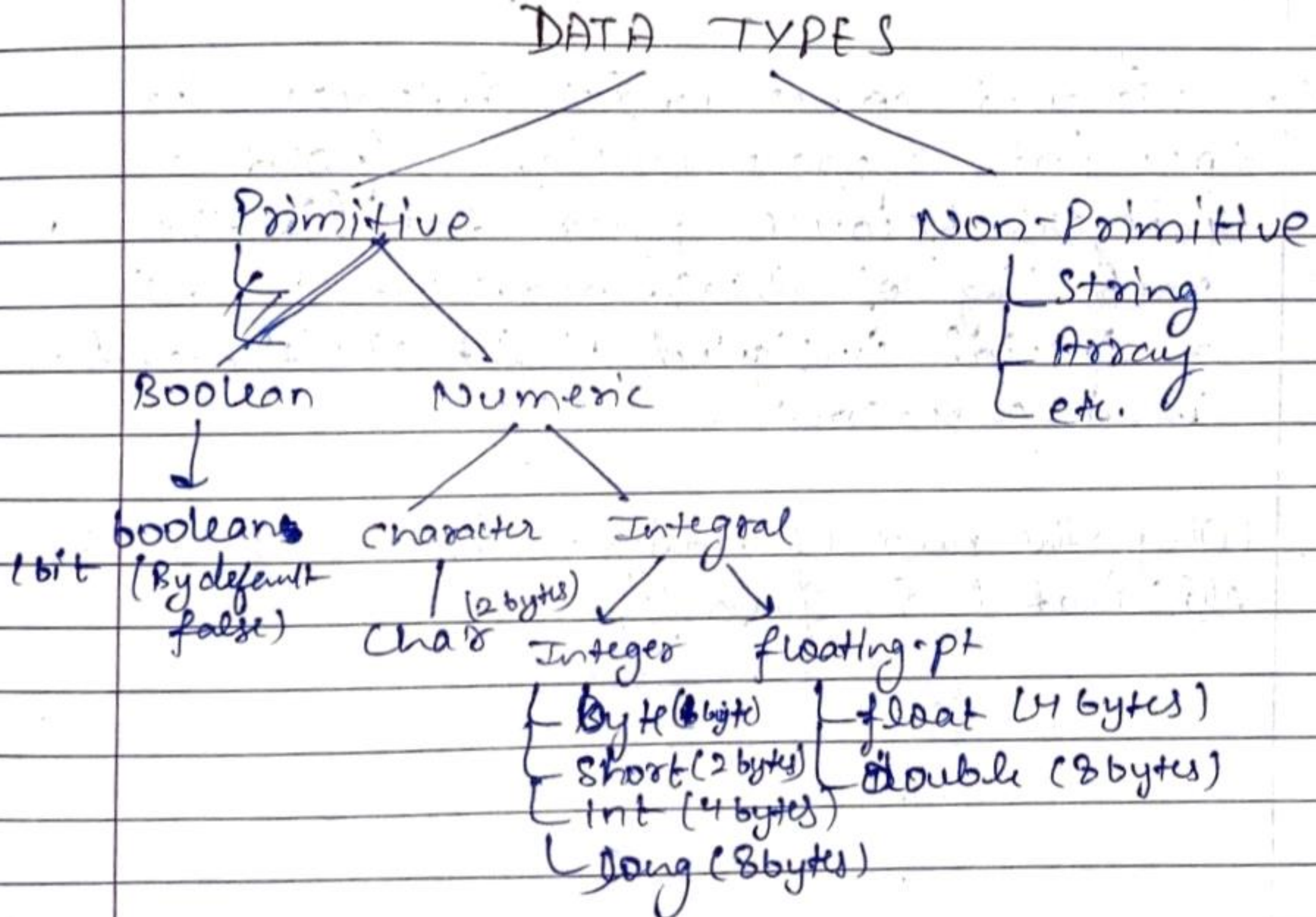
Ch-3

Data Types, Variables & Arrays

Java is a strongly typed language

Q Why Java is called so?

= No automatic coercions or conversions of conflicting types, because Java compiler checks all expressions and parameters before compiling to make sure that they are type compatible.



Data Type	Default Value	Default Size
boolean	false	1 bit
char	'\u0000'	2 bytes
byte	0	1 byte
short	0	2 bytes
int	0	4 bytes
long	0	8 bytes
float	0.0f	4 bytes
double	0.0d	8 bytes

Page No.
Date

Q Why all data types in Java have strictly defined range?
- All data types have strictly defined range due to Java's portability requirement.
- There should be trade-off b/w performance and portability i.e small loss of performance in some environm in order to achieve portability.

