ESCUELA DE SISTEMAS Y TECNOLOGÍAS

Transparencias de **ANALISTA DE SISTEMAS**
Edición 2023 - Materia: **Aplicaciones Móviles con Flutter**

TEMA: Lenguaje Dart

Consideraciones

- Estas transparencias no tienen el objetivo de suplir las clases.
- Por tanto, serán complementadas con ejemplos, códigos, profundizaciones y comentarios por parte del docente.
- El orden de dictado de estos temas está sujeto a la consideración del docente.
- Lo que sigue es un resumen de los elementos más comúnmente utilizados del lenguaje Dart, tomando como base los conocimientos ya adquiridos del lenguaje Java.

Referencias

- Sitio oficial de Dart:
  - https://dart.dev/
- Documentación oficial de Dart:
  - https://dart.dev/guides
- Tour del lenguaje Dart:
  - https://dart.dev/guides/language/language-tour

Agenda

- El Lenguaje Dart
- Null Safety
- La Función main()
- Variables
- Strings
- Lists
- Sets
- Maps
- Fechas y Horas
- Funciones
- Excepciones
- Alias de Tipos
- Orientación a Objetos
- Libraries
- Programación Asincrónica

El Lenguaje Dart (1)

- **Dart** es un lenguaje de programación de código abierto, optimizado para clientes, desarrollado por la empresa *Google*.
- La primera versión se lanzó en el año 2011 con la premisa de ofrecer una alternativa más moderna al lenguaje *JavaScript* utilizado en los navegadores *web*, pero con el tiempo esta idea fue perdiendo fuerza.
- Sin embargo en agosto de 2018 *Google* lanzó la primera versión de un *framework* para el desarrollo de aplicaciones multiplataforma llamado **Flutter**, utilizando *Dart* como lenguaje de programación (ya en su versión 2, lanzada ese mismo año).

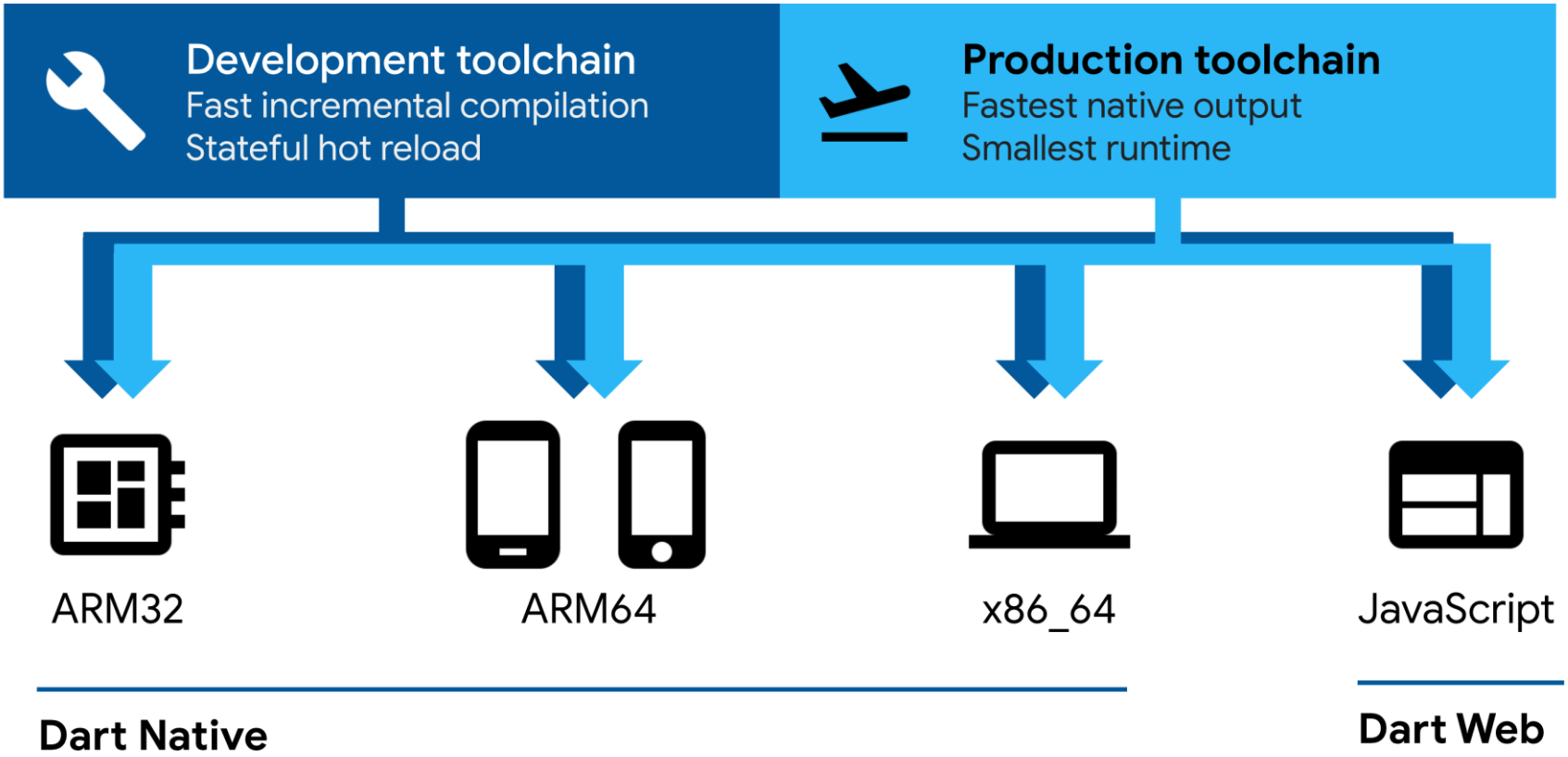
El Lenguaje Dart (2)

