JAVA VARIABLES
(Primitive Data Types In Depth)

* What is a variable?
  - It is a container which holds a value
  - How to declare?
    - Datatype Variable Name = value;

      Eg: int x = 1;

      boolean myz = True;

      Datatype — variable name — value

* Java is Static typed language i.e we mandatorily have to define the datatype of a variable.

* Java is a Strongly Typed Language i.e there is a restriction on what value can be assigned to a variable.

⇒ Variable Naming Convention:-
  - Variable name is case sensitive.
  - Variable name can be any legal identifier means can contain Unicode letters & Digits
  - Variable name can start with $, _ (underscore) and letter.
  - Variable name cannot be Java reserved Keyword like "new", "class", "while", "for", "interface", "int", "float" etc.
  - Variable should be small if it contains only 1 word else camel case should be followed.
  - For constant, variable name should be defined in Capital Letters.

Types Of Variables

- **Primitive Type**
  - Char
  - byte
  - short
  - int
  - long
  - float
  - double
  - boolean

  Integral Type: char, byte, short, int, long

  Fractional Type: float, double

- **Non-Primitive / Reference Type**
  - Class
  - interface
  - array
  - string
  - enum

So, there are 8 types of primitive variables :-

1) Char
   - 2 bytes i.e 16 bits
   - Character representation of ASCII values
   - Range : 0 to 65535 i.e. '\u0000' to '\uffff'
   - Default value is '\u0000' i.e NUL

2) byte
   - 1 byte i.e 8 bits
   - Signed 2^s complement
   - Range : -128 to 127

- default value is 0

0 _ _ _ _ _ _ _

↙

positive

1 _ _ _ _ _ _ _

↙

negative

2nd complement = Complement + 1

Let's take an example of 3

So +3 is 0 0 1 1

Now -3 is second complement of +3

i.e +3 0 0 1 1

1 1 0 0 (Complement)

+ 1

1 1 0 1 (2nd Complement)

So if we add +3 & -3

+3	0	0	1	1
-3	1	1	0	1
	0	0	0	0

ie. 0

So in total 7 bits represent number &

last bit represent sign.

3) Short

→ 2 bytes i.e 16 bits

→ Signed 2nd complement

- Range : -32768 to 32767
- Default value is 0

4) int
- 4 bytes ie 32 bits
- Range : -2^{31} to $2^{31}-1$
- Default value is 0
- Signed 2nd Complement

5) Long
- 8 bytes ie 64 bits
- Signed 2nd complement
- Range : -2^{63} to $2^{63}-1$
- Default value is 0
- Eg:- long var = 100l;

  → This l signifies that it is long type.

* We'll discuss fractional type in detail later.

6) boolean
- 1 bit
- Value : True or False
- Default Value is True

* Types of conversion :-

1) Widening / Automatic Conversion

- Automatic conversion when we go from lower data type to higher datatype

byte (1 byte)
short (2 byte)
int (4 bytes)
long (8 bytes)

Eg:- int var = 10;
long varLong = var; // Automatically converted int to long

2) Narrowing / Downcasting / Explicit Conversion

- It is opposite of widening ie going from higher data type to lower datatype.
- In this case downcasting doesn't happen automatically. So, we have to manually do it.

Eg:- int integerVariable = 10;
byte byteVariable = (byte) integerVariable;

- If we're downcasting beyond range then it'll again reset to -128 & it goes on.

So if integerVariable's value is 128 then byteVariable's value will be -128 (Next after 127 is -128 for byte)
& if it's 148 then byteVariable's value will be -108

3) Promotion during expression

- This happens internally during expression
- As soon as value of expression crosses the range of the datatype then promotion happens internally to higher datatype.
- byte & short promotes to int.
- Eg:- byte a = 1

  byte b = 127

  byte sum = a+b; // won't work since rang is crossing

So we'll have to declare it as int as the result will be int.

Although we can explicitly downcast it but value will change as per explicit downcasting terms.

↳ This is kla explicit casting during expression.

- In an expression, if one datatype is of higher datatype, then all other will also be automatically converted to higher data type.

