Claudio De Sio Cesari

Java for Aliens

learn Java from scratch and become a Pro

Volume 1

To store only two values, it can be sufficient to use just one bit, but the Java Specifications say this is not certain, and that developers don't have to worry about it.

3.3.5 Primitive Character Data Type

The `char` type allows us to store characters (one at a time).

This data type is not frequently used, because in most cases strings are needed, which allow us to store sequences of characters (see section 3.5.2). Even if, in this section and the following subsections, we will delve into different aspects of this data type, actually its use is not recommended. The neophyte to programming, may skip sections from 3.3.5.2 to 3.3.5.7 and ignore the rest of the arguments related to the `char` type for now.

Below is a simple example where the character value is assigned to a `char` type:

```
    char firstCharacter = 'a';
```

Note that each literal character type must be included between two single quotes, not to be confused with double quotes that are used for string literals. We have already used strings previously, for example:

```
String s = "Java melius semper quam latinam linguam est";
```

A typical mistake when starting to program in Java is to confuse single quotes (used for characters) with double quotes (used for strings). For example, the following instructions are both incorrect:

```
    char c = "C";
    String s = 'Meum filium maxime amo, sed ille me latinam linguam studere compellit';
```

Actually, there are three ways to assign a value (literal) to a `char` type, and all three modes require the inclusion of the value between single quotes:

- use a single character on the keyboard (for example '`&`');
- use the Unicode format with hexadecimal notation (for example '`\u0061`', which is equivalent to the number `97` and which identifies the '`a`' character);
- use an escape character (for example '`\n`' which indicates the *line feed* character (wrap).

Let's see these cases in detail in the next few sections.

3.3.5.1 Printable Keyboard Characters

We can assign a `char` to any character found on our keyboard, provided that the operating system supports the required character, and that the character is *printable* (for example the delete keys, or **Enter** etc. are not printable). In any case, we must include the value to be assigned between single quotes (i.e. the literal assignable to a `char` primitive type is always included between two single quotes). Here are some examples:

```
  char aUppercase = 'A';
  char minus = '-';
  char at = '@';
```

The `char` data type is stored in 2 bytes (16 bits), with a range consisting only of positive numbers ranging from 0 to 65535. In fact, there is a ‘mapping’ that associates a certain character to each number. This mapping (or encoding) is defined by the **Unicode** standard (further described in the next section). So, we can say that the `char` type is a numeric type.
The fact that each character is represented by an integer is important, because we can use characters in arithmetic expressions. For example, we can add a `char` to an `int`. In fact, each character is decoded by Unicode starting from an integer:

```
  int i = 1;
  char a = 'A';
  char b = (char)(a+i); // c = 'B'!
```

Note that if we had not used the cast:

```
  char b = a+i;
```

we would have obtained the following compile-time error:

```
error: incompatible types: possible lossy conversion from int to char
       char b = a+i;
                 ^
1 error
```

In fact, with the automatic promotion in expressions rule (see section 3.3.1.1), the sum between `a` (of type `char`) and `i` (of type `int`) returns a value of type `int`. So, the following instruction:

```
  int ii = a+i;
```

is valid, and the value contained in `ii` will be `66`, which is the number representing the `B` character in the Unicode encoding. Thus, in arithmetic operations involving `char` types, integers representing the character type are used, not the characters themselves.

3.3.5.2 Unicode Format (Hexadecimal Notation)

We said that the `char` primitive type is stored in 16 bits, and therefore can define as many as 65536 different characters. **Unicode** encoding deals with standardizing all the characters (and also symbols, emojis, ideograms, etc.) that exist on this planet. Unicode is an extension of the encoding known as **UTF-8**, which in turn is based on the old 8-bit **Extended ASCII** standard, which in turn contains the oldest standard known as **ASCII code** (acronym for **American Standard Code for Information Interchange**).

Since version 11, Java has supported version 10.0 of Unicode, which contains many more characters than 65536 which can store a `char`. In fact, originally, it was designed as a 16-bit encoding that was considered sufficient to represent the characters of all the languages of the world. Now instead, the Unicode standard, allows us to represent potentially over a million characters, although 137,929 numbers have already been assigned to a character. But the standard is constantly evolving. Just to give you an idea, Java 10 implemented Unicode 8.0. With Unicode 9.0, 7500 characters were added and with Unicode 10.0 another 8518 more. So, with Java 11, there are 16018 new characters that can be used, if necessary, with respect to Java 10.

Now Java 13, supports Unicode Version 12.1 (the latest Unicode version), that have added other 1239 characters.

For more information, see **http://www.unicode.org**.

Unicode can be used in three ways:

1. UTF-8: 8-bit that contains the ASCII encoding and therefore all the most used characters.
2. UTF-16: 16-bit, which contains all of the characters of most alphabets in the world. Also, the `char` type uses 16 bit and therefore with it, we are able to represent practically all of the most significant characters that exist.
3. UTF-32: 32-bit that contains encodings of other characters including those that could be considered less used. Java supports the use of UTF-32 with a ruse based on an `int` (that is to concatenate two Unicode characters), but the use of this UTF-32 can be considered very rare. To use Unicode values that are outside the 16-bit range of a `char` type, we usually use classes like `String` (which will be introduced in section 3.5.2) and `Character` (the wrapper class relative to the primitive type `char`).