- *Dart* es un lenguaje orientado a objetos con análisis estático de tipos, y una sintaxis muy similar a la de otros lenguajes como *JavaScript* o *Java*, lo que facilita mucho su aprendizaje si ya se conoce alguno de éstos.
- Puede compilar a código nativo de diferentes plataformas móviles o de escritorio (**Dart Native**), y también a *JavaScript* (**Dart Web**).
- *Dart Native* ofrece dos alternativas de compilación: **JIT** (*Just In Time*: el código se compila a medida que es requerido durante su ejecución) y **AOT** (*Ahead of Time*: el código se compila completamente antes de su ejecución).

El Lenguaje Dart (3)

- La compilación *JIT* facilita el desarrollo de las aplicaciónes, ya que éstas pueden ejecutarse en una máquina virtual, recompilándolas incrementalmente (lo que permite características de desarrollo que incrementan la productividad como el *Hot Reload*).
- La compilación *AOT* optimiza la ejecución de la aplicación cuando ya está lista para ponerse en producción, haciendo que los tiempos de carga sean más cortos y consistentes.
- *Dart Web* también tiene dos modos de compilación: **incremental** (acelerando así los ciclos de desarrollo), y **optimizada** (aplicando técnicas que hacen que el código *JavaScript* generado se ejecute más rápido).

El Lenguaje Dart (4)



El Lenguaje Dart (5)

- Sin importar la plataforma de destino o la manera de compilar el código, la ejecución del código *Dart* es llevada a cabo por el **Dart Runtime**.
- El *Dart Runtime* tiene (entre otras) las siguientes responsabilidades:
  - Manejo de la memoria y recolección de basura
  - Verificación de tipos
  - Manejo de Isolates
  - etc.
- El *Dart Runtime* forma parte de la máquina virtual de *Dart*, y también es incluido en los ejecutables autocontenidos generados para las plataformas nativas.

Null Safety

- A partir de la versión 2.12 de *Dart* se introdujo una característica llamada **Null Safety**.
- Esto significa que las variables no pueden contener un valor *null*, a menos que se indique explícitamente dicha posibilidad mediante un signo de interrogación (*?*) luego de su tipo al declararla (o sean de tipo *dynamic*).
- También es posible asignar una expresión *nullable* a una variable *non nullable* colocando al final de la misma un signo de exclamación (*!*). Esto debe hacerse si se está seguro que dicha expresión nunca generará un valor *null* como resultado. En caso contrario se lanzará una excepción en tiempo de ejecución.

