

⑦ Data types and Incrementation, Decrementation

→ Nitin Sir

02/11/22
operator :-

→ Numeric Datatype :-

① Whole number :-

① byte (1 byte)

② short (2 bytes)

③ int (4 bytes) → Commonly used datatype

④ long (8 bytes)

② Real number :-

① float

② double

① float :-

Size of float = 32 bits (4 bytes)

Max Value of float = 3.4028235×10^{38}

Min Value of float = 1.4×10^{-45}

→ `S.O.P("Size" + Float.SIZE);`

`S.O.P("Max" + Float.MAX_VALUE);`

`S.O.P("Min" + Float.MIN_VALUE);`

Eg - `float a = 10.5;` → int

↳ Incompatible types

→ Compiler treats as double by default.

↳ Possible lossy Conversion from double to float

→ to treat it ^{as} float we have use 'f' (or 'F')

Note:- by default if you specify any real number / decimal number compiler will treat it as 'double', to specify to the compiler to treat it as float, we need to suffix it with 'f' (or 'F').

Eg ① float a = 10.5; → compiler error
↳ possibly loss of precision
float b = 10.5f; ✓
float c = 25.5F; ✓

② double data type:-

Size of double = 64

Max value of = 4.96×10^{32}

Min value = $1.7976931348623157 \times 10^{-308}$

Eg double d = 23.567;

→ Compiler treats real number by default as double

→ double d = Max value of double;

Note:- Datatypes are actually represented to the compiler and JVM using reserve words.

→ reserve words are normally in "lower case".
→ To map primitive data as object in Java from JDK 1.5 Concept of "wrapper class" was introduced.

Eg.

Primitive	Class
byte	→ Byte
short	→ Short
int	→ Integer
long	→ Long
float	→ Float
double	→ Double

③ Character data type:-

③ Character data types [Char] \rightarrow Hyder Sir

Why data type?

- \rightarrow to store data.
- \rightarrow to optimize data.
- \rightarrow to store real world data.
- \rightarrow these data types will follow specific format behind the scenes. that format will help to store the data in the form 0's and 1's in the memory.

\rightarrow int \rightarrow follows Base 2 format.

Char - Primitive data type.

\rightarrow to store characters like A, B, #, C etc.

$$2^1 = 2$$

A	0
B	1

$$2^2 = 4$$

A	00
B	01
C	10
D	11

$$2^3 = 8$$

000	A
001	B
010	C
100	D
101	E
110	F
011	G
111	H

\rightarrow As the no. of characters increasing then number of bits also increased.

$$2^1 = 2 \Rightarrow 2 \text{ char} \rightarrow 1 \text{ bit}$$

$$2^2 = 4 \Rightarrow 4 \text{ char} \rightarrow 2 \text{ bits}$$

$$2^3 = 8 \Rightarrow 8 \text{ char} \rightarrow 3 \text{ bits}$$

\rightarrow Americans discovered 128 characters.

Eg A B ... Z _ @ . a b i ... 3 # ...

\rightarrow for all these 128 characters they have given binary representation, Decimal, Hexa decimal representation.

$$2^7 = 128$$

Memory 7 bits + 1 bit

\rightarrow add to standardization of Memory.

ASCII

$$1 \text{ byte} \Rightarrow 8 \text{ bit}$$

Character \rightarrow Binary \rightarrow 00000000

Decimal or Hexadecimal

They given this representation for characters

Unicode

\rightarrow 2000 \rightarrow they told to find all characters of different languages

ASCII

$$2^{16} / 65536 \rightarrow \text{character / symbol}$$

\rightarrow They give

→ UTF-16
 $2^{16} \Rightarrow 65536$

16 bits \Rightarrow 2 bytes

→ UTF-16 Unicode Transformation format
→ 2 bytes of Memory allocated for character

→ Java follows UTF. 2 bytes of memory allocated character.

ASCII - they given Binary, Decimal and Hexadecimal for 128 characters (2^7)

Unicode - they given Binary, decimal and Hexadecimal for 65536 (2^{16}) Characters.

