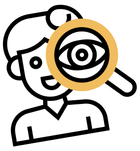
*Let us move ahead and learn about the next data type.*

## Character type data:

$ *1* ***X***

Had a look on the above characters?



Well they all fall under character type data but, computer doesn’t understand ***A@#*14***. It only understands 0’s and 1’s and hence conversion should happen. Before understanding that let us first know the different characters.
Let us consider four symbols ***A B C D*** but, none of these symbols can your computer understand because all it understands is binary numbers 0’s and 1’s. So, let us attach a binary code to each of these symbols like this

| SYMBOLS | CODE |
|---|---|
| **A** | **00** |
| **B** | **01** |
| **C** | **10** |
| **D** | **11** |

Let us consider if there were 8 symbols and look at its binary code.

| SYMBOLS | CODE |
|---|---|
| **A** | **000** |
| **B** | **001** |
| **C** | **010** |
| **D** | **011** |
| **E** | **100** |
| **F** | **101** |
| **G** | **110** |
| **H** | **111** |

As you can see, as the number of symbols increase their code size also increases.

Can you guess the code size for 16 characters?

| SYMBOLS | CODE |
|---|---|
| A | 0000 |
| B | 0001 |
| C | 0010 |
| D | 0011 |
| E | 0100 |
| F | 0101 |
| G | 0110 |
| H | 0111 |

| SYMBOLS | CODE |
|---|---|
| I | 1000 |
| J | 1001 |
| K | 1010 |
| L | 1011 |
| M | 1100 |
| N | 1101 |
| O | 1110 |
| P | 1111 |

If you are focusing then you might have noticed there is a mathematical relation between **no. of symbols** and **code length.**

Let us consider four symbols ☞ **A B C D**

| SYMBOLS | CODE |
|---|---|
| A | 00 |
| B | 01 |
| C | 10 |
| D | 11 |

→

*NO. Of Symbols*

**4**

↓

$2^2$

Similarly let us consider eight symbols ☞ **A B C D E F G H**

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

→

*NO. Of Symbols*

**8**

↓

$2^3$

If you are focusing on the **power of 2,** you can see it is only the **code length.**

Let us consider one more case of 16 symbols to understand this.

16 Symbols ☞ **A B C D E F G H I J K L M N O P**

| SYMBOLS | CODE |
|---|---|
| **A** | **0000** |
| **B** | **0001** |
| **C** | **0010** |
| **D** | **0011** |
| **E** | **0100** |
| **F** | **0101** |
| **G** | **0110** |
| **H** | **0111** |

| SYMBOLS | CODE |
|---|---|
| **I** | **1000** |
| **J** | **1001** |
| **K** | **1010** |
| **L** | **1011** |
| **M** | **1100** |
| **N** | **1101** |
| **O** | **1110** |
| **P** | **1111** |