La función main()

- Toda aplicación *Dart* debe tener una función de alto nivel (*top level function*) llamada **main**.
- La función *main()* sirve de punto de entrada a la aplicación. Su código es lo primero que se ejecuta al inciar el programa.
- Dicha funcion tiene tipo de retorno *void* y opcionalmente puede recibir un parámetro de tipo *List<String>* para obtener los argumentos pasados al invocar la aplicación desde la consola.

Variables (1)

- En *Dart* las variables se pueden definir a nivel local dentro de una función, a nivel del cuerpo de una clase (atributos de clase y de instancia), e incluso fuera una clase, como variables de alto nivel (*top level variables*).
- Las variables se pueden definir mediante la palabra **var** seguida de un identificador.
- Si al declararla se la inicializa asignándole un valor con el operador de asignación (=), el tipo de dato de la variable será inferido a partir de dicho valor. En caso contrario será de tipo *dynamic*, lo que significa que su tipo de dato podrá cambiar y no se realizará comprobación estática de tipos.

Variables (2)

- También es posible indicar el tipo de dato de forma explícita, reemplazando la palabra *var* por el tipo de dato deseado.
- Los tipos de dato más comunes son:
  - **int, double**: números enteros y decimales
  - **String**: cadenas de caracteres
  - **bool**: valores lógicos true / false
  - **List**: arreglo (vector) indexado de elementos
  - **Set**: colección sin orden de elementos únicos
  - **Map**: conjunto de pares clave - valor
  - **Object**: clase base de todas las demás clases
  - **dynamic**: deshabilita la comprobación estática de tipos

Variables (3)

- Las variables *nullables* se inicializan por defecto con el valor *null*, mientras que las *non nullables* deben ser inicializadas explícitamente con un valor apropiado antes de utilizarlas.
- La palabra **late** se utiliza para indicar que una variable *non nullable* se inicializará más adelante antes de utilizarla, bajo la responsabilidad del programador (sino se lanzará una excepción).
- Puede resultar útil sobre todo con las variables de alto nivel y las variables de instancia *non nullables* no inicializadas en la declaración, para las que *Dart* no puede determinar si han sido inicializadas antes de utilizarlas.

Variables (4)

- Esto provoca que *Dart* indique un error si no se inicializa una variable *non nullable* de alto nivel o de instancia, a menos que se marque dicha variable con la palabra *late*.
- Por otra parte, si a una variable *late* se le asigna un valor en la propia declaración, su inicialización efectiva es postergada hasta el momento en que sea utilizada por primera vez.
- Esto puede resultar útil si no se está seguro de que llegue a ser necesario utilizar dicha variable y su inicialización es costosa, pero se le quiere dejar asignado su valor desde un principio.

Variables (5)

- Una variable puede declararse con la palabra **final** si se desea que sólo pueda asignársele valor una única vez (al declararla o en otra sentencia).
- Si no se indica el tipo de dato, se infiere a partir del valor asignado en la declaración.
- Para definir una variable como constante se utiliza la palabra **const** seguida del tipo de dato (opcional), su identificador, y luego mediante el operador de asignación (=) un valor. Todo en la misma sentencia.
- Si no se indica el tipo de dato, se infiere a partir del valor asignado en la declaración.
- Dentro del cuerpo de una clase, las variables constantes deben además ser marcadas como *static*.

Strings (1)

- Los *Strings* se pueden escribir entre comillas simples o dobles.
- Se pueden utilizar tres comillas de apertura y tres de cierre para escribir *Strings* multilinea.
- Soportan interpolación. Es decir, se pueden incluir dentro del *String* variables precedidas del signo *$*
- Dicha variable será reemplazada en la cadena por su valor.
- También se pueden interpolar expresiones encerradas entre llaves luego del signo *$*
- Los *Strings* se pueden concatenar con el signo de +, y también escribiéndolos uno a continuación del otro sin utilizar ningún operador adicional.

