Chapter 9: Literals

A Java literal is a syntactic element (i.e. something you find in the *source code* of a Java program) that represents a value. Examples are `1`, `0.333F`, `false`, `&#39;X&#39;` and `"Hello world\n"`.

Section 9.1: Using underscore to improve readability

Since Java 7 it has been possible to use one or more underscores (_) for separating groups of digits in a primitive number literal to improve their readability.

For instance, these two declarations are equivalent:

Version ≥ Java SE 7

```
int i1 = 123456;
int i2 = 123_456;
System.out.println(i1 == i2); // true
```

This can be applied to all primitive number literals as shown below:

Version ≥ Java SE 7

```
byte color = 1_2_3;
short yearsAnnoDomini= 2_016;
int socialSecurtyNumber = 999_99_9999;
long creditCardNumber = 1234_5678_9012_3456L;
float piFourDecimals =  3.14_15F;
double piTenDecimals =  3.14_15_92_65_35;
```

This also works using prefixes for binary, octal and hexadecimal bases:

Version ≥ Java SE 7

```
short binary= 0b0_1_0_1;
int octal = 07_7_7_7_7_7_7_0;
long hexBytes = 0xFF_EC_DE_5E;
```

There are a few rules about underscores which **forbid** their placement in the following places:

- At the beginning or end of a number (e.g. `_123` or `123_` are *not* valid)
- Adjacent to a decimal point in a floating point literal (e.g. `1._23` or `1_.23` are *not* valid)
- Prior to an F or L suffix (e.g. `1.23_F` or `9999999_L` are *not* valid)
- In positions where a string of digits is expected (e.g. `0_xFFFF` is *not* valid)

Section 9.2: Hexadecimal, Octal and Binary literals

A `hexadecimal` number is a value in base-16. There are 16 digits, `0-9` and the letters `A-F` (case does not matter). `A-F` represent `10-15`.

An `octal` number is a value in base-8, and uses the digits `0-7`.

A `binary` number is a value in base-2, and uses the digits `0` and `1`.

All of these numbers result in the same value, `110`:

```
int dec = 110;            // no prefix   --> decimal literal
int bin = 0b1101110;      // '0b' prefix --> binary literal
int oct = 0156;           // '0' prefix  --> octal literal
```

```
int hex = 0x6E;           // '0x' prefix --> hexadecimal literal
```

Note that binary literal syntax was introduced in Java 7.

The octal literal can easily be a trap for semantic errors. If you define a leading `'0'` to your decimal literals you will get the wrong value:

```
int a = 0100;        // Instead of 100, a == 64
```

Section 9.3: Boolean literals

Boolean literals are the simplest of the literals in the Java programming language. The two possible **`boolean`** values are represented by the literals **`true`** and **`false`**. These are case-sensitive. For example:

```
boolean flag = true;    // using the 'true' literal
flag = false;           // using the 'false' literal
```

Section 9.4: String literals

String literals provide the most convenient way to represent string values in Java source code. A String literal consists of:

- An opening double-quote (`"`) character.
- Zero or more other characters that are neither a double-quote or a line-break character. (A backslash (`\`) character alters the meaning of subsequent characters; see Escape sequences in literals.)
- A closing double-quote character.

For example:

```
"Hello world"   // A literal denoting an 11 character String
""              // A literal denoting an empty (zero length) String
"\""            // A literal denoting a String consisting of one
                //     double quote character
"1\t2\t3\n"     // Another literal with escape sequences
```

Note that a single string literal may not span multiple source code lines. It is a compilation error for a line-break (or the end of the source file) to occur before a literal's closing double-quote. For example:

```
"Jello world    // Compilation error (at the end of the line!)
```

Long strings

If you need a string that is too long to fit on a line, the conventional way to express it is to split it into multiple literals and use the concatenation operator (+) to join the pieces. For example

```
String typingPractice = "The quick brown fox " +
                        "jumped over " +
                        "the lazy dog"
```

An expression like the above consisting of string literals and + satisfies the requirements to be a Constant Expression. That means that the expression will be evaluated by the compiler and represented at runtime by a single `String` object.

Interning of string literals

When class file containing string literals is loaded by the JVM, the corresponding `String` objects are *interned* by the runtime system. This means that a string literal used in multiple classes occupies no more space than if it was used in one class.

For more information on interning and the string pool, refer to the String pool and heap storage example in the Strings topic.