→ for unicode and ASCII 0 to 128 characters are same. (decimal representation)

→ Java follows UTF. (unicode)
↳ 2 bytes of Memory allocated character.

Syntax :-

Char a = 'A'

→ Character in java is within single quote. !!

Char Syntax

Char a = 'A'

→ Character in java is within single quote
, ,

Char a = "A"; (not valid)

Char a = 'AB'; (invalid)

→ it should be single character and single quote.

Eg char a = 'A';
char b = 'i';

→ for char the class is Character.

→ In other programming languages string and array ~~also~~ also treated as data types.

→ In java array and string treated as object.

→ Java is impure object oriented Language.

↳ because we primitive datatypes

→ 100% Pure object oriented when we use wrapper classes.

→ Alternative for primitive data type is wrapper class. Where everything is treated as object.

④ Truncation / Rounding zero:-

① `int a = 25;` ✓
`int b = 2;` ✓
`int c = a/b;` → $25/2 \Rightarrow 12.5$
`S.O.P(c);` → $o/p \Rightarrow 12$

② `int a = 25;`
`int b = 2;`
`float c = a/b;` → $25/2 \Rightarrow 12.5$
`S.O.P(c);` → $o/p = 12.0$

⇒ Truncation is happening

→ $\text{int} / \text{int} = \text{int}$.

→ The Result is Integer when we perform operation between ~~8 and 1~~ Integers irrespective of where it is stored.

→ 0.5 → Truncated (on Rounding to zero) getting

eg: ③ `double a = 25;`
`double b = 2;`
`double c = a/b;` → $25/2 = 12.5$
`S.O.P(c);` → $o/p = 12.5$

check - $a=0, b=2, o/p=0.0$

$a=0, b=0, o/p = \text{NaN}$

↳ not a number

Formats followed by Data types:-

→ byte - 1 bytes
short - 2 bytes → follow base 2 format
int - 4 bytes
long - 8 bytes
float - 4 bytes
double - 8 bytes → follows IEEE Single and double Precision format

Char - ~~2~~ 1 bytes → follows UNICODE

eg: $2 \overline{) 45}$
 $2 \overline{) 22-1}$
 $2 \overline{) 11-0}$
 $2 \overline{) 5-1}$
 $2 \overline{) 2-1}$
 $1-0$

45 stored in binary.
 101101
 $32 \ 8 \ 4 \ 1$

→ IEEE is more efficient
→ Whatever long takes 8 bytes to store something float takes 4 bytes for same purpose.

⇒ Why ~~Itt~~ Single and double precision format is efficient?

Ⓐ What ever ~~long~~ takes 8 bytes to store something float takes 4 bytes ~~at~~ for same purpose

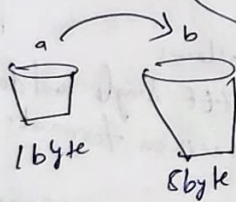
eg ①

byte a = 45;

double b;

b = a; → assignment operator.
What ever there is right side given left side.

so: P(b); → 45.0



nature of ~~double~~ double.

→ ~~type of data~~

③ Type Casting / Numeric Promotion 04/11/22

→ Changing type of data from one type to another type is called type casting / Numeric Promotion.

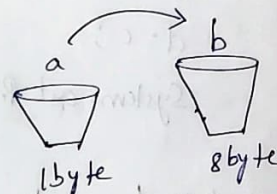
eg byte a = 45;

double b;

b = a;

so: P(b); → 45.0

↳ nature of double



→ Converting data from one type to another type. (automatic) without any efforts (or) automatically is called implicit type casting

→ implicit type casting

↳ internally

↳ automatic

↳ behind the scene ~~at~~

Implicit type conversion possible

eg

→ byte - short - int → long → float → double

Eg ①: double c = 45.5;
byte d;

d = c;

System.out.println(d); → Compilation Error.

