Primitive Types & Control Flow

By: Laams Innovation Lab

Contents

- Types of Knowledge
- Variables
- Primitive Data Types
- Operators
- Conditional Execution

Types of Knowledge

At the core computing devices do two things, **calculation** and **remembering** the result. But it does not know anything on its own.

Calculations are two types: ones which are **built in**, others which **you initiate**. Hence you create two types of knowledge

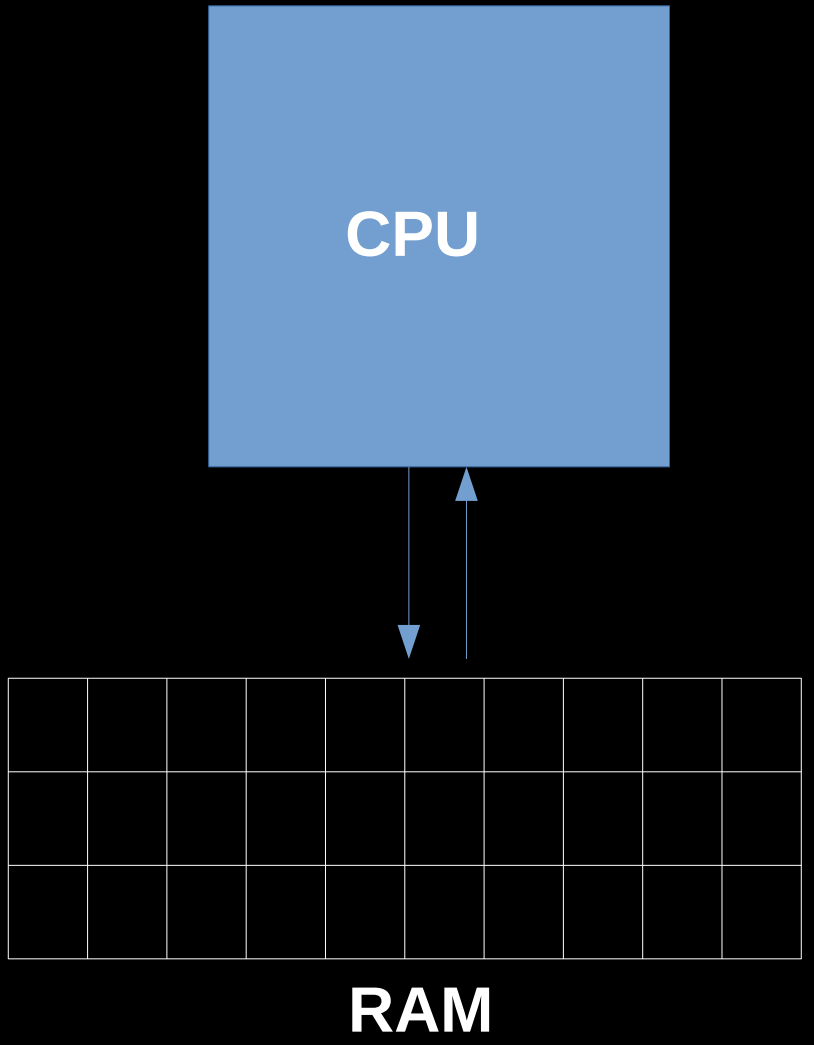
- **Declarative Knowledge**
  - Statement of Facts
- **Imperative Knowledge**
  - Is instructions, recipe, is about how to.