Strings (2)

- Dentro de un *String* se pueden escapar caracteres especiales con el signo \
- Si se precede un *String* con la letra *r* minúscula, no se procesan los caracteres escapados.
- Para convertir un número (*int* o *duble*) a un String, se puede utilizar el método *toString()*.
- Para convertir un *String* a un número se puede utilizar el método *parse(int)* o *parse(double)* de las clases *int* y *double* respectivamente.

Lists (1)

- Los arreglos indexados de una dimensión (vectores) se definen en *Dart* con el tipo **List**.
- Se puede limitar el tipo de elementos que contiene un *List* indicando un parámetro *generic* al declararlo.
- Para acceder a un elemento a partir de su índice se utilizan paréntesis rectos luego del nombre de la variable, indicando el índice dentro de éstos (el primer elemento tiene índice 0, y si se indica un índice que no existe se dispara una excepción).
- Se puede cambiar el valor de un elemento mediante el operador de asignación (=).
- Para consultar la cantidad de elementos que contiene un *List* se utiliza su propiedad *length*.

Lists (2)

- El método *add(tipo)* agrega un nuevo elemento al *List*, y el método *addAll(iterable)* permite agregar una lista de elementos.
- Los *List* literales se definen mediante paréntesis rectos, indicando opcionalmente sus elementos separados por coma (puede quedar una coma al final).
- Para agregar los elementos de otro *List* en un *List* literal se puede utilizar el operador *spread* (... o ...*?* para evitar excepciones si la lista a agregar es nula).
- También se pueden utilizar *(collection) if* y *(collection) for* para agregar elementos al definir un *List* literal.
- Se puede recorrer con un bucle *for (var e in lista) { ... }*.

Sets (1)

- El tipo **Set** define listas no ordenadas de elementos únicos (sin elementos repetidos).
- Se puede limitar el tipo de elementos que contiene un *Set* indicando un parámetro *generic* al declararlo.
- Para acceder a un elemento a partir de su índice se utiliza el método elementAt(int), indicando el índice mediante el parámetro que recibe (el primer elemento tiene índice 0, y si se indica un índice que no existe se dispara una excepción).
- Se puede quitar un elemento del *Set* con el método remove(tipo).
- Para consultar la cantidad de elementos que contiene un *Set* se utiliza su propiedad *length*.

Sets (2)

- El método *add(tipo)* agrega un nuevo elemento al *Set*, y el método *addAll(iterable)* permite agregar un conjunto de elementos.
- Los *Set* literales se definen mediante llaves, indicando opcionalmente sus elementos separados por coma (puede quedar una coma al final).
- Para definir un *Set* literal vacío y asignarlo a una variable que no especifique explícitamente su tipo (por ejemplo, declarada con *var*), se indica un parámetro generic antes de las llaves: <*tipo*>*{}*.
- Esto se hace para diferenciarlo de un *Map* literal vacío, cuya sintaxis es similar.

Sets (3)

- Para agregar los elementos de otro *Set* en un *Set* literal se puede utilizar el operador *spread* (... o ...*?* para evitar excepciones si el conjunto a agregar es nulo).
- También se pueden utilizar *(collection) if* y *(collection) for* para agregar elementos al definir un *Set* literal.
- Se puede recorrer con un bucle *for (var e in conjunto) { ... }*.

Maps (1)

- Un **Map** es un objeto que contiene un conjunto de pares clave - valor. Las claves no pueden repetirse, pero los valores sí.
- Se puede limitar el tipo de las claves así como también el de los valores de un *Map*, indicando parámetros *generic* al declararlo.
- Para acceder a un valor a partir de su clave se utilizan paréntesis rectos luego del nombre de la variable, indicando la clave dentro de éstos (si se indica una clave que no existe se obtiene el valor *null*).
- Se puede cambiar o incluso agregar un valor mediante el operador de asignación (=).

