**1. Overview**

**Groovy is a dynamic, scripting language for the JVM**. It compiles to bytecode and blends seamlessly with Java code and libraries.

In this article, we’re going to take a look some of the essential features of [Groovy](http://www.groovy-lang.org/), including basic syntax, control structures, and collections.

Then we will look at some of the main features that make it an attractive language, including null safety, implicit truth, operators, and strings.

**2. Environment**

If we want to use Groovy in Maven projects, we need to add the following to the *pom.xml:*

<**build**>

<**plugins**>

// ...

<**plugin**>

<**groupId**>org.codehaus.gmavenplus</**groupId**>

<**artifactId**>gmavenplus-plugin</**artifactId**>

<**version**>1.5</**version**>

</**plugin**>

</**plugins**>

</**build**>

<**dependencies**>

// ...

<**dependency**>

<**groupId**>org.codehaus.groovy</**groupId**>

<**artifactId**>groovy-all</**artifactId**>

<**version**>2.4.10</**version**>

</**dependency**>

</**dependencies**>

The most recent Maven plugin can be found [here](https://mvnrepository.com/artifact/org.codehaus.gmavenplus/gmavenplus-plugin) and the latest version of the *groovy-all* [here](https://mvnrepository.com/artifact/org.codehaus.groovy/groovy-all).

**3. Basic Features**

There are many useful features in Groovy. Now, let's look at the basic building blocks of the language and how it differs from Java.

Now, let's look at the basic building blocks of the language and how it differs from Java.

**3.1. Dynamic Typing**

**One of the most important features of Groovy is its support for dynamic typing.**

Type definitions are optional and actual types are determined at runtime. Let's take a look at these two classes:

**class** **Duck** {

String getName() {

'Duck'

}

}

**class** **Cat** {

String getName() {

'Cat'

}

}

Those two classes define the same *getName*method, but it is not defined explicitly in a contract.

Now, imagine that we have a list of objects containing ducks and cats that have the *getName* method. With Groovy, we can do the following:

Duck duck = **new** Duck()

Cat cat = **new** Cat()

**def** list = [duck, cat]

list.each { obj ->

println obj.getName()

}

The code will compile, and the output of the code above would be:

Duck

Cat

**3.2. Implicit Truthy Conversion**

Like in JavaScript, Groovy evaluates every object to a boolean if required, e.g. when using it inside an *if* statement or when negating the value:

**if**("hello") {...}

**if**(15) {...}

**if**(someObject) {...}

There are a few simple rules to remember about this conversion:

* Non-empty *Collections,*arrays, maps evaluate to *true*
* *Matcher* with at least one match evaluates to *true*
* *Iterators* and *Enumerations* with further elements are coerced to *true*
* Non-empty *Strings*, *GStrings* and *CharSequences*, are coerced to *true*
* Non-zero numbers are evaluated to *true*
* Non-null object references are coerced to *true*

**If we want to customize the implicit truthy conversion, we can define our *asBoolean()* method.**

**3.3. Imports**

Some packages get imported by default, and we don’t need to import them explicitly:

**import** java.lang.\*

**import** java.util.\*

**import** java.io.\*

**import** java.net.\*

**import** groovy.lang.\*

**import** groovy.util.\*

**import** java.math.BigInteger

**import** java.math.BigDecimal

**4. AST Transforms**

AST (**Abstract Syntax Tree**) transforms allows us to hook into the Groovy compilation process and customize it to meet our needs. This is done at compilation time, so there is no performance penalty when running application. We can create our AST transformations, but we can also use the built-in ones.

We can create our transformations, or we can benefit from the built-in ones.

Let's take a look at some annotations worth knowing.

**4.1. Annotation *TypeChecked***

This annotation is used for forcing the compiler to do strict type checking for annotated pieces of code. The type checking mechanism is extensible, so we can even provide even stricter type checking than available in Java when desired.

Let's take a look at the example below:

**class** **Universe** {

@TypeChecked

**int** answer() { "forty two" }

}

If we try to compile this code, we'll observe the following error:

[Static type checking] - Cannot **return** value of type java.lang.String on method returning type **int**

The *@TypeChecked* annotation can be applied to classes and methods.

**4.2. Annotation *CompileStatic***

This annotation allows the compiler to execute compile-time checks as it is done with Java code. After that, the compiler performs a static compilation, thus bypassing the Groovy metaobject protocol.

When a class is annotated, all methods, properties, files, inner classes, etc. of the annotated class will be type-checked. When a method is annotated, static compilation is applied only to those items (closures and anonymous inner classes) that are enclosed by that method.

**5. Properties**

In Groovy, we can create POGOs (Plain Old Groovy Objects) that work the same way as POJOs in Java, although they’re more compact because **getters and setters are automatically generated for public properties** during compilation. It's important to remember that they will be generated only if they’re not already defined.

