Built-in Data Types in Dart

By: Laams Innovation Lab

💡 Tip 1:

Computers do two things: **storing data** & **computing data.**

Questions inferred from Tip 1

1) How is real-world **data** represented in a computer?
2) If everything is binary, then, why all the **data types**?
3) Can I make my own **data type**?
4) Can I convert a **data type** into another?
5) How does a computer **store** different **data types**?
6) How does a computer **compute** all sorts of **data types**?

1) How is real-world data represented in Computers?

Computers don’t understand **data**. They have billions of **transistors** which could be **storing** *electricity* **On** or letting of it **Off**.

Humans **represent** the transistors **On** or **Off** states as **0s** or **1s**, which are representation of data in *binary system*.

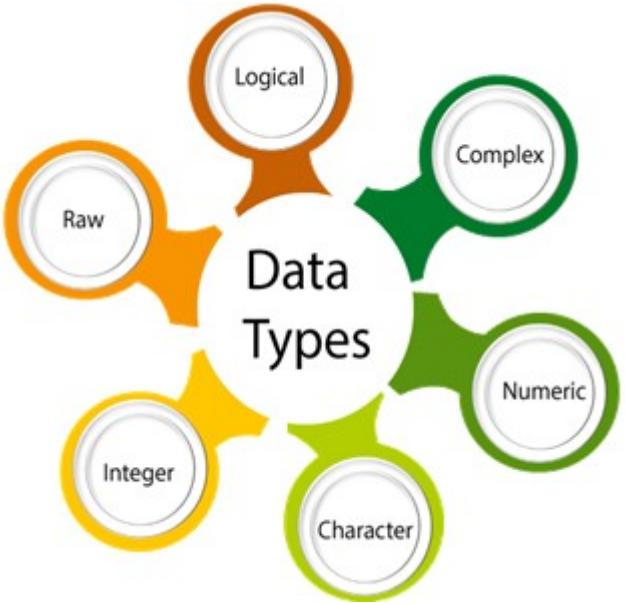
2) If everything is binary, then, why all the data types?

It is very difficult to think **data** in **binary**; therefore, data types helps us think in **abstract**.

Data types determines how the **programmer intends** to use the **data**.

Data Types tells computers how to **interpret** and **compute** the data, *constraining* the **operations** that can be done on the **data**.

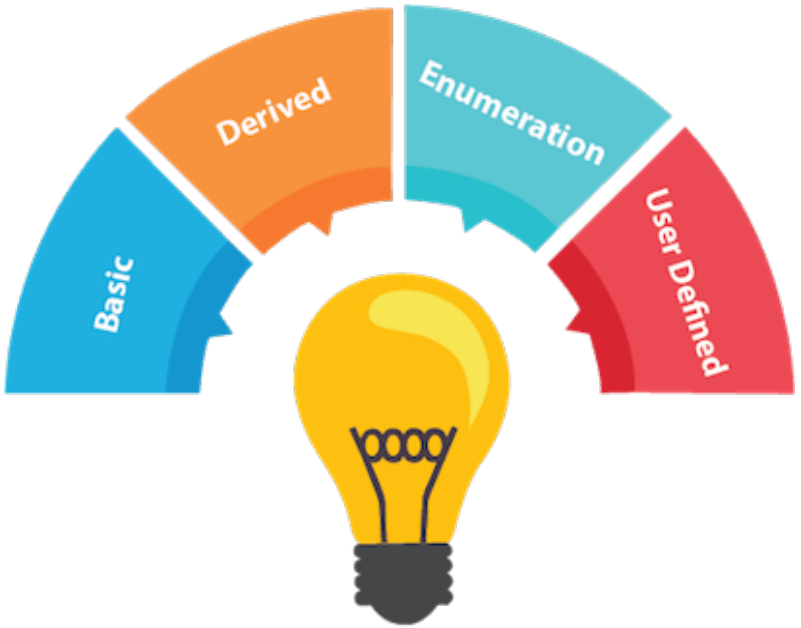
Data Types helps the computers decide how & how much **memory** should be allocated for the **data**



Data Types

Data Types are classified into four categories:

- **Basic Data Types:** Primitive
- **Derived Data Types:** those that are defined in terms of other data types
- **Enumerations**: enum
- **User Defined:** The types which can be defined by the programmer for better **data** representation.



