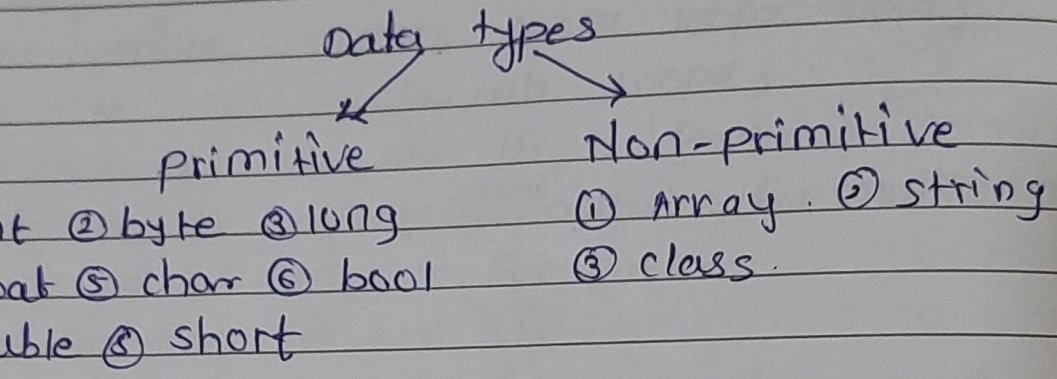
① How Java works?
Java is compiled into the byte code and then it is interpreted to machine code

Naming conventions

① For classes we use pascalconvention

② For functions we use camelCase convention.

* Data types



① int ② byte ③ long
④ float ⑤ char ⑥ bool
⑦ double ⑧ short

① Array ② string
③ class.

* Variable
variable is like container that ~~ut~~ store a value.
Ex

```
int num = 8;
```

int → datatype, num → variable name, 8 → value it stores.

① Primitive data type

① byte → value ranges from -128 to 127
→ Takes 1 byte
→ Default value is 0.

② short → value ranges from $-(2^{16})/2$ to $(2^{16})/2-1$
→ Takes 2 byte
→ Default value 0.

③ int → range $-(2^{32})/2$ to $(2^{32}/2-1$
→ Takes 4 byte
→ default 0.

④ float → range (see docs)
→ Take 4 byte
→ default 0.0f

⑤ long → value range ~~-8~~ $-(2^{64})/2$ to $(2^{64})/2-1$
→ 8 bytes
→ default 0.

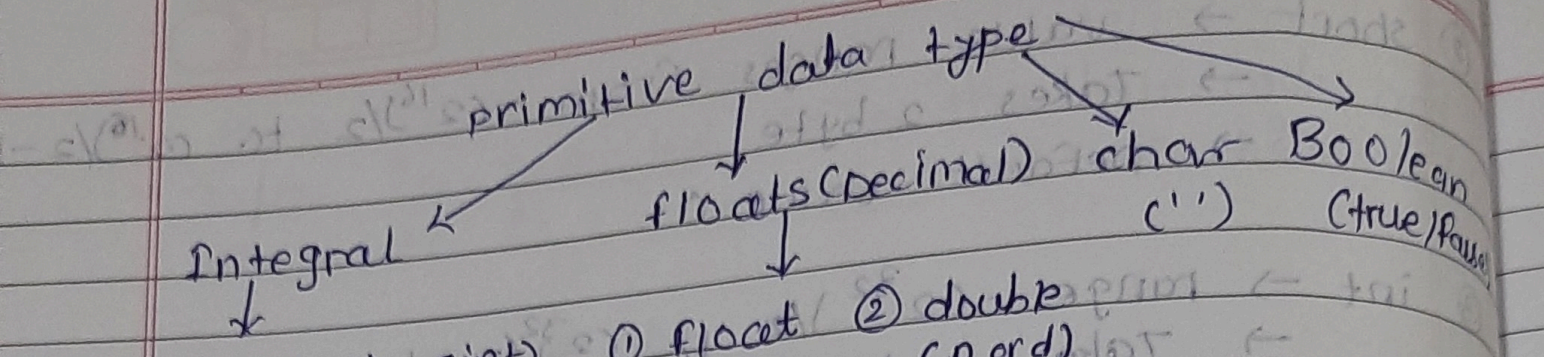
⑥ double → range (see docs)
→ takes 8byte
→ default 0.0d

⑦ char → range 0 to 65535 $(2^{16}-1)$
→ 2 byte – because it support unicode.
→ default 0.000

⑧ boolean → value can be true or false.
→ size depends on JVM
→ default value false.

1 byte = 8 bits

$-2^{32}/2$ $4 \times 8 = 32$



* Literals:
A constant value which can be assigned to the variable is called literal.

