# 7-primitives-and-literals

## Contents



- Primitives
  - Number
  - Boolean
  - Character
- Literals
  - Types
  - Range
  - Separation
  - Sequences
  - Casting
  - Promotion

## Primitives

Java defines eight primitive types of data: `byte`, `short`, `int`, `long`, `char`, `float`, `double`, and `boolean`. The primitive types are also commonly referred to as simple types,

### Number

| Name | Width | Range |
|---|---|---|
| long | 64 | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| int | 32 | -2,147,483,648 to 2,147,483,647 |
| short | 16 | -32,768 to 32,767 |
| byte | 8 | -128 to 127 |
| double | 64 | 4.9e-324 to 1.8e+308 |
| float | 32 | 1.4e-045 to 3.4e+038 |

### Boolean

It can have only one of two possible values, true or false. This is the type returned by all relational operators, as in the

### Character

In Java, the data type used to store characters is char. However, C/C++ programmers beware: char in Java is not the same as char in C or C++. In C/C++, char is 8 bits wide. This is not the case in Java. Instead, Java uses Unicode to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages. It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more. At the time of Java's creation, Unicode required 16 bits. Thus, in Java char is a 16-bit type. The range of a char is 0 to 65,536 - holds 2 bytes, and has the same range as `unsigned short` (in other languages)

## Literals

Are the special types of values that can be represented in code, literally. They are usually stored in a special way as a part of the program's data segment or the stack. But what is important to note is that they are immutable, and identical. Meaning that a number literal `1` is identical to another number literal `1` in the code, same applies for String literals too, a literal of "hello" is the same as "hello" somewhere else in another part of the code, which is unlike other languages where string literals do not exist or are not treated at compile time

### Types

### Integer literals

- base 10 - e.g. `1,2,3,4,100,999`
- base 16 - e.g. `0xFE,0xDE,0xA0`
- base 8 - e.g. `073,012`
- base 2 - e.g. `0b1010,0b1111`

### Boolean literals

- boolean - e.g. `false` or `true`

Floating literals

- double - e.g. `9.54`
- float - e.g. `9.54f`

### Character literals

- char - `\u0061`

### String literals

- string - "value"

```
String str1 = "hello"; // created
String str2 = "hello"; // reused

System.out.println(str1 == str2); // true
```

Note that unlike other literals, the string literal below has some very special properties, similarly to other languages where string literals are stored on the stack or the data segment of a program. Java has a similar way of storing string literals in a pool. This pool of strings is allocated on the heap and when a string literal is created the JVM checks if that string already exists in the pool, reusing the existing object - string, otherwise a new one is created. String literals are also immutable.

The process of declaring a string using double quotes (e.g., String str = "hello";), is called automatic string interning, it is stored in the string pool. However, if you create a string using the new keyword (e.g., String str = new String("hello");), the string is not automatically interned. This creates a new string object on the heap.

Compile time constant String literals which are concatenated are optimized during the byte code compilation step, and result into a single string instead of multiple.

```
String s1 = "hello" + " " + "world";
String s2 = "hello world";
System.out.println(s1 == s2); // true, both are optimized to the same string
   literal
```

String literals are treated as constants, meaning that they can be used where constants are expected, such as in switch statements or final variables.

### Range

By default all literals which we specify are actually of type `int`, and all decimal or floating point literals (e.g. 9.54) ones default to `double` type. To signify a literal of other type one must suffix it with an additional character such as `l` for long `f` for float, otherwise a compilation error will occur

```
    long longer = 0x7fffffffffL; // in range, suffixed with l to signify it
       is long
    int integer = 0x7fffffffff; // compilation error
    byte character = 0x43434; // compilation error
```

### Separation

With java 7 it is also possible to add `underscores` for spacing integer literals, for better readability, for example having something like that `int readable = 999_999_999_999` or `int readable = 999___999___999` is quite helpful, one can also do this with other radix types such as hex `int readable = 0xFF_FF_FF_FF`. This is also applicable for double, float and so on.

### Sequences

| Sequence | Description |
|---|---|
| | Octal character (ddd) |
| | Hexadecimal Unicode character (xxxx) |
| ' | Single quote |
| ” | Double quote |
| \ | Backslash |
| Carriage return | |
| | New line (also known as line feed) |
| Form feed | |
| Tab | |

| Sequence | Description |
|---|---|
| Backspace | |

### Casting

In java by default the language runtime will perform automatic casting when the two types are compatible or the destination type is of a higher ordinance than the source. An `int` is always big enough to hold a `byte`, but the inverse is not true. Also `char` and `boolean` unlike in other languages are not compatible, they can not be converted from one to the other. If we want to convert from a higher ordinance type to a lower ordinance type we have to use a cast operator to explicitly tell java that we would like to perform what is called a `narrowing` cast or conversion

```
byte wrong = 1024; // that will produce a compile time error
byte right = (byte) 1024; // this will work, however overflow
int flat = (int) 1.24f; // the result will be 1 assigned to var
```

### Promotion

When evaluating expressions, java would automatically promote types, consider the following

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c; // a * b will be automatically promoted to int during
   compile time
```

The `a*b` will easily exceed the size of byte, which is `0-255`, however java would make sure that the result of the expression `a*b` is not of type byte, which are the types of the arguments included in the expression, but rather it will automatically promote them to higher value, in this case it might be short or int, such that the result of the expression fits without overflowing

```
byte b = 50;
b = b * 2; // this will produce an error precisely because the automatic
   promotion.
```

However consider the case above, since java is doing the automatic promotion at compile time, the result of `b*2` will be interpreted as `int` or `short` meaning we would try to assign the intermediate result of `b*2` to `b` which is `byte` that is not allowed, and since we are doing an assignment from a higher ordinance type to a lower one, compile time error occurs

`Automatic promotion rules: First, all byte, short, and char values are promoted to int, as just described. Then, if one operand is a long, the whole expression is promoted to long . If one operand is a float, the entire expression is promoted to float. If any of the operands are double, the result is double.`

```
byte b = 42;
char c = 'a';
short s = 1024;
int i = 50000;
float f = 5.67f;
double d = .1234;
double result = (f * b) + (i / c) - (d * s);
System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
System.out.println("result = " + result);
```

- `f \* b`, b is promoted to a `float` and the result of the sub expression is `float`.
- `i/c`, c is promoted to `int`, and the result is of type `int`.
- `d \* s`, the value of s is promoted to `double`, and the type of the sub expression is `double`.
- finally, these three intermediate values, `float, int, and double` are considered. The outcome of `float` plus an `int` is a `float`. Then the resultant `float` minus the last double is promoted to `double`,