We can directly assign a `char` a Unicode value in hexadecimal format using 4 digits, which uniquely identifies a given character, prefixing it with the prefix `\u` (in this case always lower case). For example:

```
char phiCharacter = '\u03A6';
char nonIdentifiedUnicodeCharacter = '\uABC8';
```

In this case we're talking about **literal in Unicode format** (or **literal in hexadecimal format**). In fact, using 4 digits with the hexadecimal format, exactly 65536 characters are covered. Please note that, with this syntax, we indicate the number that identifies a certain character in the Unicode encoding. Since there are also integers from `0` (which corresponds to the number `48`) to `9` (which corresponds to the number `57`) in the coding, if we want to print the number `0` with the syntax just seen, we should write:

```
char zero = '\u0030'; //the hexadecimal number 30 is equal to the decimal 48
System.out.println(zero);
```

The value '`\u0000`' is equivalent to the `null` **character**, i.e. a character that does not print anything. Obviously, it is also possible to write:

```
char zero = '0';
```

3.3.5.3 Special Escape Characters

In a `char` type it is also possible to store **special escape characters**, that is, sequences of characters that cause particular behaviors in the printing:

- `\b` is equivalent to a *backspace*, that is a cancellation to the left (equivalent to the **Delete** key);
- `\n` is equivalent to a *line feed* (equivalent to the **Enter** key);
- `\\` equals only one `\` (just because the `\` character is used for escape characters);
- `\t` is equivalent to a horizontal tab (equivalent to the **TAB** key);
- `\'` is equivalent to a single quote (a single quote delimits the literal of a character);
- `\"` is equivalent to a double quote (a double quote delimits the literal of a string);
- `\r` represents a *carriage return* (special character that moves the cursor to the beginning of the line);
- `\f` represents a *form feed* (disused special character representing the cursor moving to the next page of the document).

Note that assigning the literal '"' to a character is perfectly legal, so the following statement:

```
System.out.println('"');
```

which is equivalent to the following code:

```
  char doubleQuotes = '"';
  System.out.println(doubleQuotes);
```

is correct and will print the double quote character:

```
"
```

If we tried not to use the escape character for a single quote, for example, with the following statement:

```
  System.out.println(''');
```

we will get the following compile-time errors, since the compiler will not be able to distinguish the character delimiters:

```
error: empty character literal
        System.out.println(''');
                          ^
error: unclosed character literal
        System.out.println(''');
                            ^
2 errors
```

Since the string literal delimiters (see section 3.5.2) are represented with double quotes, then the situation is reversed: it is possible to represent single quotes within a string:

```
  System.out.println("'IQ'");
```

that will print:

```
'IQ'
```

On the other hand, we must use the \" escape character to use double quotes within a string. So, the following instruction:

```
  System.out.println(""IQ"");
```

will cause the following compilation errors:

```
error: ')' expected
        System.out.println(""IQ"");
                            ^
error: ';' expected
        System.out.println(""IQ"");
                              ^
2 errors
```

Instead, the following instruction is correct:

```
  System.out.println("\"IQ\"");
```

and will print:

```
"IQ"
```

In the next four sections, we will examine a series of extreme cases that may arise during the use of `char` primitive data types. As the alien icon tells you, if you are on your first experience with Java, you can safely skip these sections and come back later when you feel ready. This is not immediately necessary information for novices, and only a small percentage of professional Java programmers already know about it.

3.3.5.4 Write Java Code with the Unicode Format

The Unicode literal format can also be used to replace any line of our code. In fact, the compiler transforms the Unicode format into a character, and then evaluates the syntax. For example, we could rewrite the following statement:

```
int i = 8;
```

in the following way:

```
\u0069\u006E\u0074 \u0069 \u003D \u0038\u003B
```

In fact, if we add the following to the statement in the previous line:

```
System.out.println("i = " + i);
```

it will print:

```
i = 8
```

Undoubtedly, this is not a useful way to write our code. But we have to know this feature in order to understand some mistakes that rarely happen.

3.3.5.5 Unicode Format for Escape Characters

The fact that the Unicode hexadecimal format is transformed by the compiler before it evaluates the code, has some consequences, especially when dealing with escape characters. For example, let's consider the *line feed* character (which can be represented with the escape character `\n`, which corresponds in the Unicode encoding to the decimal number 10 (which corresponds to the hexadecimal number A). If we try to define it using the Unicode format:

```
char lineFeed = '\u000A';
```

we will get the following compile-time error:

```
error: illegal line end in character literal
        char lineFeed = '\u000A';
                        ^
1 error
```

In fact, the compiler transforms the previous code into the following before evaluating it:

```
char lineFeed = '
';
```

that is, the Unicode format has been transformed into the *newline* character, and the previous syntax is not a valid syntax for the Java compiler.
In the same way, the single quote character (‘) that corresponds to the hexadecimal number 27 (equivalent to the decimal number 39) and that we can represent with the escape character \’, cannot be represented with the Unicode format:

```
char apex = '\u0027';
```