Maps (2)

- Para consultar la cantidad de valores que contiene un *Map* se utiliza su propiedad *length*.
- El método *addAll(Map)* permite agregar un conjunto de valores (con sus correspondientes claves) al *Map*.
- Los *Map* literales se definen mediante llaves, indicando opcionalmente sus valores precedidos por la clave correspondiente y un signo de dos puntos, separados por coma (puede quedar una coma al final).
- Para agregar elementos de otro *Map* en un *Map* literal se puede utilizar el operador *spread* (... o ...?).
- También se pueden utilizar *if* y *for* para agregar valores al definir un *Map* literal.
- Se puede recorrer con un bucle *for (var c in mapa.keys) { ... }* o *for (var v in mapa.values) { ... }*.

Fechas y Horas (1)

- Para trabajar con fechas y horas en *Dart* existen las clases *DateTime* y *DateFormat*.
- La clase **DateTime** cuenta con el método *now()* para obtener un objeto *DateTime* con la fecha y hora actual.
- También cuenta con el método *parse(String)* para obtener un objeto *DateTime* a partir de una cadena de texto con la información de la fecha y opcionalmente la hora, en formato estandarizado.
- Un objeto *DateTime* tiene propiedades como *weekday*, *year*, *month*, *day*, *hour*, *minute*, *second*, *etc*. para acceder a las partes que componen la fecha y hora que representa.

Fechas y Horas (2)

- También cuenta con métodos como *add(Duration)* y *subtract(Duration)* para sumarle o restarle un intervalo de tiempo.
- Un objeto **Duration** se puede construir a partir de una cantidad de días, horas, minutos, segundos, milisegundos y microsegundos.
- Los métodos booleanos *isAtSameMomentAs(DateTime)*, *isAfter(DateTime)* e *isBefore(DateTime)* permiten consultar si una fecha y hora es igual, posterior, o anterior a otra.
- El método *difference(DateTime)* devuelve un objeto *Duration* con la diferencia entre una fecha y hora, y otra.

Fechas y Horas (3)

- Si se desea formatear o parsear una fecha y hora a partir de un formato personalizado, se utiliza la clase **DateFormat**.
- Se necesita agregar al proyecto la dependencia *intl* e importar la librería '*package:intl/intl.dart*'.
- Se puede construir un objeto *DateFormat* a partir de un *String* con el formato de la fecha y hora.
- También cuenta con una gran cantidad de constructores con nombre que definen formatos estandarizados.
- Luego se puede formatear una fecha y hora a texto con el método *format(DateTime)*, o convertir un texto en una fecha y hora con el método *parse(String)*.

Funciones (1)

- En *Dart* las funciones se pueden definir a nivel del cuerpo de una clase (métodos de clase y de instancia), fuera de una clase como funciones de alto nivel (*top level functions*), e incluso a nivel local dentro de otra función.
- Las funciones son objetos de tipo **Function**, de manera que pueden ser asignadas a una variable o pasadas como argumentos a través de los parámetros de otras funciones.
- Si al declararlas se omite el tipo de retorno, retornan *dynamic*.
- Si se desea indicar que no retornan un valor utilizable se utiliza el tipo *void*.

Funciones (2)

- Si el cuerpo de una función contiene solo una expresión, se puede utilizar la sintaxis de flecha:

  bool par(int numero) => numero % 2 == 0;

- Las funciones pueden recibir cualquier cantidad de parámetros posicionales requeridos, y a continuación parámetros posicionales opcionales o parámetros con nombre (no ambos). Puede quedar una coma al final.
- Si se omite el tipo de los parámetros, se consideran *dynamic*.
- Los parámetros posicionales opcionales se indican entre paréntesis rectos, separados por coma.

Funciones (3)

- Si un parámetro posicional opcional es de tipo *nullable* y no se se le asigna un valor predeterminado con el operador de asignación (=), su valor predeterminado será *null*. Y si es de tipo *non nullable* y no se le asigna un valor predeterminado produce un error de compilación.
- Los parámetros con nombre se indican entre llaves, separados por coma, su nombre no puede comenzar con subguión, y se les puede asignar un valor predeterminado con el operador de asignación (=).
- Por defecto son opcionales, a menos que se marquen con la palabra *required*.