Some of these **data types** could be built-in to a PL. Built-in **data types** are predefined by a programming language for the purposes of **abstraction**, **convenience**, and **productivity**.

Dart's Built-in Data Types

Dart is a very productive programming language and has numerous **built-in data types**.

Since, Dart is an **object-oriented** language; therefore, all **data types** in Dart are **objects**.

The built-in **data types** in Dart could be divided in to *primitives* and *data structures*.

Dart's Built-in Primitive Data Types

Dart's built-in primitive **data** **types** include **bool**, **num**, **int**, **double** and **String**. Each of which **represents** a specific type of **data** and has its own designated **memory space**.

Each of them can be **instantiated** with designated built-in **literals** or by calling their **constructors**.

Operations could be performed on them using specific **operators**, calling their **properties** or **methods**, and using other **control flows** and **functions**.

Some **data** **types** could be converted to other **data** **types** using specific **procedures** or **methods**.

Null Data Type

Representation: **Null data type** represents an **empty** value in Dart.

Size: 1 byte (8 bits)

Instantiation: Can only be initialized with the **literal null** value, or an object which is **nullable**.

Operations: There are multiple ways to do operations on this **data type**:

- **Using its properties method:** which can be accessed after the data type is initialized.
- **Using operators:** Such as comparison, logical, type test, assignment, if-false operators.

Conversion: Can only be converted to **String**, either by calling its *toString()* method or by **interpolating** it inside a **String literal**.

bool Data Type

Representation: **bool data type** could represent a *yes* | *no*, *correct* | *incorrect* and **true** | **false** in Dart.

Size: 1 byte (8 bits)

Instantiation: Can be initialized with both **literals** and **constructors**:

- ***Literals*** are limited to **true** | **false** values only.
- ***Constructors*** include: **bool**.fromEnvironment(), **bool**.hasEnvirontment().

Operations: There are multiple ways to do operations on this **data type**:

- **Using its properties method:** which can be accessed after it is initialized.
- **Using operators:** Such as comparison, logical, type test, assignment, if-false, null-aware and ! operators.

Conversion: Can only be converted to **String**, either by calling its *toString()* method or by *interpolating* it inside a **String** *literal*.

num Data Type

Representation: num data **type** is the super-class for **int** and **double** and represents rational **numbers**.

Size: 4-8 bytes (32-64 bits) depending on the computer system architecture.

Instantiation: Can be initialized with both *literals* and *constructors*:

- ***Literals*** are constrained to **0.0-9.9** quintrillion values only.
- ***Constructor*** include: **num**.tryParse(**String** value).

Operations: There are multiple ways to do operations on this **data type**:

- **Using its properties method:** which can be accessed after it is initialized.
- **Using operators:** Such as arithmetic, comparison, logical, type test, assignment, if-false, null-aware and bit-wise operators.

Conversion: It can be converted and casted to and from multiple data types:

- Can convert to **int** using toInt() and other methods, to **double** using toDouble() and other methods, and to **String**, either by calling its *toString()* and other methods or by **interpolating** it inside a **String literal**. And be converted from **String** using the tryParse() method.
- Can be casted from **int** and **double** using the **as** operator. to int using toInt(), to double using toDouble()

int Data Type

Representation: **int data type** represents positive and negative**integers**.

Size: 4-8 bytes (32-64 bits) depending on the computer system architecture.

Instantiation: Can be initialized with both *literals* and *constructors*:

- ***Literals*** are constrained to **0-9** quintrillion values only.
- ***Constructor*** include: **int**.tryParse(**String** value), **int**.fromEnvironment(**String** value).