Also in this case, the JVM will transform the previous code in this way:

```
char apex = ''';
```

which will give rise to the following compile-time errors:

```
error: empty character literal
        char apex = '\u0027';
                    ^
error: unclosed character literal
        char apex = '\u0027';
                           ^
2 errors
```

The first error is due to the fact that the first pair of quotes does not contain a character, while the second error indicates that specifying the third apex is an unclosed character literal.
Also, with regard to the *carriage return* character, represented by the hexadecimal number D (corresponding to the decimal number 13), and already representable with the escape character \r, there are problems. In fact, if we write:

```
char carriageReturn = '\u000d';
```

we will get the following compile-time error:

```
error: illegal line end in character literal
        char carriageReturn = '\u000d';
                              ^
1 error
```

In fact, the JVM has transformed the number in Unicode format into a *carriage return* by returning the cursor to the beginning of the line, and what was supposed to be the first apex became the second.
As for the character \, represented by the hexadecimal number 5C (corresponding to the decimal number 92), and, represented by the escape character \\, if we write:

```
  char backSlash = '\u005C';
```

we will get the following compile-time error:

```
error: unclosed character literal
        char backSlash = '\u005C';
                         ^
1 error
```

This is because the previous code will have been transformed into the following:

```
  char backSlash = '\';
```

and therefore the \’ pair of characters is considered as an escape character corresponding to an apex ‘, and therefore the literal closure is missing another single quote.
On the other hand, if we consider the character “, represented by the hexadecimal number 22 (corresponding to the decimal number 34), and, represented by the escape character \”, if we write:

```
  char quotationMark = '\u0022';
```

there will be no problem. But if we use this character within a string:

```
  String quotationMarkString = "\u0022";
```

we will get the following compile-time error:

```
error: unclosed string literal
        System.out.println("\u0022");
                                  ^
1 error
```

since the previous code will have been transformed into the following:

```
  String quotationMarkString = """;
```

3.3.5.6 Unicode Format and Comments

An even stranger situation is found when using single-line comments for Unicode formats such as *carriage return* or *line feed*. For example, despite being commented, both of the following statements would give rise to compile-time errors!

```
  // char lineFeed = '\u000A';
  // char carriageReturn = '\u000d';
```

This is because the hexadecimal formats are always transformed by the compiler with the *line feed* and *carriage return* characters, which are not compatible with the single line comments, because they print characters outside the comment!
To solve the situation, use the multi-line comment notation, for example:

```
  /* char lineFeed = '\u000A';
     char carriageReturn = '\u000d'; */
```

Another mistake that can cause a programmer to lose a lot of time, is when the sequence `\u` is used in a comment. For example, with the following comment, we will get a compile-time error:

```
  /*
   * The file will be generated inside the C:\users\claudio folder
   */
```

If the compiler does not find a sequence of 4 hexadecimal characters valid after `\u`, it will print the following error:

```
error: illegal unicode escape
 * The file will be generated inside the C:\users\claudio folder
                                            ^
1 error
```

3.3.5.7 Casting Between short and char

In section 3.3.5.2, we saw a cast between an `int` type and a `char` type. The rules of casting between `int` and `char` are obvious, because `int` is a type of data that stores data in 32 bits, while `char` stores them in 16. So, the following snippet is valid:

```
  int i = '\n';
  System.out.println(i);
```

and will print the value `10` representing the *line feed* character.
But what is the cast relationship between `char` and `short` since both store 16-bit numbers? A

clear difference between the two types is that char is a type that can be defined without a sign. In fact, it is represented only by positive numbers ranging from 0 to 65535. Instead the short type has a range that varies from -32768 to +32767. So, not all short values can be stored in a char, and not even all the char values can be stored in a short. For this reason, if we want to assign a short to a char or vice versa, a cast is required in any case. This is also necessary when it is certain that a value of one type falls within the representation range of the other type. For example, we have already seen that it is possible to assign a value to a certain integer type that is included in its representation interval. For example, the following are valid statements:

```
char c = 'a'; //the character is equal to the decimal number 65
short s = 65;
```

But although s can hold the value 65, to assign c to s there is still a need for a cast not to get a compile-time error:

```
s = (short)c;
```

Instead, let's consider an example, where we force, using cast non-compatible values in the two types we are examining:

```
char cc = '\uffff'; // the hexadecimal FFFF corresponds to the decimal 65535
short ss = (short)cc;
```

In this case, for the same rule used with the byte type in section 3.3.1.2, the variable ss will have value -1. Instead, in this case:

```
short sss = -32768;
char ccc = (char)sss;
System.out.println(ccc);
```

the ccc character will have an undefined value, and a question mark will be printed.

3.4 Non-Primitive Data Types: Reference

We have already seen how to instantiate objects from a certain class. We must first declare an object of this class with a syntax of this type:

```
ClassName objectName;
```

and then instantiate it using the keyword new:

```
objectName = new ClassName();
```

To declare an object is therefore very similar to declaring a type of primitive data. The "name" we give to an object is called a **reference**. We are not talking about a traditional variable, but