Funciones (4)

- Si un parámetro con nombre es de tipo *nullable* y no se le asigna un valor predeterminado, su valor predeterminado será *null*. Y si es de tipo *non nullable* y no se le asigna un valor predeterminado, produce un error de compilación si no se lo marca con la palabra *required*.
- Las funciones también pueden ser anónimas, es decir que pueden no tener nombre.
- Como cualquier otra función, las funciones anónimas pueden asignarse a una variable y/o pasarse como argumento a través de los parámetros de otras funciones.

Excepciones (1)

- Las excepciones en *Dart* derivan de **Exception** o de **Error** (errores de programación o del sistema) y son *unchecked*, lo que significa que los métodos no deben declararlas en su firma si no se las captura.
- Las excepciones pueden manejarse con la estructura **try - on [catch] - finally**:

```
try { ... }
on MiExcepcion catch (e) { ... }
on MiOtraExcepcion { ... }
catch (e, s) { ... } // Captura cualquier tipo.
finally { ... }
```

Excepciones (2)

- Se utiliza *on* <*tipo*> para indicar el tipo de la excepción a capturar, y/o *catch (e)* en caso de querer tener acceso a dicho objeto.
- El *catch* puede recibir un segundo parámetro de tipo *StackTrace* para acceder a la información de la pila de llamadas.
- Para relanzar una excepción capturada se utiliza la palabra *rethrow*.
- Se puede lanzar explícitamente una excepción o incluso cualquier tipo de objeto con la palabra *throw*.

Alias de Tipos (1)

- La palabra **typedef** permite definir un alias para cierto tipo de dato:

  typedef ListaEnteros = List<int>;

  ListaEnteros numeros = [1, 2, 3, 4, 5];

- El tipo de dato también podría ser una función de ciertas características:

  typedef BooleanoAPartirDeEntero = bool Function(int e);

Orientación a Objetos (1)

- En *Dart*, cualquier cosa que se pueda almacenar en una variable es un objeto y todos los objetos son instancia de una clase.
- Todas las clases, a excepción de la clase *Null*, derivan de la clase *Object*.
- Para acceder a los atributos y métodos de un objeto se utiliza el operador . (punto), pero también existe el operador ?. (interrogación punto) para evitar que se lance una excepción si la variable apunta a *null*.
- No existen los modificadores de acceso. Los miembros de una clase son públicos, a menos que su nombre comience con un subguión para que no se puedan acceder desde fuera de la biblioteca (*library*).

Orientación a Objetos (2)

- En *Dart* no existe la sobrecarga de operaciones (es decir, definir en la misma clase varias operaciones con el mismo nombre pero distinta firma). Por lo cual tampoco existe la sobrecarga de constructores.
- Existen diferentes tipos de constructores. Por ejemplo, se puede definir un constructor dentro de una clase, definiendo un método que se llame igual que ésta (**constructor sin nombre**) y que no indique tipo de retorno.
- También se pueden definir **constructores con nombre**, cuyos nombres comienzan con el nombre de la clase y luego separando mediante un punto finalizan con un identificador.

Orientación a Objetos (3)

- Si no se define explícitamente ningún constructor en una clase, *Dart* proporciona un constructor por defecto, que no recibe parámetros, cuya implementación invoca al constructor sin parámetros de la clase base.
- Los valores de los atributos *non nullable* y/o *final* deben inicializarse antes de la ejecución del cuerpo del constructor.
- Esto puede hacerse asignándoles un valor al momento de declararlos. También utilizando parámetros de inicialización formales (*this.parametro* o *super.parametro*) en el constructor. O mediante una lista de inicialización en el constructor.

Orientación a Objetos (4)

- La lista de inicialización se define colocando : (dos puntos) luego de cerrar los paréntesis de los parámetros del constructor, separando cada inicialización de la siguiente con una coma, antes de abrir las llaves del cuerpo.
- Al final de la lista de inicialización se puede invocar a un constructor de la clase base mediante *super(...)*.
- También se puede invocar a otro constructor de la misma clase mediante *this(...)* luego de los : (dos puntos), omitiendo el cuerpo del mismo. Este tipo de constructor se conoce como **constructor redireccionador** y no puede tener parámetros de inicialización formales ni invocar a la clase base.