***NO. Of Symbols***

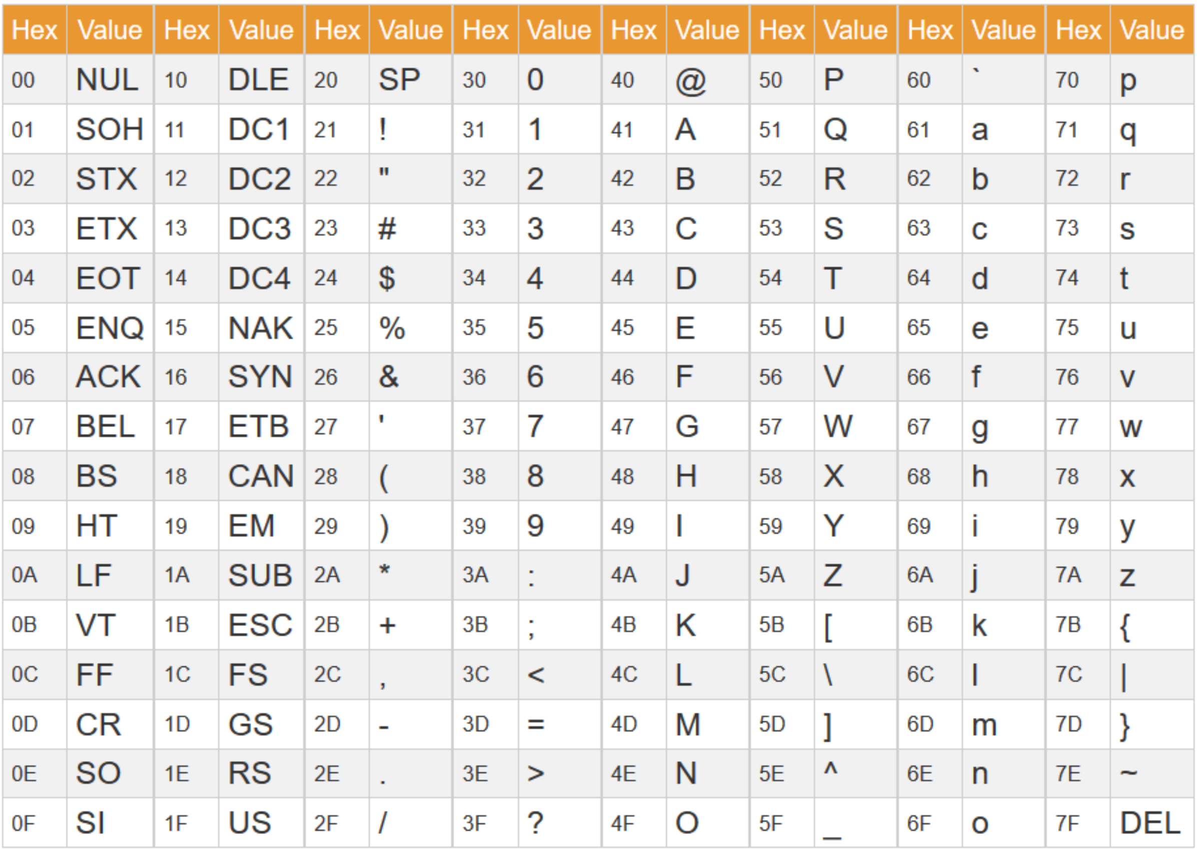
**16**

$\downarrow$

$\mathbf{2^{4}}$

**Code Length - 4**

Now you understood the relation between code length and number of symbols. Like this Americans found **128 symbols** and gave the name as **ASCII**. But Java does not follow ASCII as it only consist of English symbols. Have a look at the ASCII table below.



| Hex | Value | Hex | Value | Hex | Value | Hex | Value | Hex | Value | Hex | Value | Hex | Value | Hex | Value |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | NUL | 10 | DLE | 20 | SP | 30 | 0 | 40 | @ | 50 | P | 60 | ` | 70 | p |
| 01 | SOH | 11 | DC1 | 21 | ! | 31 | 1 | 41 | A | 51 | Q | 61 | a | 71 | q |
| 02 | STX | 12 | DC2 | 22 | " | 32 | 2 | 42 | B | 52 | R | 62 | b | 72 | r |
| 03 | ETX | 13 | DC3 | 23 | # | 33 | 3 | 43 | C | 53 | S | 63 | c | 73 | s |
| 04 | EOT | 14 | DC4 | 24 | $ | 34 | 4 | 44 | D | 54 | T | 64 | d | 74 | t |
| 05 | ENQ | 15 | NAK | 25 | % | 35 | 5 | 45 | E | 55 | U | 65 | e | 75 | u |
| 06 | ACK | 16 | SYN | 26 | & | 36 | 6 | 46 | F | 56 | V | 66 | f | 76 | v |
| 07 | BEL | 17 | ETB | 27 | ' | 37 | 7 | 47 | G | 57 | W | 67 | g | 77 | w |
| 08 | BS | 18 | CAN | 28 | ( | 38 | 8 | 48 | H | 58 | X | 68 | h | 78 | x |
| 09 | HT | 19 | EM | 29 | ) | 39 | 9 | 49 | I | 59 | Y | 69 | i | 79 | y |
| 0A | LF | 1A | SUB | 2A | * | 3A | : | 4A | J | 5A | Z | 6A | j | 7A | z |
| 0B | VT | 1B | ESC | 2B | + | 3B | ; | 4B | K | 5B | [ | 6B | k | 7B | { |
| 0C | FF | 1C | FS | 2C | , | 3C | < | 4C | L | 5C | \ | 6C | l | 7C | \| |
| 0D | CR | 1D | GS | 2D | - | 3D | = | 4D | M | 5D | ] | 6D | m | 7D | } |
| 0E | SO | 1E | RS | 2E | . | 3E | > | 4E | N | 5E | ^ | 6E | n | 7E | ~ |
| 0F | SI | 1F | US | 2F | / | 3F | ? | 4F | O | 5F | _ | 6F | o | 7F | DEL |

