The Painless Painless Primer

Nils Bandener · 2020-05-05

In the most recent versions of Elasticsearch, the Painless language is your only option for general-purpose scripting. Painless is syntax-wise similar to Groovy and Kotlin, its runtime library is built upon the Java runtime library. Yet, there are some nuanced differences which might be confusing.

As documentation, Elastic offers the Painless Guide, which is however quite high-level. Additionally, the Painless Language Specification is offered, which contains all the important details but is kind of hard to digest.

We're trying to fill this gap by this article this gap by highlighting important functionality and not-so-obvious details.

Operators

Most operators in the Painless programming language are identical to the operators available in Java or similar languages. In some cases, there are differences which can be similarly found in Groovy, Kotlin or other scripting languages:

The operators *==* and *!=* test for equality and inequality of values and objects. This is the same as calling *equals()* in Java.

The operators *===* and *!==* then test for object identity. This is the same as *==* and *!=* in Java.

The reference operators *a.b* and *a?.b* allow simplified access of Java-style attributes defined with getter and setter methods. Thus, you can access an attribute available via a *getB()* method with the syntax *a.b*. The same holds for setters.

Null safe references are available with the syntax *a?.b*. If a is null, the expression results in null. If the normal reference operator would have been used, an exception would have occurred.

The so-called elvis operator *a?:b* results in *b* if *a* is *null*. If *a* is not *null*, it will result in *a*. Thus, the expression *a?:b* is effectively the same as *a != null ? a : b*.

See the language spec for all operators.

Variables

Generally, variables in Painless are defined like in Java. Specifying generic type parameters is not supported, though. Additionally, there is the keyword *def*, which allows you to define a typeless variable.

```
int primitiveTypeVariable = 1;
List referenceTypeVariable = new ArrayList();
def dynamicTypeVariable = new ArrayList();
```

The *def* keyword in Painless indeed defines a typeless variable. Painless does not do any type inference during compile time like it would be the case with *var* declarations in Java or *def* in Groovy with type inference enabled.

This has the consequence that trying to use non-existing members on variables defined with *def* results in a runtime error. On the other hand, using the actual type for the variable declaration results in a compilation error for non-existing members. Furthermore, accessing members of *def* variables is slightly slower than accessing members of variables with explicitly defined types.

```
def l = new ArrayList(); return l.x(); // Runtime error

List l = new ArrayList(); return l.x(); // Compile-time error
```

Loops

The loops available in Painless are exactly the same as in Java: You have *for* (in two varieties), *while* and *do/while*.

```
for (def x : ctx._source.my_val) {
    result += x;
}
```

Types

The types available in the Painless programming language are mostly the same like in Java. Painless has all of Java's primitive types: *int*, *float*, *boolean*, *char*, etc. Each primitive type is accompanied by a corresponding wrapper type like *Integer*, *Float*, *Boolean*. Similar to Java, Painless does automatic conversion (boxing and unboxing) between primitive and wrapper type as needed. The only exception are assignments in variable declarations.

```
int x = 1; // OK
int x = Integer.valueOf(1); // Error
```

See the official Painless spec for an overview of all types.

Additionally, Painless supports a limited selection of the types from the Java runtime API. Note that if a type is supported by Painless, it might not support the whole range of methods and members available in Java. This might be due to security considerations or language restrictions. Painless only supports method overloading based on the number of parameters, but not on the type of parameters. Thus, whenever the Java API offers several methods that differ only by parameter types, Painless will only support one of them. You can check the official docs for the supported methods.

We'll cover some of the important types in the next sections.

String

You can declare string literals both with double quotes or single quotes:

```
String a = "abc";
String a = 'abc';
```

The only difference between single-quoted strings and double-quoted strings is that single-quoted strings don't need to use escapes for double quotes and double-quoted strings don't need to use escapes for single quotes. Painless does not support variable interpolation inside string literals like other languages.

String concatenation is done with the *+* operator. String equality can be tested using the *==* operator.

Important methods on *String* are length() and substring(begin, end). See the language spec for all available methods.

List

Lists are ordered sequences of objects of any type. Painless offers a literal (called list initialization operator by the language spec) for initializing lists:

```
List empty = [];
List list = [1, 2, "three"];
```

Lists initialized this way are mutable. Values can be any expression.

Alternatively, you can initialize lists similar to Java:

```
List empty1 = new ArrayList();
List empty2 = new LinkedList();
```

You can read and update elements using the *list[index]* syntax:

```
List x = [1, 2, 3];
def y = x[0];
x[0] = 10;
```

Using indexes outside of the bounds of the list results in an exception being thrown.

You can add elements to a list using the *add()* method. Check a List's size with *size()*:

```
List x = [1, 2, 3];
x.add(4);
int s = x.size();
```

In Painless, you can use lambda expressions just like in Java. Thus, the stream API can be useful for bulk and aggregate operations on a list.

Calculate a sum of a list of numbers:

```
List list = [1, 2, 3];
int sum = list.stream().reduce(0, (a, b) -> a + b);
```

Calculate a sum of attributes of objects stored inside a list:

```
List list = [[a:1], [a:2], [a:3]];
int sum = list.stream().reduce(0, (a, b) -> a.a + b.a);
```

Sort a list of objects based on attribute values:

```
List list = [[a:1], [a:2], [a:3]];
list.sort((a, b) -> a.a - b.a);
```

Convert a list of strings into a list containing the same strings in upper case.

```
List list = ["a", "b", "c"];
List uppercaseList = list.stream().map(String::toUpperCase).collect(Collectors.toList());
```

See the language spec for all available methods.

Map

Maps are collections of pairs of keys and values. Painless offers a literal for initializing maps:

```
Map empty = [:];
Map map = ["a": "x", "b": "y", "c": "z"];
```

Maps created using the literal are mutable. You can access elements using the syntax *map[key]*:

```
Map map = [:];
map["a"] = "b";
def x = map["a"];
```

Important methods for Maps are *size()*, *keySet()* and *entrySet()*.

See the language spec for all available methods.

Arrays

Besides *List*, Painless also offers array types, which are equivalent to Java arrays.

```
int[] array = new int[10];
int[][] matrix = new int[10][10];
int[] preset = new int[] {1, 2, 3};
```

The advantage of arrays compared to Lists is that multi-dimensional arrays are easier to initialize. Also, primitive types are packed in arrays more densely than in Lists.

Access to array elements works like for lists: *array[index]*. If you specify a negative index, you are accessing the array in reverse order. I.e., the index -1 refers to the last element.

```
int[] array = new int[]{1, 2, 3};
int x = array[0];  // x is 1
int y = array[-1]; // y is 3
```

Arrays cannot be expanded. Trying to access an index that is out of bounds will result in an exception. The number of elements of an array can be retrieved using *array.length*.

Dates

There is quite a variety of different date types available in Painless, like in Java. Thus, Painless also does not have a date literal.

When working with Elasticsearch, you will mostly need the type *ZonedDateTime*.

```
ZonedDateTime date = ZonedDateTime.of(1978, 12, 31, 23, 55, 0, 0, ZoneId.of('Z'));
```

Details can be found in the language spec.

Testing Painless scripts

If you just want to try Painless scripts and view the results, you can use the Elasticsearch REST API endpoint */scripts/painless/execute*.

```
POST /_scripts/painless/_execute

{
  "script": {
    "source": "params.a - 1",
    "params": {
      "a": 43
    }
  }
}
```

Results in:

```
{
    "result": "42"
}
```

Where to go next

- Deep dive into Signals by reading our fine documentation
- Ask any question on our Signals Alerting forum

Image: Shutterstock / Sam72