Operations: There are multiple ways to do operations on this **data type**:

- **Using its properties method:** which can be accessed after it is initialized.
- **Using operators:** Such as arithmetic, comparison, logical, type test, assignment, if-false, null-aware and bit-wise operators.

Conversion: It can be convert to **double** using toDouble(), to signed and unsigned integers of different bitlengths using toSigned(int width) and other methods, and to **String**, either by calling its *toString()* and other methods or by **interpolating** it inside a **String literal**. And can be converted from **String** using the tryParse() method. It can also be **casted** as **num** using **as** operator.

double Data Type

Representation: double data **type** represents positive and negative **decimal numbers**.

Size: 4-8 bytes (32-64 bits) depending on the computer system architecture.

Instantiation: Can be initialized with both *literals* and *constructors*:

- ***Literals*** are constrained to **0.0-9.9** quintrillion values only.
- ***Constructor*** include: **int**.tryParse(**String** value), **int**.fromEnvironment(**String** value).

Operations: There are multiple ways to do operations on this **data** **type**:

- ***Using its properties method:*** which can be accessed after it is initialized.
- ***Using operators:*** Such as arithmetic, comparison, logical, type test, assignment, if-false, null-aware and bit-wise operators.

Conversion: It can be convert to **int** using toInt() and other methods, and to **String**, either by calling its *toString()* and other methods or by **interpolating** it inside a **String** **literal**. And can be converted from **String** using the tryParse() method.

String Data Type

Representation: **String data type** represents a **series of sequential characters**.

Size: Each of its characters takes 2 bytes (16 bits).

Instantiation: Can be initialized with both *literals* and *constructors*:

- ***Literals*** are constrained to “ ” **(string literal)** only.
- ***Constructor*** include: **String**.fromCharCode(**int** charCode), **int**.fromEnvironment(**String** value). String.fromCharCodes(Iterable<int> codes)

Operations: There are multiple ways to do operations on this **data type**:

- ***Using its properties method:*** which can be accessed after it is initialized.
- ***Using operators:*** Such as concatenation, comparison, logical, type test, assignment, if-false and null-aware operators.

Conversion: It can be convert to **int** using int.tryParse(), to **double**, using double.tryParse(). Any type can be converted to String using toString() method

Dart's Built-in Data Structures

Dart's built-in data structures are generic, which makes them type safe, and include, List<E>, Set<E> and Map<K,V>, which could represent a **collection** of specific **data type** of data, and whose needed **memory space** depends on these collections elements.

Each of them can be **instantiated** with designated built-in **literals** or by calling their **constructors**.

Operations could be performed on them using specific **operators**, calling their **properties** or **methods**, and using other **control flows** and **functions**.

Some **data structures** could be converted to other **data types or data structures** using specific **procedures** or **methods**.

List<E> Data Structure

Representation: **List<E> data structure** is **generic containers** for storing an indexable ***collection of primitives or data structures*** *in sequential order*. Each item in a list is called and element (separated by comma); hence the <E> generic after its name.

Size: depends of the type of objects the collection stores.

Instantiation: Can be initialized with both *literals* and *constructors*:

- ***Literals*** are constrained to **<E>[]** only.
- ***Constructor*** include: **List<E>**.castFrom(), **List<E>**.unmodifiable(), **List<E>**.filled(), **List<E>**.from(), **List<E>**.generate(), **List<E>**.of()

Operations: There are multiple ways to do operations on this **data type**:

- ***Using its properties method:*** which can be accessed after it is initialized.
- ***Using operators:*** Such as cascade, spread, comparison, logical, type test, assignment, if-false, null-aware, collection if and collection for operators.

Conversion: can be converted to String using toString() method

Set<E> Data Structure

Representation: **Set<E>** **data** **structure** is **generic containers** for storing **an unordered dynamic collection of unique items**.

Size: depends of the type of objects the collection stores.