ASCII stands for **American standard code for information interchange.**
It is a **7-bit** binary representation for 128 symbols.
Java does not follow ASCII as it is a English biased language and does not support symbols of other languages.
Hence java follows **UNICODE** which provides binary representation for **65,536** symbols of commonly spoken languages across the world.
It is a **16-bit** code and hence a char variable in java takes **2 bytes** of memory.

Let us write a simple code to print character type data

```
Class Demo
{
        Public static void main(String[] args)
        {
                Char ch = ‘a’;
                System.out.println(ch);
        }
}
```

**Output: a**

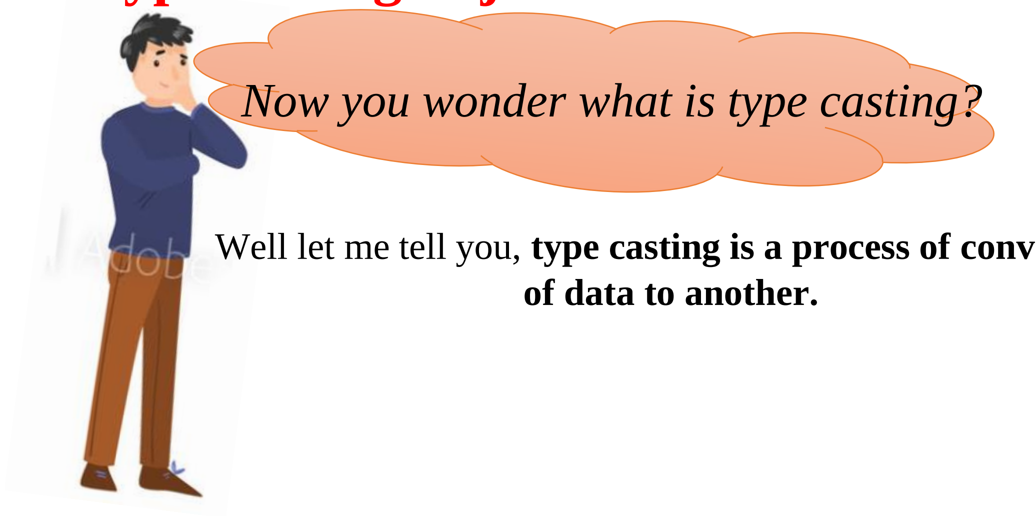
Let us now look at the last data type that is **boolean** data type.

## Boolean data type:

To store yes/no type data or true/false type data, java provides boolean data type. Size of this data type is decided by JVM and we have already learned JVM is platform dependent, hence the size of this data type will differ depending on the type of operating system.

**The remaining types of data that is audio, video and still pictures are handled using built in libraries**.

## Type casting in java:



Well let me tell you, **type casting is a process of converting one type of data to another.**

In Java, there are two types of casting:

**Implicit casting** (automatically) - converting a smaller type to a larger type size byte -> short -> char -> int -> long -> float -> double.