Section 9.5: The Null literal

The Null literal (written as `null`) represents the one and only value of the null type. Here are some examples

```
MyClass object = null;
MyClass[] objects = new MyClass[]{new MyClass(), null, new MyClass()};

myMethod(null);

if (objects != null) {
     // Do something
}
```

The null type is rather unusual. It has no name, so you cannot express it in Java source code. (And it has no runtime representation either.)

The sole purpose of the null type is to be the type of `null`. It is assignment compatible with all reference types, and can be type cast to any reference type. (In the latter case, the cast does not entail a runtime type check.)

Finally, `null` has the property that `null instanceof <SomeReferenceType>` will evaluate to `false`, no matter what the type is.

Section 9.6: Escape sequences in literals

String and character literals provide an escape mechanism that allows express character codes that would otherwise not be allowed in the literal. An escape sequence consists of a backslash character (`\`) followed by one ore more other characters. The same sequences are valid in both character an string literals.

The complete set of escape sequences is as follows:

Escape sequence	Meaning
`\\`	Denotes an backslash (`\`) character
`\'`	Denotes a single-quote (`'`) character
`\"`	Denotes a double-quote (`"`) character
`\n`	Denotes a line feed (`LF`) character
`\r`	Denotes a carriage return (`CR`) character
`\t`	Denotes a horizontal tab (`HT`) character
`\f`	Denotes a form feed (`FF`) character
`\b`	Denotes a backspace (`BS`) character
`\<octal>`	Denotes a character code in the range 0 to 255.

The **`<octal>`** in the above consists of one, two or three octal digits ('0' through '7') which represent a number between 0 and 255 (decimal).

Note that a backslash followed by any other character is an invalid escape sequence. Invalid escape sequences are treated as compilation errors by the JLS.

Reference:

- JLS 3.10.6. Escape Sequences for Character and String Literals

Unicode escapes

In addition to the string and character escape sequences described above, Java has a more general Unicode escaping mechanism, as defined in JLS 3.3. Unicode Escapes. A Unicode escape has the following syntax:

```
'\' 'u' <hex-digit> <hex-digit> <hex-digit> <hex-digit>
```

where **<hex-digit>** is one of '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f', 'A', 'B', 'C', 'D', 'E', 'F'.

A Unicode escape is mapped by the Java compiler to a character (strictly speaking a 16-bit Unicode *code unit*), and can be used anywhere in the source code where the mapped character is valid. It is commonly used in character and string literals when you need to represent a non-ASCII character in a literal.

Escaping in regexes

TBD

Section 9.7: Character literals

Character literals provide the most convenient way to express **char** values in Java source code. A character literal consists of:

- An opening single-quote (') character.
- A representation of a character. This representation cannot be a single-quote or a line-break character, but it can be an escape sequence introduced by a backslash (\) character; see Escape sequences in literals.
- A closing single-quote (') character.

For example:

```
char a = 'a';
char doubleQuote = '"';
char singleQuote = '\'';
```

A line-break in a character literal is a compilation error:

```
char newline = '
// Compilation error in previous line
char newLine = '\n';  // Correct
```

Section 9.8: Decimal Integer literals

Integer literals provide values that can be used where you need a **byte**, **short**, **int**, **long** or **char** instance. (This example focuses on the simple decimal forms. Other examples explain how to literals in octal, hexadecimal and binary, and the use of underscores to improve readability.)

Ordinary integer literals

The simplest and most common form of integer literal is a decimal integer literal. For example:

```
0       // The decimal number zero       (type 'int')
```

```
1       // The decimal number one        (type 'int')
42      // The decimal number forty two  (type 'int')
```

You need to be careful with leading zeros. A leading zero causes an integer literal to be interpreted as *octal* not decimal.

```
077    // This literal actually means 7 x 8 + 7 ... or 63 decimal!
```

Integer literals are unsigned. If you see something like `-10` or `+10`, these are actually *expressions* using the unary `-` and unary `+` operators.

The range of integer literals of this form have an intrinsic type of **`int`**, and must fall in the range zero to 231 or 2,147,483,648.

