# Basic elements of a Java program

## Basic data types

In previous sections we have talked about variables. We have learnt that we can use them to store values, and these values can be of different types. In this section we are going to learn about the basic data types provided by Java, and how we can use them to store values in our programs.

If you are interested, you can read the Java Language Specification section about primitive types.

### 1. Numeric types

There are two main numeric types in Java:

- Integer values, which can be represented by `byte`, `short`, `int` or `long` data types
- Real value, which can be represented by `float` or `double` data types.

#### 1.1. Integer data types

As we have said before, we can choose among 4 different data types to represent integer values. The choice can be determined by the range of values that we need to deal with. In this table you can see the range of values allowed by each data type:

| Data type | Memory (bytes) | Range allowed |
|---|---|---|
| `byte` | 1 | -128 to 127 |
| `short` | 2 | -65536 to 65535 |
| `int` | 4 | -2.147.483.648 to 2.147.483.647 |
| `long` | 8 | up to 18-19 digit numbers |

For instance, if we want to manage the age of a person, we could use an `int` variable, but we would waste some memory, since this age is usually lower than 100, and we would just need a single byte to store it. We could use a `byte` variable instead:

```
byte age = 34;
```

However, if we want to store the price of an object, we should use a `short` or even an `int` variable:

```
short price = 4200;
int higherPrice = 2223424;
```

### 1.2. Real data types

If we want to deal with real numbers, Java provides two different data types, each one with its own memory space and range:

- `float` data type needs 4 bytes of memory, and lets us manage numbers with up to 6-7 significant numbers. For instance, if we want to store PI value in a float variable with just 4 or 5 fraction digits (i.e. *3.14159*), we can use a `float` variable.

```
float pi = 3.14159;
```

- `double` data type needs 8 bytes of memory, and lets us manage numbers with up to 15 significant numbers. This way we can store more fraction digits, if we want to:

```
double pi = 3.14159265359;
```

Regarding `float` variables, if we want to assign them a direct value, we must specify an `f` symbol at the end of this value. So the example given before should be written like this (otherwise we'll get a compilation error):

```
float pi = 3.14159f;
```

### 1.3. The overflow problem

When we are working with numbers, we may need to do some operations that exceed the maximum range allowed by a data type. For instance, if we are working with two `byte` values and we add them, we may exceed the maximum range allowed by `byte` data type, which is 127. This situation is called **overflow**.

So, we must take care of the data types that we choose for each situation, taking into account the different operations that we expect to do with these variables.

**The integer operators do not indicate overflow or underflow in any way.**

## 2. Text types

In order to deal with texts, Java provides two data types:

- `char` data type if we want to use single characters or symbols
- `String` data type if we want to manage complex texts (with more than one character or symbol).

Regarding `char` data type, it is 2 bytes length, so that we can represent any possible character or symbol. We just declare the corresponding variable, and assign the character represented between single quotes:

```
char symbol = 'a';
```

If we want to work with longer texts, then we use `String` variables, specifying the text between double quotes:

```
String text = "Hello world";
```

### 2.1. Escape sequences

There are some special characters that can't be represented easily with the keyboard in a source file. For instance, the *new line* character, or even the quotes inside a quoted text. For this purpose, we can use **escape sequences**, this is, special symbols that represent these unwritable elements. This is a list of the most popular escape characters or sequences:

| Sequence | Meaning |
|---|---|
| `\n` | New line |
| `\t` | Tabulation |
| `\"` | Double quotes |
| `\'` | Single quotes |
| `\\` | Backslash `\` |

These escape sequences can be placed inside a char or string value:

```
char newLine = '\n';
String message = "Hello world.\n\"Quoted text\"";
```

### 2.2. Character operations

We can do some basic operations with characters. You need to take into account that Java internally treats characters as numeric values, assigning each character a numeric code. For instance, alphabet characters

are represented by consecutive numeric values, from `a` to `z`. This way, if we add 3 to `a` value, we will get `d` value:

```
char symbol = 'a';
symbol += 3;
```

We can also use `+` operator in texts (*strings*), but in this case we are not doing any addition, we are just concatenating texts or expressions. This expressions produces the text "Hello3":

```
String text = "Hello" + 3;
```

Keep in mind that you can't mix arithmetic and text operations in a single line directly. The following expression produces a result of "Hello32":

```
String text = "Hello" + 3 + 2;
```

If you want to calculate the addition and then concatenate the result, then you must prioritize the addition using parentheses. This expression produces a result of "Hello5":

```
String text = "Hello" + (3 + 2);
```

## 3. Conversion between data types

Sometimes we need to convert a value of some type into another different type. The way we do this step depends on the types involved.

### 3.1. Some basic conversions. Typecasting

The conversions between numeric values are quite straightforward. We just need to do a **typecast**, this is, specify between parentheses the data type to which we want to convert the expression. In this example, we are converting `pi` real value to an integer (so we get 3 as final result):

```
float pi = 3.1416f;
int piInteger = (int)pi;
```

The opposite step can also be done. In this case, we are converting an integer value into a double one (the final value will be 5.0):

```
int number = 5;
double realNumber = (double)number;
```

However, this step is NOT necessary if the source type is smaller than the destination type. For instance, a `byte` doesn't need to be converted into `int`:

```
byte value = 3;
int number = value;
```

Typecast can be useful, for instance, to convert integer divisions into real ones. This example divides two integer values, but, as we are converting one of them into `float`, then the final result will be a real number, with the corresponding fraction digits, and can be stored in a `float` variable:

```
float result = (float) 3 / 2;
```

If we mix two different types in an arithmetic operation, then Java converts the result to the highest of them. This multiplication gets a `float` number because one of the operands is `float`:

```
float a = 3.5f;
int b = 4;
float result = a * b;
```

If an **integer** operator has at least one operand of type `long`, then the operation is carried out using 64-bit precision, and the result of the numerical operator is of type `long`. If the other operand is not long, it is first widened to type long by numeric promotion. **Otherwise, the operation is carried out using 32-bit precision, and the result of the numerical operator is of type `int`**. If either operand is not an int, it is first widened to type int by numeric promotion.

```
byte a = 3, b = 2;
byte result = (byte)(a + b); // + result is an int, so casting to byte is nee
```

### 3.2. Converting from / to string

In some situations, we may read numeric values from textual sources, such as text file, or user input. In this case, we need to **convert the text into the corresponding numeric value**. To do this, Java provides some useful instructions. Here you can see the most useful ones:

- `Integer.parseInt` converts a text value into `int`:

```
int value = Integer.parseInt("23");
```

- `Float.parseFloat`, `Double.parseDouble`, `Byte.parseByte`, `Short.parseShort` and `Long.parseLong` do the same with their corresponding data types:

```
float value = Float.parseFloat("3.1416");
```

If we want to convert a **numeric value into a string**, we can choose one of these solutions:

- Concatenate the numeric value with an empty string `""`:

```
int number = 23;
String text = "" + number;
```

- Use `String.valueOf` instruction to convert the specified value to string:

```
int number = 23;
String text = String.valueOf(number);
```

**Exercise 1:**

Create a program called **Ages.java** that:

- Defines two `byte` variables to store your age and the age of a friend
- Defines another `byte` variable to store the addition of both ages (you may need to typecast the result)
- Defines a `float` variable to store the average of these ages, including fraction digits
- Prints the message *"The age average is "* followed by the average calculated in previous step