a file system path or the **`package:`** scheme

The **`package:`** scheme specifies libraries provided by a package manager such as the pub tool

Specifying a library prefix

If you import two libraries that have conflicting identifiers, then you can specify a prefix for one or both libraries

For example, if library1 and library2 both have an Element class, then you might have code like this

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

Importing only part of a library

If you want to use only part of a library, you can selectively import the library

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

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

Using Dart Packages

You should first go to **`https://pub.dev/packages/`** to search for and to get information about the package you want to import

You need to create a **`pubspec.yaml`** file to specify the dependencies at the top directory of your application

Example **`pubspec.yaml`** file

```
name: example
  dependencies:
       meta: ^1.1.8
```

Using Dart Packages

You can get installation information and the version number from the `https://pub.dev/packages/`

Once you have a `pubspec`, you can run `pub get` command from the top directory of your application to get the packages