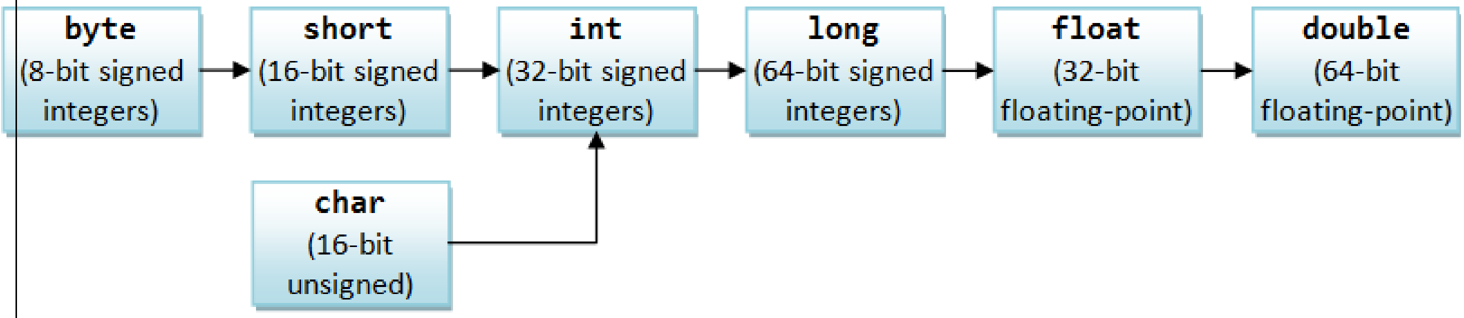
**Explicit casting** (manually) - converting a larger type to a smaller size type.

# Implicit type casting:

When a smaller data type is converted to a larger data type, the conversion is automatically performed by **the java complier** and is referred to as implicit type casting.

**Advantage**: No loss of precision.

Consider the **Implicit type casting chart** given below to understand this:


byte (8-bit signed integers)
short (16-bit signed integers)
int (32-bit signed integers)
long (64-bit signed integers)
float (32-bit floating-point)
double (64-bit floating-point)
char (16-bit unsigned)

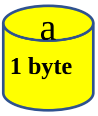
**Orders of Implicit Type-Casting for Primitives**

Let us consider a code snippet to understand this:

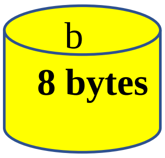
```
byte a = 45;
double b;
 b = a;
```

let us understand implicit type casting using the above code snippet

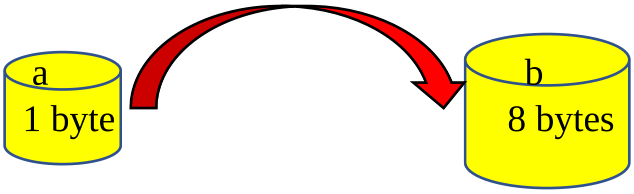
a is a variable of type byte whose size is 1 byte.



b is a variable of type double whose size is 8 bytes.



b = a; we are now trying to store the data present in **a into b.**
a is of type byte and can store 1 byte. b is of type double and can store 8 bytes.
we are trying to store data of smaller size into larger size.



This conversion is implicitly done without user interaction and hence it is referred to as implicit type casting.

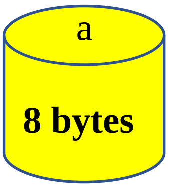
## Explicit type casting:

When a larger data type is converted to a smaller data type, the conversion **not automatically performed** by **the java complier** and must be done by **programmer explicitly** and hence it is referred to as implicit type casting.

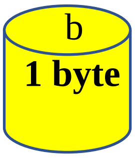
Let us consider a simple code snippet to understand this, the way we understood implicit type casting.

```
double a = 45.5;
byte b;
 b = a;  → error
```

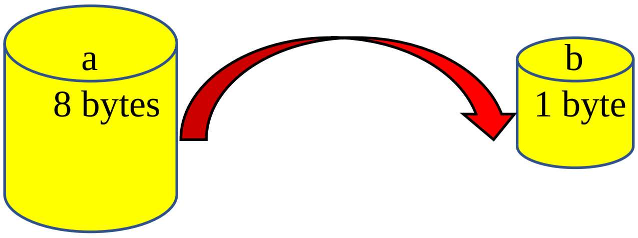
a is a variable of type double whose size is 8 bytes.



b is a variable of type byte whose size is 1 byte.



**b = a;** will give you ***error*** as you are trying to store a larger type of data into smaller type.



The above conversion will result in error as **loss of precision** occurs.
To get the error free output, we have to explicitly convert the data as shown below.

```
double a = 45.5;
byte b;
b = byte(a);
```

**b is of type byte and it will only store 45 and 0.5 is lost during the conversion which is the disadvantage of explicit type casting.**