Instantiation: Can be initialized with both *literals* and *constructors*:

- ***Literals*** are constrained to **<E>{ }** only.
- ***Constructor*** include: **Set<E>()**, **Set<E>**.castFrom(), **Set<E>**.unmodifiable(), **Set<E>**.filled(), **List**.from(), **Set<E>**.generate(), **Set<E>**.of()

Operations: There are multiple ways to do operations on this **data** **type**:

- ***Using its properties method:*** which can be accessed after it is initialized.
- ***Using operators:*** Such as cascade, spread, comparison, logical, type test, assignment, if-false, null-aware, collection if and collection for operators.

Conversion: can be converted to String using toString() method

Map<K, V> Data Structure

Representation: **Map<K, V> data structure** is **generic containers** for storing a *key-value pair* ***dynamic collection of primitives or data structures***.

Size: depends of the type of objects the collection stores.

Instantiation: Can be initialized with both *literals* and *constructors*:

- ***Literals*** are constrained to **<K, V>{ }** only.
- ***Constructor*** include: **Map<E, V>()**, **Map<E, V>**.fromEntries(), **Map<E, V>**.from(), **Map<E, V>**.fromIterable(), **Map<E, V>**.fromIterables(), **Map<E, V>**.unmodifiable(), **Map<E, V>**.castFrom()

Operations: There are multiple ways to do operations on this **data type**:

- ***Using its properties method:*** which can be accessed after it is initialized.
- ***Using operators:*** Such as cascade, spread, comparison, logical, type test, assignment, if-false, null-aware, collection if and collection for operators.

Conversion: can be converted to String using toString() method

Runes Data Type

Representation: Runes data structure represents **iterable Unicode points of a String and symbols of the world's writing system**

Size: Each rune takes 2 bytes (16 bits).

Instantiation: Can be initialized by calling String properties and methods and by calling Runes *constructor*:

- Each rune an be created using ***.runes, codeUnits(), codeUnitAt(init index)*** **of String** value, in the form of a single rune (code unit) or in of list of code unites.
- ***Constructor*** include: **Runes(String string)**

Operations: Can use concatenation, comparison, logical, type test, assignment, if-false, null-aware and bit-wise operators.

Conversion: It can be convert to **String** using toString() method. Can be converted to List<rune> using toList() method. Can be converted to Set<rune> using toSet() method.

Symbol Data Type

Symbol data type represent **an operator or identifier declared in a Dart program**

We can instantiate a Symbol using its contructor Symbol(); or by prefixing an identifier with #.

Built-in types

The Dart language has special support for the following:

- Numbers (`int`, `double`)
- Strings (`String`)
- Booleans (`bool`)
- Lists (`List`, also known as *arrays*)
- Sets (`Set`)
- Maps (`Map`)
- Runes (`Runes`; often replaced by the `characters` API)
- Symbols (`Symbol`)
- The value `null` (`Null`)

3) Can I make my own data type?

Yes, you can! You can make your own data types using **enums** and **classes**.

Enumeration

Enumeration or **enums** are a special kind of classes used to represent a fixed number of constant integer values and properties.

Representation: it could represent ranking, classification, or choices.

Definition & Instantiation: it can be defined with the following syntax and instantiated using the Identifier.element.

```
enum Identifier {
 element, element, element
 }
```

Class

Every Object in Dart is an instance of a **class**, which create a **data type** with **custom properties and behaviors**, you can use the class.

To define a class you can use the class keyword, identifier and a block, inside which you can **declare** the properties and methods of the class

Classes have default constructors, which you can override.

```
class Identifier {
properties,
methods,
constructors
 }
```

4) Can I convert a data type into another?

Yes, you can! However, how and which data types or data structures you can convert depends on the type of object being converted, the type object is being converted to and the logic behind it.

All data types and data structures can be converted to String by calling their toString() method or by interpolating them inside a String literal.

5) How does a computer store different data types?

By creating variables. **Variables** allocate a specific amount of memory for the data type, that could be used by processor for **computing**.

6) How does a computer compute all sorts of data types?

Using the objects **properties** and **methods**

Using dart's built in **operators**

Using built-in and user defined **functions**

Using other **control flow** Statements