⑥ Explicit type Casting / Narrowing type Casting :-

→ Converting a higher data type into a lower one is called narrowing type casting. It is also known as Explicit Conversion (on narrowing).

Eg ①: double a = 45.5;
byte b;

b = (byte) a;

System.out.println(b); → 45

⇒ ~~Value after decimal~~ - precision

→ Value after decimal - precision

→ o/p: 45.5
x
→ loss of precision might occur.

Eg ②: byte ab = 10;
byte ac = 20;

byte result = ~~ab~~ ab * ac;

S.o.p (result) → Error

↳ Cannot convert from
into to byte.

Eg ③: ~~byte~~

byte ab = 10

byte ac = 20

int result = ab * ac;

S.o.p(result) → o/p: 200

⑥ operator:-

operators are used to perform operations on Variables and Values.

Eg ① `int a = 10 + 5;` $\rightarrow 15$

(i) Incrementation:- Increasing Existing Value by 1.

Eg ① `a++` \rightarrow incrementation

`int a = 5;`

`a = a + 1;` $\rightarrow 5 + 1 = 6$

`s.o.p(a);`

Alt

`int a = 5`

`a++;`

`System.out.println(a);` $\rightarrow (a=6)$

(ii) Decrementation:- decreasing Existing Value by 1.

Eg ① `int a = 5;`

`a = a - 1` (or `a--;`)

`s.o.p(a);`

(i.a) Pre and Post Incrementation

`a++` \rightarrow Post incrementation

`++a` \rightarrow Pre incrementation.

Eg ① `int a = 5;`

`s.o.p(a);` \rightarrow o/p: 5

`a++;`

`s.o.p(a);` \rightarrow o/p: 6

`int a = 5;` o/p: 5

`s.o.p(a);` \rightarrow 5

`++a;` o/p: 6

`s.o.p(a);` \rightarrow 6

Eg ②

`int a = 5;`

`int b;`

`b = a++;`

`s.o.p(a);` $\rightarrow 6$

`s.o.p(b);` $\rightarrow 5$

`int a = 5;`

`int b;`

`b = ++a`

`s.o.p(a)` $\rightarrow 6$

`s.o.p(b)` $\rightarrow 6$

`a` $\xrightarrow{++}$ `5` `6`

`b` `5`

`a` `5` `6`

`b` `6`

\rightarrow Post incrementation
a is stored in b
and incremented
after that

Eg ③

⑥ Eg-

int a=5; a 5 6 7 8

int b; b 19

$$b = \underbrace{a}_{\text{Post}}++ + \underbrace{a}_{\text{Post}}++ + \underbrace{a}_{\text{Pre}}++$$

(i) S.o.p(a);

S.o.p(b);

Eg 3

int a=5; a 5 6 7 8

int b; b 19

$$b = \underbrace{a}_{\text{Post}}++ + \underbrace{a}_{\text{Post}}++ + \underbrace{a}_{\text{Pre}}++$$

5 + 6 + 8 = 19

S.o.p(a); $\rightarrow a=8$

S.o.p(b); $\rightarrow b=19$

(i)

b = a++	b = ++a
↳ use it store it here it	increment it store it

Eg (ii) int a=5 a

int b; b

b = ++a + a++ + ++a + a--;

S.o.p(a)

S.o.p(b)

Eg (ii)

int a=5; a 5 6 7 8

int b; b 28

$$b = \underbrace{(+a)}_6 + \underbrace{(a++)}_6 + \underbrace{(++a)}_8 + \underbrace{(a--)}_{+8} \rightarrow 28$$

S.o.p(a); $\rightarrow 7$

S.o.p(b); $\rightarrow 28$

Eg (iii) int a=5, b;

$$b = \underbrace{(a++)}_5 + \underbrace{(--a)}_5 - \underbrace{(a--)}_4 - \underbrace{(a++)}_4$$

S.o.p(a); $\rightarrow 4$

S.o.p(b); $\rightarrow 1$

a 4

b 1