Note that 231 is 1 greater than `Integer.MAX_VALUE`. Literals from 0 through to `2147483647` can be used anywhere, but it is a compilation error to use `2147483648` without a preceding unary `-` operator. (In other words, it is reserved for expressing the value of `Integer.MIN_VALUE`.)

```
int max = 2147483647;     // OK
int min = -2147483648;    // OK
int tooBig = 2147483648;  // ERROR
```

Long integer literals

Literals of type **`long`** are expressed by adding an `L` suffix. For example:

```
0L          // The decimal number zero       (type 'long')
1L          // The decimal number one        (type 'long')
2147483648L // The value of Integer.MAX_VALUE + 1

long big = 2147483648;    // ERROR
long big2 = 2147483648L;  // OK
```

Note that the distinction between **`int`** and **`long`** literals is significant in other places. For example

```
int i = 2147483647;
long l = i + 1;           // Produces a negative value because the operation is
                          // performed using 32 bit arithmetic, and the
                          // addition overflows
long l2 = i + 1L;         // Produces the (intuitively) correct value.
```

Reference: JLS 3.10.1 - Integer Literals

Section 9.9: Floating-point literals

Floating point literals provide values that can be used where you need a **`float`** or **`double`** instance. There are three kinds of floating point literal.

- Simple decimal forms
- Scaled decimal forms
- Hexadecimal forms

(The JLS syntax rules combine the two decimal forms into a single form. We treat them separately for ease of explanation.)

There are distinct literal types for **`float`** and **`double`** literals, expressed using suffixes. The various forms use letters to express different things. These letters are case insensitive.

Simple decimal forms

The simplest form of floating point literal consists of one or more decimal digits and a decimal point (`.`) and an optional suffix (`f`, `F`, `d` or `D`). The optional suffix allows you to specify that the literal is a **`float`** (`f` or `F`) or **`double`** (`d` or `D`) value. The default (when no suffix is specified) is **`double`**.

For example

```
0.0     // this denotes zero
.0      // this also denotes zero
0.      // this also denotes zero
3.14159 // this denotes Pi, accurate to (approximately!) 5 decimal places.
1.0F    // a `float` literal
1.0D    // a `double` literal.  (`double` is the default if no suffix is given)
```

In fact, decimal digits followed by a suffix is also a floating point literal.

```
1F      // means the same thing as 1.0F
```

The meaning of a decimal literal is the IEEE floating point number that is *closest* to the infinite precision mathematical Real number denoted by the decimal floating point form. This conceptual value is converted to IEEE binary floating point representation using *round to nearest*. (The precise semantics of decimal conversion are specified in the javadocs for `Double.valueOf(String)` and `Float.valueOf(String)`, bearing in mind that there are differences in the number syntaxes.)

Scaled decimal forms

Scaled decimal forms consist of simple decimal with an exponent part introduced by an `E` or `e`, and followed by a signed integer. The exponent part is a short hand for multiplying the decimal form by a power of ten, as shown in the examples below. There is also an optional suffix to distinguish **`float`** and **`double`** literals. Here are some examples:

```
1.0E1    // this means 1.0 x 10^1 ... or 10.0 (double)
1E-1D    // this means 1.0 x 10^(-1) ... or 0.1 (double)
1.0e10f  // this means 1.0 x 10^(10) ... or 10000000000.0 (float)
```

The size of a literal is limited by the representation (**`float`** or **`double`**). It is a compilation error if the scale factor results in a value that is too large or too small.

Hexadecimal forms

Starting with Java 6, it is possible to express floating point literals in hexadecimal. The hexadecimal form have an analogous syntax to the simple and scaled decimal forms with the following differences:

1. Every hexadecimal floating point literal starts with a zero (`0`) and then an `x` or `X`.
2. The digits of the number (but *not* the exponent part!) also include the hexadecimal digits `a` through `f` and their uppercase equivalents.
3. The exponent is *mandatory*, and is introduced by the letter `p` (or `P`) instead of an `e` or `E`. The exponent represents a scaling factor that is a power of 2 instead of a power of 10.

Here are some examples:

```
0x0.0p0f    // this is zero expressed in hexadecimal form (`float`)
0xff.0p19   // this is 255.0 x 2^19 (`double`)
```

Advice: since hexadecimal floating-point forms are unfamiliar to most Java programmers, it is advisable to use them sparingly.

Underscores

Starting with Java 7, underscores are permitted within the digit strings in all three forms of floating point literal. This applies to the "exponent" parts as well. See Using underscores to improve readability.

Special cases

It is a compilation error if a floating point literal denotes a number that is too large or too small to represent in the selected representation; i.e. if the number would overflow to +INF or -INF, or underflow to 0.0. However, it is legal for a literal to represent a non-zero denormalized number.

The floating point literal syntax does not provide literal representations for IEEE 754 special values such as the INF and NaN values. If you need to express them in source code, the recommended way is to use the constants defined by the `java.lang.Float` and `java.lang.Double`; e.g. `Float.NaN`, `Float.NEGATIVE_INFINITY` and `Float.POSITIVE_INFINITY`.