Notes: This point comes under integers, Java does not support unsigned positive only integers because java tackles the meaning of high-order bit by using a special unsigned right shift operator(>> ~~i.e the reason~~

Q Why the need of unsigned integer type is eliminated in java?

10:54

javanotes.pdf - Read-only

UNICODE-Universal International Standard Character Encodi

Page No.
Date 8|2|20

* Byte = -128 to 127
* Short = -32768 to 32767
* int = -2,147,483,648 to -2,147,487,647

Q1 why byte or short would not be effici then using int in cases when the larger range of int is not required?
When bytes & short values are used in an expression, they are promoted to int using the concept of type promo

* long = 64 bit

Floating point types

$$\begin{tikzcd} & \text{Floating point types} \arrow[dl] \arrow[dr] & \\ \text{float} & & \text{double} \end{tikzcd}$$

float
- single precision
- Faster, space is half as comp. to double

double
- double precision
- double precision is faster for high speed mathematical calculations like sine, cosine etc.

ffff.. → Highest possible range

Q2 why char uses 2 bytes and what is \u0000 ?
↳ It means lowest possible range.
Range of char is 0 to 65536

Here is a program that demonstrates **char** variables:

✓

```
// Demonstrate char data type.
class CharDemo {
  public static void main(String args[]) {
    char ch1, ch2;

    ch1 = 88; // code for X
    ch2 = 'Y';

    System.out.print("ch1 and ch2: ");
    System.out.println(ch1 + " " + ch2);
  }
}
```

This program displays the following output:

```
ch1 and ch2: X Y
```

Notice that **ch1** is assigned the value 88, which is the ASCII (and Unicode) value that corresponds to the letter *X*. As mentioned, the ASCII character set occupies the first 127 values in the Unicode character set. For this reason, all the "old tricks" that you may have used with characters in other languages will work in Java, too.

Although **char** is designed to hold Unicode characters, it can also be used as an integer type on which you can perform arithmetic operations. For example, you can add two characters together, or increment the value of a character variable. Consider the following program:

```
// char variables behave like integers.
class CharDemo2 {
  public static void main(String args[]) {
    char ch1;

    ch1 = 'X';
    System.out.println("ch1 contains " + ch1);

    ch1++; // increment ch1
    System.out.println("ch1 is now " + ch1);
  }
}
```

The output generated by this program is shown here:

```
ch1 contains X
ch1 is now Y
```

In the program, **ch1** is first given the value *X*. Next, **ch1** is incremented. This results in **ch1** containing *Y*, the next character in the ASCII (and Unicode) sequence

Booleans

Java has a primitive type, called **boolean**, for logical values. It can have only one of two possible values, **true** or **false**. This is the type returned by all relational operators, as in the case of **a < b**. **boolean** is also the type *required* by the conditional expressions that govern the control statements such as **if** and **for**.

Here is a program that demonstrates the **boolean** type:

```
// Demonstrate boolean values.
class BoolTest {
  public static void main(String args[]) {
    boolean b;

    b = false;
    System.out.println("b is " + b);
    b = true;
    System.out.println("b is " + b);

    // a boolean value can control the if statement
    if(b) System.out.println("This is executed.");

    b = false;
```

```
        if(b) System.out.println("This is not executed.");

        // outcome of a relational operator is a boolean value
        System.out.println("10 > 9 is " + (10 > 9));
    }
}
```

The output generated by this program is shown here:

```
b is false
b is true
This is executed.
10 > 9 is true
```

There are three interesting things to notice about this program. First, as you can see, when a **boolean** value is output by **println( )**, "true" or "false" is displayed. Second, the value of a **boolean** variable is sufficient, by itself, to control the **if** statement. There is no need to write an **if** statement like this:

```
if(b == true) ...
```

Third, the outcome of a relational operator, such as <, is a **boolean** value. This is why the expression **10>9** displays the value "true." Further, the extra set of parentheses around **10>9** is necessary because the + operator has a higher precedence than the >.

Integer literals
- → decimal (base 10) → can't have leading 0
- → octal (base 8) → represented by leading 0 eg 07
- → hexadecimal (base 16) → (0x or 0X)

NOTE (1) 09 will produce compiler error as 9 is outside of octal no. (0 to 7)

When a literal value is assigned to a byte or short variable, no error is generated if the literal value is within the range of the target type eg an integer literal can always be assigned to a long variable