Variables

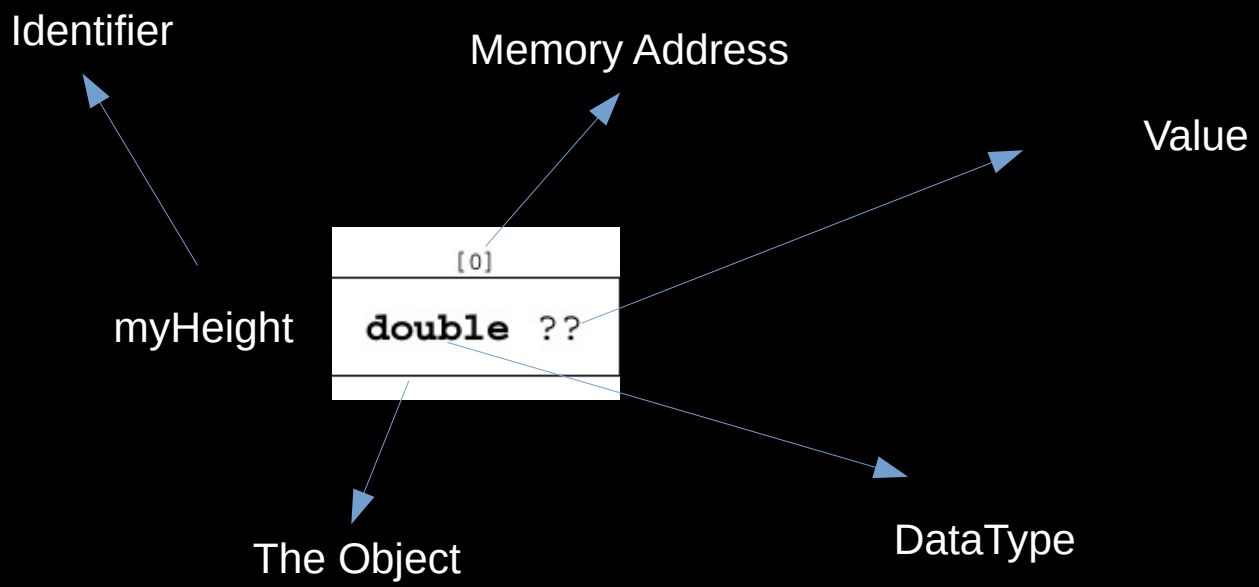
Variable is a reference to a chunk of **memory address with a specific type**, whose **value (data itself)** can most of the time be replaced with another.

Variables are recognized by the machine based on their memory address, represented in hexadecimal, in lower level languages, and recognized by the **programmer** through its **identifier (name)**.

Think of variables as containers holding information for you which you can change later or use it for a specific purpose.



Variables


Identifier
Memory Address
Value
[0]
myHeight
double ??
The Object
DataType

Syntax of a Variable

1. Variable Definition:

[keyword] DataType[?] identifier “Assignment Operator” value;

2. Variable Declaration:

[late] [keyword] DataType[?] identifier;

3. Variable Assignment & Re-assignment Syntax:

Identifier = value;

Syntax of a dynamic type Variable

4. Variable with dynamic type Definition:

[keyword] dynamic identifier = value;

5. Variable with dynamic type Declaration:

[late] [keyword] dynamic identifier;

6. Variable with dynamic type Assignment & Re-assignment Syntax:

Identifier = value;

Syntax of Type Inferred Variable

7. Type inferred variable definition:

[keyword] identifier = value;

Variable Identifiers Naming Conventions

- **DO NOT** use special words for naming variables
- Variable identifier **MUST NOT** start with number
- Name of the variable must be a **noun**
- Name identifiers with **lowerCamelCase (e.g. myAge)**
- Do Capitalize acronyms (HTTP, URL)
- A variable defined should be used.
- Use Intention-Revealing Names
- Make Meaningful Distinction
- Use Pronounceable Names
- Avoid Mental Mapping

Variables Important Information

- A defined variable should be used.
- Use type inferred variables at local level
- Try to use constant variables as much as possible
- Use _ when variable is not in use
- Declarations are bound to the scope in which they appear
- Use _ before the identifier of private variables
- Do not prefix variables

Built-in Types

void, Null, bool, num (int, double), String

Operators and its Types

- Arithmetic Operators
- Increment and decrement operators
- Equality and Relational Operators
- Type Test Operators
- Logical Operators
- Assignment Operators
- Bitwise Operators
- Cascade notation, spread operator, conditional operators

Arithmetic operators

Dart supports the usual arithmetic operators, as shown in the following table.

Operator	Meaning
`+`	Add
`-`	Subtract
`-expr`	Unary minus, also known as negation (reverse the sign of the expression)
`*`	Multiply
`/`	Divide
`~/`	Divide, returning an integer result
`%`	Get the remainder of an integer division (modulo)

Dart also supports both prefix and postfix increment and decrement operators.

Operator	Meaning
`++var`	`var = var + 1` (expression value is `var + 1`)
`var++`	`var = var + 1` (expression value is `var`)
`--var`	`var = var - 1` (expression value is `var - 1`)
`var--`	`var = var - 1` (expression value is `var`)

Equality and relational operators

The following table lists the meanings of equality and relational operators.

Operator	Meaning
==	Equal; see discussion below
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Type test operators

The `as`, `is`, and `is!` operators are handy for checking types at runtime.