Orientación a Objetos (5)

- Si a partir de una clase se van a construir instancias que nunca cambien su estado, se pueden marcar sus atributos de instancia con la palabra *final* y los constructores con la palabra *const*. Este tipo de constructor se conoce como **constructor constante**.
- Esto hará que las instancias que se creen a partir de dicha clase utilizando la palabra *const*, sean constantes en tiempo de compilación, mejorando así la performance de la aplicación.

Orientación a Objetos (6)

- Si un constructor no siempre crea una instancia de su propia clase (por ejemplo, si devuelve una instancia obtenida de un *cache*, o crea una instancia de una clase derivada), se debe marcar con la palabra *factory*.
- Este tipo de constructor se llama **constructor fábrica**, y puede también utilizarse si se necesita inicializar atributos *final* aplicando cierta lógica compleja que no se pueda manejar desde la lista de inicialización.

Orientación a Objetos (7)

- En *Dart*, todos los atributos de una clase tienen un *getter* implícito. Además, si no son *final* o a pesar de serlo están marcados con la palabra *late*, también tienen un *setter* implícito.
- Se pueden crear métodos *getter* y *setter* personalizados con las palabras **get** y **set** respectivamente.
- Al definir un método *get* no se indican paréntesis para definir parámetros.
- Al definir un método *set* no se estila poner *void* como tipo de retorno (ni ningún otro tipo) y se debe indicar entre paréntesis un parámetro para recibir el valor a guardar en el objeto.

Orientación a Objetos (8)

- En *Dart* cada clase puede extender de una única clase (**extends**), pero puede implementar varias interfaces (**implements**) y utilizar varios *mixins*.
- No existe la palabra *interface* para definir una interfaz. Cualquier clase define una interfaz implícitamente (típicamente, para definir una interfaz, se define una clase abstracta).
- Al redefinir operaciones, se puede utilizar la annotation *@override* (con minúsculas) para indicarle explícitamente al compilador la intención de hacerlo.
- Para comprobar si una variable apunta a una instancia de determinado tipo se utiliza la palabra **is**.
- Para castear una variable a un subtipo se utiliza **as**.

Orientación a Objetos (9)

- Los **mixins** permiten reutilizar el código de una clase en otras, de manera que el código del *mixin* se incorpore a la clase que lo aplica.
- Para definir un *mixin* se debe crear una clase que extienda de *Object* y no defina constructores. Se utiliza la palabra *mixin* para declararlo, a menos que se quiera utilizar como una clase normal, en cuyo caso también se puede declarar con la palabra *class*.
- Se puede restringir el tipo de clase en la que se podrá aplicar un mixin con la palabra *on* a continuación del nombre, seguida del tipo base permitido.
- Para aplicar uno o varios mixins a una clase se utiliza la palabra *with*, separándolos con coma.

Libraries (1)

- Una **library** es un conjunto de clases, variables, funciones, etc.
- Para importar una *library* en un archivo de código *Dart* se utiliza la palabra **import** y a continuación entre comillas la *URI* de la *library*. Pueden haber múltiples sentencias *import* en el mismo archivo.
- La *URI* de las libraries incluidas en *Dart* utilizan el esquema *dart:* (*dart:core*, *dart:convert*, *dart:math*, *dart:io*, etc.), y las publicadas en repositorios como **pub.dev** utilizan el esquema *package:*
- Las *libraries* que forman parte del mismo proyecto pueden importarse indicando la ruta del archivo.

Libraries (2)

- Al importar una *library*, todos los elementos con identificadores que no comiencen con un subguión (_) estarán disponibles para ser utilizados.
- Si se desea importar únicamente algunos elementos de una library puede utilizarse la palabra **show** o **hide** a continuación de la *URI* de la *library*, seguida de los identificadores de los elementos a importar u ocultar, separándolos mediante comas.
- Para evitar colisiones de identificadores al importar una *library* se puede utilizar la palabra **as** luego de la *URI* para asignarle un prefijo. Luego, para utilizar los elementos importados debe escribirse el prefijo y un punto antes de su identificador.