This gives us the flexibility of defining attributes as open fields while retaining the ability to override the behavior when setting or getting the values.

Consider this object:

**class** **Person** {

String name

String lastName

}

Since the default scope for classes, fields, and methods is *public –*this is a public class, and the two fields are public.

The compiler will convert these into private fields and add *getName()*, *setName()*, *getLastName()* and *setLasfName()* methods. If we define the *setter* and *getter* for a particular field, the compiler will not create a public method.

**5.1. Shortcut Notations**

Groovy offers a shortcut notation for getting and setting properties. Instead of the Java-way of calling getters and setters, we can use a field-like access notation:

resourceGroup.getResourcePrototype().getName() == SERVER\_TYPE\_NAME

resourceGroup.resourcePrototype.name == SERVER\_TYPE\_NAME

resourcePrototype.setName("something")

resourcePrototype.name = "something"

**6. Operators**

Let's now take a look at new operators added on top of those known from plain Java.

**6.1. Null-Safe Dereference**

The most popular one is the null-safe dereference operator*“?”* which allows us to avoid a *NullPointerException*when calling a method or accessing a property of a *null* object. It’s especially useful in chained calls where a *null* value could occur at some point in the chain.

For example, we can safely call:

String name = person?.organization?.parent?.name

In the example above if a *person*, *person.organization*, or *organization.parent*are *null*, then *null* is returned.

**6.2. Elvis Operator**

The Elvis operator*“?:*” lets us condense ternary expressions. These two are equivalent:

String name = person.name ?: defaultName

and

String name = person.name ? person.name : defaultName

They both assign the value of *person.name* to the name variable if it is *Groovy true* (in this case, not *null* and has a *non-zero* length).

**6.3. Spaceship Operator**

The spaceship operator *“<=>”* is a relational operator that performs like Java’s *compareTo()* which compares two objects and returns -1, 0, or +1 depending on the values of both arguments.

If the left argument is greater than the right, the operator returns 1. If the left argument is less than the right, the operator returns −1. If the arguments are equal, 0 is returned.

The greatest advantage of using the comparison operators is the smooth handling of *nulls* such that *x <=> y* will never throw a *NullPointerException*:

println 5 <=> null

The above example will print 1 as a result.

**7. Strings**

There are multiple ways for expressing string literals. The approach used in Java (double-quoted strings) is supported, but it is also allowed to use single quotes when preferred.

Multi-line strings, sometimes called heredocs in other languages, are also supported, using triple quotes (either single or double).

Multi-line strings, sometimes called heredocs in other languages, are also supported, using triple quotes (either single or double).

Strings defined with double quotes support interpolation using the*${}* syntax:

**def** name = "Bill Gates"

**def** greeting = "Hello, ${name}"

In fact, any expression can be placed inside the *${}*:

**def** name = "Bill Gates"

**def** greeting = "Hello, ${name.toUpperCase()}"

A String with double quotes is called a GString if it contains an expression *${}*, otherwise, it is a plain *String*object.

The code below will run without failing the test:

**def** a = "hello"

**assert** a.**class**.name == 'java.lang.String'

**def** b = 'hello'

**assert** b.**class**.name == 'java.lang.String'

**def** c = "${b}"

**assert** c.**class**.name == 'org.codehaus.groovy.runtime.GStringImpl'

**8. Collections and Maps**

Let's take a look at how some basic data structures are handled.

**8.1. *Lists***

Here’s some code to add a few elements to a new instance of *ArrayList*in Java:

List<String> list = **new** **ArrayList**<>();

list.add("Hello");

list.add("World");

And here’s the same operation in Groovy:

List list = ['Hello', 'World']

Lists are by default of type *java.util.ArrayList* and can also be declared explicitly by calling the corresponding constructor.

There isn’t a separate syntax for a *Set*, but we can use type *coercion* for that. Either use:

Set greeting = ['Hello', 'World']

or:

**def** greeting = ['Hello', 'World'] **as** Set

**8.2. *Map***

The syntax for a *Map* is similar, albeit a bit more verbose, because we need to be able to specify keys and values delimited with colons:

**def** key = 'Key3'

**def** aMap = [

'Key1': 'Value 1',

Key2: 'Value 2',

(key): 'Another value'

]

After this initialization, we will get a new *LinkedHashMap*with the entries: *Key1 -> Value1, Key2 -> Value 2, Key3 -> Another Value*.

We can access entries in the map in many ways:

println aMap['Key1']

println aMap[key]

println aMap.Key1

**9. Control Structures**

**9.1. Conditionals: *if-else***

Groovy supports the conditional *if/else* syntax as expected:

**if** (...) {

// ...

} **else** **if** (...) {

// ...

} **else** {

// ...