101 → Integral literal.
10.1f → float
10.1 → double
'A' → character Literal
true → boolean literal
"Harry" → string literal

* Scanner

```
import java.util.scanner;

Scanner sc = new Scanner(System.in)
                          ↳ keyboard reader data

Scanner sc =
    int a = sc.nextInt(); → int.
```

* operators & Expression:

Types of operators

① Arthmatic → +, -, *, %, /, ++, --
② Assignment → = , +=
③ comparision → ==, >=, <=
④ logical → &&, ||
⑤ Bitwise → &, |
⑥ * Modulo operator → %. 6 % 4 = 2 → remainder

operator
6 + 4 = 10
operand

* Associativity – Left to right for * and /.

① int a = 6*5 - 3412 → higher precedence is *
= 30 - 3412
= 30 - 17
= 13

② int b = 6015 - 3412 → Left to right Asso.
= 30 - 3412
= 30 - 17
=

* */ = L to R
+- = L to R
++, = → R to L

* Resulting data type after arithmatic operation.

R = b+s → int --→ b = byte + sort.
R = s+i → int
R = long + float → float
R = char + int → int
R = char + short → int
R = long + double → double
R = float + double → double

* Increment and decrement operators.

① Increment

i++ → print ∴ then increment.
++i → increment ∴ then print.

② Decrement

i-- → print ∴ then decrement
--i → decrement and then print.

* string.

string is like a class. in Java.

```
String name;
name = new string ("Rasika");
// string name = "Rasika"; → valid.
```

* different ways to print in Java:

① system.out.print() → No new line at the end.
② system.out.println() → new line at the end.
③ system.out.printf()

④ system.out.format ()

* system.out.printf ("%c", ch);

%d → for int
%f → for float
%c → for char
%s → for string.

* string Methods:

String name = "Rasika";
0 1 2 3 4 5

① name.length() → Return length of string name
Ex: String name = "Rasika";
int vallue = name.length();
System.out.println (vallue);
O/P ⇒ 6

② name.toLowerCase () → Return new string which has all the lowercase characters from the string name
Ex:

String a = "Rasika";
S.O.println (a);
int value = a.length ();
S.O.P. (value);
String lstring = a.toLowercase ();
system.out.println (lstring);

O/P ⇒ rasika Suppose: RASIKA
O/P ⇒ rasika.

④ name.toUppercase() → Return new string which has all the uppercase characters from the string name.

Ex: string a = "Rasika";
string Ustring = a.toUppercase();
system.out.println(a);

O/P ⇒ RASIKA.

③ Trim name.trim ⇒ Return a new string after removing all the leading and trailing space from the original string.
string nonTrimmedstring = "---- Rasika ----";
system.out.println(nonTrimming string);

string O/P ⇒ ----Rasika----

string trimmedstring;
system.out.println(non Trimmed string.trim());

O/P ⇒ Rasika ⇒ NO space.

⑤ name.substring(int, ~~short~~ start) ⇒ return a substring From start to end

Ex: if int string a = "Rasika";
system.out.println(name.substring(2));

ex O/P ⇒ sika

⑥ name.substring(int start to intend); Return Starting from start index to the end index.

ex: String a = "Rasika";
0 1 2 3 n-1

System.out.println(a.substring(1, 4));
start index, ending index

O/P: asi ⇒ 1 to 3 index printed.

⑦ name.replace('r', 'p') → Return a new starting after replacing r with p. Pasika is return in this case:

ex: string a = "Rasika";

System.out.println(name.replace(old char: 'R', new char: 'P'));

O/P ⇒ Pasika:

system.out.println(name.replace(target: "Ra", replacement: "Pra"));

O/P ⇒ Prasika.

⑧ name.startsWith("Ra") → Return true if name starts with string Ra.

ex: String a = "Rasika";

S.O.P(name.startsWith("Ra")); (ra) ⇒ false.
O/P ⇒ true

(9) S.O.P (a.endswith("ka")); } name.endswith("ka")
O/P ⇒ True.

(10) ~~char~~name.charAt(2) ⇒
Return character at a given index Position. s in this case.

ex : string a = "Rasika";

S.O.P(a.charAt(2));
O/P ⇒ s. (0 1 2 3 4 5 Rasika)

(11) name.index of (s) → the index of the given string
str

for ex: name.index of ("ar") returns 1 which is the 1st occurence of ar in string "Harray",

ex: string a = "Rasika";

S.O.P(a.index of("ika"));

O/P = 3.

S.O.P(a.index of (str: "Ras", from index: 4));
string form Ras ~~end~~ index 4.
from

(12) name.index of ("s", 3) = Harry.

(13) name.last index of ("r") → return the last index of the given string. in this case.

ex:
string a = "Rasika";
S.O.P(a.lastIndex (str: R)) O/P = 0

(14) name.lastIndexOf("r",2) → return the last index of the given string before index 2.

ex: Rasika
S.O.P(a.lastIndexOf(str:"i", fromIndex:3));

O/P => 3.

