Dart Data Types

- Numbers
- Strings
- Booleans
- Lists
- Maps

Numbers

Integer − Integer values represent non-fractional values, i.e., numeric values without a decimal point epresented using the **int** keyword,

Double − Dart also supports fractional numeric values i.e. values with decimal points. The Double data type in Dart represents a 64-bit

Strings

The keyword **String** is used to represent string literals. String values are embedded in either single or double quotes.

Boolean

The Boolean data type represents Boolean values true and false. Dart uses the **bool** keyword to represent a Boolean value

List and Map

The data types list and map are used to represent a collection of objects. A **List** is an ordered group of objects

The **Map** data type represents a set of values as key-value pairs. The **dart: core** library

Dynamic Type

Dart is an optionally typed language. If the type of a variable is not explicitly specified, the variable's type is **dynamic**.

Variables

A variable is "a named space in the memory"

- 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 numbe

Type Syntax

```
var name = 'Smith';
String name = 'Smith';
int num = 10;
```

The dynamic keyword

Variables declared without a static type are implicitly declared as dynamic. Variables can be also declared using the dynamic keyword in place of the var keyword.

```
dynamic x = "tom";

  print(x);
```

Final and Const

The **final** and **const** keyword are used to declare constants. Dart prevents modifying the values of a variable declared using the final or const keyword. These keywords can be used in conjunction with the variable's data type or instead of the **var** keyword

The **const** keyword is used to represent a compile-time constant. Variables declared using the **const** keyword are implicitly final.

final Keyword

```
final variable_name
```

```
final val1 = 12;

  print(val1);


const pi = 3.14;

  const area = pi*12*12;

  print("The output is ${area}");


 final v1 = 12;

  const v2 = 13;

  v2 = 12;
```

Operators

- **Operands** − Represents the data
- **Operator** − Defines how the operands will be processed to produce a value.

- Arithmetic Operators
- Equality and Relational Operators
- Type test Operators
- Bitwise Operators
- Assignment Operators
- Logical Operators

Arithmetic Operators

```
var num1 = 101;
 var num2 = 2;
 var res = 0;

 res = num1+num2;
 print("Addition: ${res}");

 res = num1-num2;
 print("Subtraction: ${res}");

 res = num1*num2;
 print("Multiplication: ${res}");

 res = num1/num2;
 print("Division: ${res}");

 res = num1~/num2;
 print("Division returning Integer: ${res}");

 res = num1%num2;
 print("Remainder: ${res}");

 num1++;
 print("Increment: ${num1}");

 num2--;
```

```
  print("Decrement: ${num2}");
```

Equality and Relational Operators

```
var num1 = 5;
  var num2 = 9;
  var res = num1>num2;
  print('num1 greater than num2 ::  ' +res.toString());

  res = num1<num2;
  print('num1 lesser than  num2 ::  ' +res.toString());

  res = num1 >= num2;
  print('num1 greater than or equal to num2 ::  ' +res.toString());

  res = num1 <= num2;
  print('num1 lesser than or equal to num2  ::  ' +res.toString());

  res = num1 != num2;
  print('num1 not equal to num2 ::  ' +res.toString());

  res = num1 == num2;
  print('num1 equal to num2 ::  ' +res.toString());
```

test Operators

is Example

```
void main() {
   int n = 2;
   print(n is int);
}
```

is! Example

```
void main() {
   double  n = 2.20;
   var num = n is! int;
   print(num);
}
```

Bitwise Operators

Operator	Description	Example
Bitwise AND	a & b	Returns a one in each bit position for which the corresponding bits of both operands are ones.
Bitwise OR	a | b	Returns a one in each bit position for which the corresponding bits of either or both operands are ones.
Bitwise XOR	a ^ b	Returns a one in each bit position for which the corresponding bits of either but not both operands are ones.

Bitwise NOT	~ a	Inverts the bits of its operand.
Left shift	a ≪ b	Shifts a in binary representation b (< 32) bits to the left, shifting in zeroes from the right.
Signpropagating right shift	a ≫ b	Shifts a in binary representation b (< 32) bits to the right, discarding bits shifted off.

```
var a = 2;  // Bit presentation 10
  var b = 3;  // Bit presentation 11

  var result = (a & b);
  print("(a & b) => ${result}");
  result = (a | b);
  print("(a | b) => ${result}");
  result = (a ^ b);
  print("(a ^ b) => ${result}");

  result = (~b);
  print("(~b) => ${result}");

  result = (a < b);
  print("(a < b) => ${result}");

  result = (a > b);
  print("(a > b) => ${result}");
```

Assignment Operators

```
var a = 12;
  var b = 3;

  a+=b;
  print("a+=b : ${a}");

  a = 12; b = 13;
  a-=b;
  print("a-=b : ${a}");

  a = 12; b = 13;
  a*=b;
  print("a*=b' : ${a}");

  a = 12; b = 13;
  a/=b;
  print("a/=b : ${a}");

  a = 12; b = 13;
  a%=b;
  print("a%=b : ${a}");
```

Logical Operators

```
var a = 10;
  var b = 12;
  var res = (a<b)&&(b>10);

  print(res);
```

```
var a = 10;

  var b = 12;

  var res = (a>b)||(b<10);


  print(res);

  var res1 =!(a==b);

  print(res1);
```

&& and ||

```
var a = 10

var result = (a<10 && a>5)
```

```
var a = 10

var result = ( a>5 || a<10)
```