}

**9.2. Conditionals: *switch-case***

The *switch* statement is backward compatible with Java code so that we can fall through cases sharing the same code for multiple matches.

The most important difference is that a *switch* can perform matching against multiple different value types:

**def** x = 1.23

**def** result = ""

**switch** ( x ) {

**case** "foo":

result = "found foo"

**break**

**case** "bar":

result += "bar"

**break**

**case** [4, 5, 6, 'inList']:

result = "list"

**break**

**case** 12..30:

result = "range"

**break**

**case** Number:

result = "number"

**break**

**case** ~/fo\*/:

result = "foo regex"

**break**

**case** { it < 0 }: // or { x < 0 }

result = "negative"

**break**

default:

result = "default"

}

println(result)

The example above will print *number.*

**9.3. Loops: *while***

Groovy supports the usual *while*loops like Java does:

**def** x = 0

**def** y = 5

**while** ( y-- > 0 ) {

x++

}

**9.4. Loops: *for***

Groovy embraces this simplicity and strongly encourages *for* loops following this structure:

**for** (variable **in** iterable) { body }

The *for* loop iterates over *iterable*. Frequently used iterables are ranges, collections, maps, arrays, iterators, and enumerations. In fact, any object can be an iterable.

Braces around the body are optional if it consists of only one statement. Below are examples of iterating over a *range*, *list*, *array*, *map*, and *strings*:

**def** x = 0

**for** ( i **in** 0..9 ) {

x += i

}

x = 0

**for** ( i **in** [0, 1, 2, 3, 4] ) {

x += i

}

**def** array = (0..4).toArray()

x = 0

**for** ( i **in** array ) {

x += i

}

**def** map = ['abc':1, 'def':2, 'xyz':3]

x = 0

**for** ( e **in** map ) {

x += e.value

}

x = 0

**for** ( v **in** map.values() ) {

x += v

}

**def** text = "abc"

**def** list = []

**for** (c **in** text) {

list.add(c)

}

Object iteration makes the Groovy *for-*loop a sophisticated control structure. It’s a valid counterpart to using methods that iterate over an object with closures, such as using *Collection’s each* method.

The main difference is that the body of a *for*loop isn’t a closure, this means this body is a block:

**for** (x **in** 0..9) { println x }

whereas this body is a closure:

(0..9).each { println it }

Even though they look similar, they’re very different in construction.

A closure is an object of its own and has different features. It can be constructed in a different place and passed to the *each*method. However, the body of the *for-*loop is directly generated as *bytecode* at its point of appearance. No special scoping rules apply.

**10. Exception Handling**

The big difference is that checked exceptions handling is not enforced.

To handle general exceptions, we can place the potentially exception-causing code in a *try/catch* block:

**try** {

someActionThatWillThrowAnException()

} **catch** (e)

// log the error message, and/or handle in some way

}

By not declaring the type of exception we catch, any exception will be caught here.

**11. Closures**

Simply put, a closure is an anonymous block of executable code which can be passed to variables and has access to data in the context where it was defined.

They’re also similar to anonymous inner classes, although they don’t implement an interface or extend a base class. They are similar to lambdas in Java.

Interestingly, Groovy can take full advantage of the JDK additions that have been introduced to support lambdas, especially the streaming API. We can always use closures where lambda expressions are expected.

Let's consider the example below:

**def** helloWorld = {

println "Hello World"

}

The variable *helloWorld* now holds a reference to the closure, and we can execute it by calling its *call*method:

helloWorld.call()

Groovy lets us use a more natural method call syntax – it invokes the *call*method for us:

helloWorld()

**11.1. Parameters**

Like methods, closures can have parameters. There are three variants.

In the latter example, because there’s nothing declpersistence\_startared, there is only one parameter with the default name *it*. The modified closure that prints what it is sent would be:

**def** printTheParam = { println it }

We could call it like this:

printTheParam('hello')

printTheParam 'hello'

We can also expect parameters in closures and pass them when calling:

**def** power = { **int** x, **int** y ->

**return** Math.pow(x, y)

}

println power(2, 3)

The type definition of parameters is the same as variables. If we define a type, we can only use this type, but can also it and pass in anything we want:

**def** say = { what ->

println what

}

say "Hello World"

**11.2. Optional Return**

The last statement of a closure may be implicitly returned without the need to write a return statement. This can be used to reduce the boilerplate code to a minimum. Thus a closure that calculates the square of a number can be shortened as follows:

**def** square = { it \* it }

println square(4)

This closure makes usage of the implicit parameter *it* and the optional return statement.

**12. Conclusion**

This article provided a quick introduction to the Groovy language and its key features. We started by introducing simple concepts such as basic syntax, conditional statements, and operators. We also demonstrated some more advanced features such as operators and closures.