(15) name.equals("Rasika") =>
returns true if the given string is equal to "Rasika" false otherwise.

ex: a="Rasika";
S.O.P(a.equals("Rasika"));
S.O.P(a.equals("rasika"));
O/P => True
O/P => false

(16) name.equalsIgnoreCase =>
return true if two strings are equal ignoring the case of characters.

ex: a="Rasika";

S.O.P(name.equalsIgnoreCase("rasika"));

O/P = True.

* Escape sequence characters:
sequance of character after backshash '\'
= Escape seq^n character.
S.O.P ("I am escape sequence "\" double quto");

S.O.P ("I am escape seq^n "\\ double quto");

init

* While loop:

ex: int i=1;
while (i<=3){
S.O.P (i);
i++;
}

O/P => 1, 2, 3

while (boolean cond^n)
{
} //statement

* Do-While loop:

do {
//code
} while (condition);

ex) int a=1;
do {
S.O.P (a);
a++;
} while (a<=10);

O/P => 1 to 10 no print.

* for loop:

```
for (intialize ; check expression, update) {
    //code;
}
```

ex:

```
for (int i=1 ; i<=10; i++) {
    s.o.p(i);
}
```

o/p ⇒ 1 to 10 no print.

* decrementing for loop

ex

```
for (int i=3 ; i>=0 ; i--) {
    s.o.p (i);
}
```

o/p ⇒ 3
2
1
0

```
for (int i=100 ; i>=1 ; i--) {
    S.o.p (i);
}
```

o/p ⇒ 100 to 100 print in reverse order.

* conditions:

① If-else ② switch case.

```
if (cond^-to-be-check) {
    statements-if-cond^-true;
}
else {
    statements-if-cond^-false;
}
```

ex:

```
int age = 18;
if (age >= 18) {
s.o.p ("you can drive");
}
else {"you can't drive");
  }
}
}
```

O/P ⇒ you can drive.

* Relational operators

==, >=, >, <, <=, !=

* Logical operators

&& → AND
|| → OR
! → NOT

① AND

Y && Y = Y
Y && N = N
N && Y = N
N && N = N

② OR

Y || Y = Y
Y || N = Y
N || Y = Y
N || N = N.

* else if

ex:

```
int age;
S.O.P("Enter your age");
Scanner sc = new scanner (system.in);
  age = sc.NextInt();

if (age > 56) {
    S.O.P(" you are experienced");
}

else if (age > 46){
    S.O.P(" you are semi-experienced");
  }

else if (age > 36){
    S.O.P (" you are semi semi -/- );
}
```

```
else {
    S.O.P("you are not experienced");

  }
}
}
```

O/P ⇒ Enter your age
20
you are not experienced.

* switch case:

```
ex:  int age;
     S.O.P("Enter your age");
     Scanner sc = new Scanner(System.in);
     age = sc.nextInt();
     switch(age){
     case 18:
     S.O.P("you are going to become adult");
       break;

     case 23:
       S.O.P("you are going to join a job");
       break;

     case 60:
       S.O.P("you are going to get retired");
       break;

default:
  S.O.P("Enjoy your life");
  }
}
```

O/P => Enter your age
23
you are going to join a job.

Array:

Array is a collection of similar type of data.

ex:
Storing marks of 5 students.

int [] marks = new int [5]

* Accessing Array elements

array elements can be accessed as

marks [0] = 100
marks [1] = 70
⋮
marks [4] = 98

① int [] marks; ⟶ declaration
marks = new int [5] → memory allocation.

② int [] marks = new int [5] ⇒ declaration + memory allocation.

③ int [] marks = {100, 70, 80, 71, 98} ⇒ declare + initize.

* array length:

```
int [] marks = { 98,45,79,99,80};

S.O.P(marks.length);
```

O/P => 5 → length of array.

* for each

```
int[] marks ={98, 90, 78, 87, 80 };
   for (int ele:marks){
S.O.P(ele);
  }
```

O/P => 98
90
78
87
80

2D Array:

```
int [][] flats = new int [2][3];
```

Java Method:

① Syntax of Method

```
data type name () {
    // Method body
}
```

Following method returns sum of two no.

```
int sum (int a, int b) {
    int c = a+b;
    return c;
}
```

② calling Method:

```
calc obj = new calc();  → obj creation
obj.sum (a,b);          → Method call upon obj.
```

③ void return type

when we Don't want our method to return anything we used void as the return type.

④ Static keyword:

⑤ Process of method invocation in Java

Consider the method sum;

```
int sum (int a, int b)
{
```

return a+b;
}

The method is called like this
calc obj = new calc();
c= obj.sum(2,3)

⑥ Method overloading.

void foo();
void foo(int a)
void foo(int a, int b)

⑦ Variable Arguments:

Public Static void foo(int ... arr)
{
// arr is avaible here as int[] arr
}

foo can be called with 0 or more arguments like this:

foo(7) foo(7,8,9) foo(1,2,7,8,9)

We can also create a fun^n bar like this

Public Static void bar(int a, int arr)
{
//code

bar can be called as bar(1), bar(1,2), bar(1,7 etc