LIMITS AND DATA SIZE
JAVA

Primitive data types in Java

Java has a small set of **primitive data** types. We could consider them fundamental, since most of the other types, the structured or complex types, are compositions based on these most basic ones. These primitive data types are used to handle the most basic types of information, such as numbers of various classes or data of type true / false (also known as "Boolean values" or simply "Boolean").

Integer numeric types

In Java there are four types intended to store integers. The name and characteristics of these types are as follows:

- byte: as its name denotes, it uses a single byte (8 bits) of storage. This allows you to store values in the range [-128, 127].
- short: uses twice the storage as above, which makes it possible to represent any value in the range [-32,768, 32,767].
- int: uses 4 bytes of storage and is the most widely used integer data type. The range of values it can represent is from -231 to 231-1.
- long: is the largest integer type, 8 bytes (64 bits), with a range of values from -263 to 263-1.

Floating point numeric types

Floating-point number types allow you to represent both very large and very small numbers in addition to decimal numbers. Java has 2 specific types in this category:

- float: known as a simple precision type, it uses a total of 32 bits. With this type of data it is possible to represent numbers in the range from 1.4x10-45 to 3.4028235x1038.
- double: follows a storage scheme similar to the previous one, but using 64 bits instead of 32. This allows you to represent values in the range of 4.9x10-324 to 1.7976931348623157x10308.

Booleans and characters

Apart from the 6 data types that we have just seen, intended to work with numbers in different ranges, Java defines two other primitive types:

- boolean: has the purpose of facilitating work with "true / false" (Boolean) values, generally resulting from evaluating expressions. The two possible values of this type are true and false.
- char: used to store individual characters (letters, to understand us). Actually, it is also considered a numeric type, although its usual representation is that of the character whose code it stores. It uses 16 bits and the UTF-16 encoding of Unicode is used.

Structured data types

The primitive data types that we have just seen are characterized by being able to store a single value. Except for this small set of primitive data types, which make it easy to work with numbers, characters, and Boolean values, all other Java types are objects, also called structured types or "Classes."

Character strings

Although strings are not a simple type in Java, but an instance of the String class, the language grants a rather special treatment to this type of data, which causes that, at times, it seems to be working with a primitive type .

Although when we declare a string we are creating an object, its declaration does not differ from that of a variable of primitive type of the ones we have just seen:

String courseName = "Introduction to Java";

Vectors or arrays

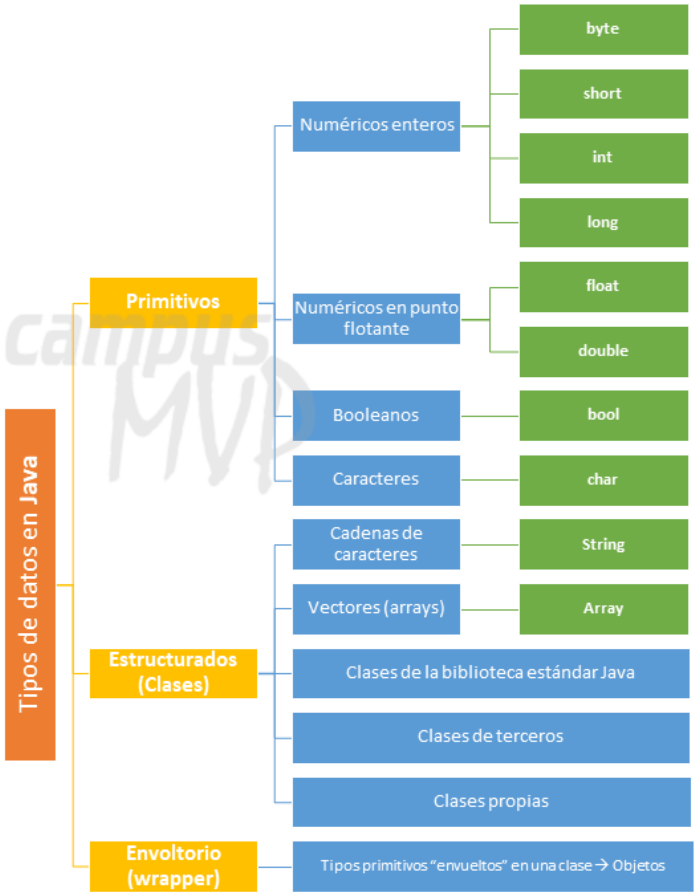
Vectors are collections of data of the same type. They are also popularly known as arrays and even as "arrays" (although the latter denomination is discouraged because it is a poor adaptation of English).
A vector is a data structure in which each element corresponds to a position identified by one or more integer numerical indices.

User defined types

In addition to the basic structured types that we have just seen (strings and vectors) in Java there are countless classes on the platform, and third parties, to perform almost any operation or task that may occur: reading and writing files, sending emails, run other applications or create more specialized text strings, among a million other things.

Wrapper types

Java has structured data types equivalent to each of the primitive types we have seen.
Thus, for example, to represent a 32-bit integer (int) that we have seen at the beginning, Java defines a class called Integer that represents and "wraps" the same data but adds certain useful methods and properties on top of it.



PYTHON

Size of an object

Knowing how many references an object has can help you find cycles or a memory leak, but it is not enough to determine which objects are consuming the most memory. That requires knowledge of how big the objects are.
sys_getsizeof.py

```
import sys


class MyClass:
    pass


objects = [
    [], (), {}, 'c', 'string', b'bytes', 1, 2.3,
    MyClass, MyClass(),
]

for obj in objects:
    print('{:>10} : {}'.format(type(obj).__name__,
                               sys.getsizeof(obj)))
```

getsizeof () reports the size of an object in bytes.

```
$ python3 sys_getsizeof.py

      list : 64
     tuple : 48
      dict : 240
       str : 50
       str : 55
     bytes : 38
       int : 28
     float : 24
      type : 1056
   MyClass : 56
```

The reported size for a custom class does not include the size of attribute values.
sys_getsizeof_object.py

```
import sys


class WithoutAttributes:
    pass


class WithAttributes:
    def __init__(self):
        self.a = 'a'
        self.b = 'b'
        return


without_attrs = WithoutAttributes()
print('WithoutAttributes:', sys.getsizeof(without_attrs))

with_attrs = WithAttributes()
print('WithAttributes:', sys.getsizeof(with_attrs))
```

This can give a false impression of how much memory is being consumed.

```
$ python3 sys_getsizeof_object.py

WithoutAttributes: 56
WithAttributes: 56
```

For a more complete estimate of the space used by a class, provide a __sizeof __ () method to calculate the value by aggregating the sizes of an object's attributes.
sys_getsizeof_custom.py

```
import sys


class WithAttributes:
    def __init__(self):
        self.a = 'a'
        self.b = 'b'
        return

    def __sizeof__(self):
        return object.__sizeof__(self) + \
            sum(sys.getsizeof(v) for v in self.__dict__.values())


my_inst = WithAttributes()
print(sys.getsizeof(my_inst))
```

This version adds the base size of the object to the sizes of all the attributes stored in the inner __dict__.

```
$ python3 sys_getsizeof_custom.py

156
```