Libraries (3)

- Para definir una *library* personalizada se utiliza la palabra **library** y a continuación un identificador. Puede haber una sóla sentencia *library* por archivo.
- Se puede definir parte de la *library* en otro archivo que deberá tener una sentencia **part of** con el identificador de la *library* a la que pertenece, y en el archivo principal de la *library* deberá importarse con una sentencia **part**.
- Una *library* también puede exportar otras *libraries* con sentencias **export** seguidas de la *URI* correspondiente. Al importar dicha *library* en otro archivo, se importarán también todas las que ésta exporte.

Programación Asincrónica (1)

- En *Dart* existen muchas funciones que retornan un objeto de tipo *Future* o *Stream*.
- Estas funciones son funciones asincrónicas, que retornan inmediatamente luego de invocarlas, sin esperar a que se complete la tarea (frecuentemente costosa) que realizan.
- Un **Future** es un objeto que representa la promesa de un resultado de un proceso asincrónico.
- Su método *then(...)* permite registrar una función *callback* para que ésta se ejecute cuando el proceso sea completado exitosamente (recibiendo por parámetro el valor devuelto por el proceso).

Programación Asincrónica (2)

- También cuenta con el método *catchError(...)* para manejar errores, y *whenComplete(...)* para realizar acciones al finalizar, independientes del resultado exitoso o erróneo.
- Para definir una función asincrónica personalizada se puede utilizar la palabra **async** luego de cerrar los paréntesis de los parámetros y antes de abrir las llaves del cuerpo.
- Una función marcada con *async* debe retornar *void* o un objeto *Future<TIPO>* donde *TIPO* define el tipo que retorna dicha función.

Programación Asincrónica (3)

- Al invocar una función asincrónica, el programa continuará su ejecución sin esperar a que ésta complete su tarea, a menos que se anteceda su invocación con la palabra **await** (en cuyo caso la función donde se encuentra dicha invocación deberá ser asincrónica marcándola con la palabra *async*).
- También se puede definir una función asincrónica generadora utilizando la palabra **async*** luego de cerrar los paréntesis de los parámetros y antes de abrir las llaves del cuerpo.
- Una función marcada con *async** debe retornar un *Stream*<*TIPO*> donde *TIPO* define el tipo de los eventos (objetos) emitidos por el *Stream*.

Programación Asincrónica (4)

- Un **Stream** es una secuencia de eventos asincrónica que puede ser utilizada para transmitir y recibir datos de manera eficiente.
- Permite manejar secuencias de valores de manera asíncrona, lo que lo hace útil para situaciones en las que se espera la llegada gradual de datos o eventos a lo largo del tiempo.
- Los *Streams* siguen un modelo unidireccional, donde los datos fluyen desde la fuente hacia los consumidores, que pueden escuchar (suscribirse) a los eventos del *Stream* y reaccionar a ellos.
- Se les puede aplicar operaciones de transformación para modificar, filtrar o combinar los eventos antes de que lleguen a los consumidores.

Programación Asincrónica (5)

- También manejan errores de manera eficiente, permitiendo capturar y manejar excepciones que puedan ocurrir durante la transmisión de datos.
- Para emitir eventos en el *Stream* que devuelve una función generadora, se utiliza la palabra **yield**.
- Al utilizar la palabra *yield*, la ejecución de la función generadora se detiene temporalmente para permitir que otros procesos se ejecuten antes de continuar con la siguiente emisión.
- También se puede utilizar la palabra *yield** para emitir todos los valores de otro *Stream* provistos por otra función generadora antes de continuar con la siguiente emisión.

Programación Asincrónica (6)

- Es posible suscribirse a un *Stream* mediante su método *listen(...)*, que recibe por parámetro una función *callback* para manejar qué hacer con el evento emitido, y devuelve un **StreamSubscription**.
- También se le pueden pasar por parámetro funciones *callback* para manejar errores y/o la finalización de la emisión.
- Estas funciones *callback* también se pueden definir con los métodos *onData(...)*, *onError(...)* y *onDone(...)* del objeto *StreamSusbscription* devuelto.
- Al finalizar la emisión, es conveniente cancelar la suscripción con el método *cancel()* del objeto *StreamSubscription*.