Eg:- int a = 34;

double doubleVar = 20d;

int sum = a + doubleVar; // give error

double sum = a + doubleVar;

⇒ Kind Of Variables :-

* Member / Instance Variable

- It is a variable of the class i.e is created when an object of the class containing it is created. So, each object of the class has its individual copy of member variable.

* Local Variable

- These variables are the variables that are defined inside a method.
- If the method finishes, it gets destroyed.

* Static / Class Variable
- Only one copy of static / class variable exists. All objects can access it using class name.

* Method Parameters
- These are the variables that are passed to a method.

* Constructor Parameters
- These are the variables that are passed to a constructor.

⇒ Fractional Types

* How float & double are stored in memory?
  - Float

1 Bit	8 Bits	23 Bits
Stores sign ie 0 → positive, 1 → negative	Stores exponent	Stores Mantissa / Significant

Eg :- 4.125f

Step - 1 : Convert to binary

4 → 100 & .125 = 0.125 × 2 = 0.25 → 0

.25 × 2 = 0.5 → 0

.5 × 2 = 1.00 → 1

0.125 = 0.001

So binary equivalent of 4.125 = 100.001

Step-2 : Make it in the form of $(1.xxx) \times 2^{\text{exponent}}$

$100.001 \Rightarrow 1.00001 \times 2^{2}$

(Since base is binary, moving 2 decimal left adds 2 as power of 2)

Step-3 : Add bias to the exponent

$100.001 \Rightarrow \underbrace{1.00001}_{\text{Mantissa}} \times 2^{2 \rightarrow \text{exponent}}$

Since bias for float is 127 we'll add it to exponent
ie $127 + 2 = 129$

Step-4 Fill the numbers

$0 \; \underbrace{10000001}_{} \; \underbrace{00001}_{} 000 \cdots 0$

Sign bit — Binary of 129 ie exponent — Since Mantissa is 00001 rest bits are empty.

Now let's convert this binary form back to float

$(-1)^{\text{sign}} \times (1 + \text{mantissa}) \times 2^{\text{exponent} - 127}$

This is reverse of the above process

$(-1)^{0} \times (1 + 00001) \times 2^{129-127}$

$1 \times (1 + 2^{-5}) \times 2^{2}$

$1 \times (1 + 0.03125) \times 4$ ie $\underline{4.125}$

Now for 0.7f

Step-1

$0.7 \times 2 = 1.4 \quad 1$

$.4 \times 2 = 0.8 \quad 0$

$.8 \times 2 = 0.6 \quad 1$

$.6 \times 2 = 0.2 \quad 1$

$.2 \times 2 = 0.4 \quad 0$

$.4 \times 2 = 0.8 \quad 0$

$.8 \times 2 = 0.6 \quad 1$

⋮

Keeps repeating

So binary is 0.10110 0110 0110 0110

Step-2 : It becomes $(1.0110\ 0110\ 0110) \times 2^{-1}$

-1 since decimal shifted 1 pos^n to right

Step-3 : $-1 + 127 = 126$ (Add 127 bias)

Step-4: Now assemble / store

$$\underset{\text{Sign}}{0}\ \underbrace{01111110}_{\text{Exponent}}\ \overset{2^{-1}\ -2\ -3\ -4\ -5\ \cdots}{\underbrace{0110\ 0110\ 0110\ 0110\ 0110\ 011}_{\text{Mantissa}}}$$

& if we revert back to float

$$(-1)^0 \times (1 + \text{mantissa}) \times 2^{126-127}$$

$$\text{Mantissa} = \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^6} + \frac{1}{2^7} + \frac{1}{2^{10}} + \frac{1}{2^{11}} + \frac{1}{2^{14}} + \frac{1}{2^{15}} + \frac{1}{2^{18}} + \frac{1}{2^{19}} \cdots$$

So mantissa will be 0.399414062

so $1 \times (1 + 0.399414062) \times 1^{-1}$

$= \underbrace{0.699707031}$

So here value comes out to be less than 0.7 when we stored, so we generally use Big Decimal instead of float

* This is same for double as well, just that double is 64 Bits