Operator	Meaning
`as`	Typecast (also used to specify library prefixes)
`is`	True if the object has the specified type
`is!`	True if the object doesn't have 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.

Logical operators

You can invert or combine boolean expressions using the logical operators.

Operator	Meaning
`!expr`	inverts the following expression (changes false to true, and vice versa)
`||`	logical OR
`&&`	logical AND

Here's an example of using the logical operators:

```
if (!done && (col == 0 || col == 3)) {
  // ...Do something...
}
```

Assignment operators

As you've already seen, 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;
```

Compound assignment operators such as += combine an operation with an assignment.

=	*=	%=	>>>=	^=
+=	/=	<<=	&=	|=
-=	~/=	>>=		

Bitwise and shift operators

You can manipulate the individual bits of numbers in Dart. Usually, you'd use these bitwise and shift operators with integers.

Operator	Meaning
`&`	AND
`|`	OR
`^`	XOR
`~expr`	Unary bitwise complement (0s become 1s; 1s become 0s)
`<<`	Shift left
`>>`	Shift right
`>>>`	Unsigned shift right

Here's an example of using bitwise and shift operators:

Conditional expressions

Dart has two operators that let you concisely evaluate expressions that might otherwise require if-else statements:

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

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

Cascade notation

Cascades (`..`, `?..`) allow you to make a sequence of operations on the same object. In addition to function calls, you can also access fields on that same object. This often saves you the step of creating a temporary variable and allows you to write more fluid code.

Other operators

You've seen most of the remaining operators in other examples:

Operator	Name	Meaning
`()`	Function application	Represents a function call
`[]`	List access	Refers to the value at the specified index in the list
`.`	Member access	Refers to a property of an expression; example: `foo.bar` selects property `bar` from expression `foo`
`?.`	Conditional member access	Like `.`, but the leftmost operand can be null; example: `foo?.bar` selects property `bar` from expression `foo` unless `foo` is null (in which case the value of `foo?.bar` is null)

Conditional Execution

If else Statement:

- Starts with the keyword **if**, and determines whether the **dependent block** gets executed based on Boolean satisfy-ability of the **controlling expression(s)**.
  - Controlling expression(s) is/are a or a series of conditions
  - Dependent block is the block of code which is either executed or skipped based the Boolean satisfy-ability of the controlling expression(s)
- If block can be used alone or followed either by:
  - dependent bloc prefixed with **else,** leading to formation of a **selection statement**.

Conditional Execution

Single if else Statement Syntax:

```
If ( Controlling Expression(s) ){
    Dependent Block Statements;
}
```

Simple if else Statement Syntax:

```
If ( Controlling Expression(s) ){
    Dependent Block Statements;
} else {
    Dependent Block Statements;
}
```

Conditional Execution

Complex if else Statement Syntax:

```
If ( Controlling Expression(s) ){
    Dependent Block Statements;
} else if ( Controlling Expression(s) ){
    Dependent Block Statements;
} else if ( Controlling Expression(s) ){
    Dependent Block Statements;
} else {
    Dependent Block Statements;
}
```

Conditional Execution

Switch Statements (multiple selection):

```
switch ( Controlling Expression(s) ) {
    case  value1:
        Dependent Bloc;
        [break];
    case  value2:
        Dependent Bloc;
        [continue];
    case  value3:
        Dependent Bloc;
    case:
        Dependent Bloc;
}
```

Vocabulary

- Variable
- Function
- Control Flow Statement
- Class
- Object
- Declaration
- Assignment
- Definition
- Abstraction
- Algorithm
- print()
- stdin.readLineSyn()
- stdout.write()
- Comment
- num
- Double
- int

- Theory of Computation
- Computibility Theory
- Computational Thinking
- Problem Solving
- Complexity Theory
- Decomposition
- Pattern Recognition
- Everything is number
- null
- bool
- String
- Real Numbers
- Integers
- Whole Numbers
- Natural Numbers

- Rational Numbers
- Irrational Numbers
- Imaginary Numbers
- Operators
- Special/Key or Reserved Words
- Interpolation
- concatenation
- Data Type
- void main(){}
- Return
- library/package/plugin
- Routine/Procedure/Program
- Subroutine/sub-procedure/sub-program
- assignment

- Initialization
- void
- Memory address
-
-
-
-
-
-
-