Groovy with Eclipse - Tutorial

Lars Vogel

Version 2.2 **Groovy**

This article gives a short overview of the Groovy language including collections, loops, gstrings, MOP, closures, operator overloading, XML handing and using Groovy together with Java class. It also describes how to use the Eclipse IDE for developing Groovy.

This article assumes that you have already Eclipse installed and that you have used Eclipse for Java development. This article was written using Groovy 2.4, Eclipse 4.4 (Luna) and Java 1.8.

---

1. Groovy

1.1. What is Groovy?

Groovy is an optionally typed, dynamic language that runs on the JVM and is tightly integrated with the Java programming language. Groovy source code is compiled into Java byte-code; Groovy does not follow the approach to generate source code nor does the Groovy runtime interpret Groovy code at runtime. To run Groovy code in a Java virtual machine, only the Groovy `JAR` file must be present in the classpath at runtime.

Groovy supports standard Java constructs including annotations, generics, static imports, enums, varargs and lambda expression.

It provides lots of simplifications compared to the Java programming language and advanced language features as properties, closures, dynamic methods, the Meta Object Protocol (MOP), native support for lists, maps, regular expressions, duck typing and the elvis operator.

Groovy describes itself as feature-rich and Java-friendly language.

1.2. Groovy classes and scripts

A Groovy source files ends with the `.groovy` extension. This file can contain a Groovy script or a Groovy class. A Groovy script is a code listing which does not include a class definition. Groovy scripts are converted at compile time to a class which extends the `groovy.lang.Script` class.

The classical "Hello world" program can be written as a short Groovy script.

```
println 'Hello World'
```

1.3. Compatibility with Java

Groovy runs inside the JVM and can use Java libraries. Every Groovy type is a subclass of `java.lang.Object`.

Groovy code can call Java code and Java code can call Groovy code. Every Groovy class is compiled into a Java class and you can use the `new` operator in Java to create instances of the Groovy class. This instance can be used to call methods or to pass as parameter to a fitting Java method. Groovy classes can extend Java classes and Java classes can also extend Groovy classes.

Groovy is almost compatible to Java, e.g., almost every Java construct is valid Groovy code. This makes the migration to Groovy for a Java programmer relatively smooth.

1.4. Reasons to use Groovy

Groovy focus on simplicity and ease of use as its leading principle. This makes using Groovy very productive.

The enhancements of Groovy compared to Java can be classified as:

- Groovy language features
- Groovy specific libraries
- Additional methods to existing Java classes by the Groovy Developer Kit, this is commonly known as the Groovy JDK.

The following list contains some of the example how Groovy archives this.

- Simplification - Groovy does not require semicolons at the end of statements. The `return` keyword can be left out, by default Groovy returns the last expression of the method, top level parentheses can be left out, the `public` keyword can be left out, it is the default in Groovy. It also allows optional typing.
- Flexibility - Groovy allows to change classes and methods at runtime, e.g. if a method is called which does not exist on a class, the class can intercept this call and react to it. This allows for example that Groovy provides a very flexible builder pattern.
- Ease of use - Groovy has list, maps and regular expressions directly build into the language.
- Simplification in I/O - parsing and creating XML, JSON and files is very simple with Groovy.

1.5. Imports in Groovy

Groovy automatically imports the following packages and classes which can be used in Groovy without specifying the package name.

- groovy.lang.*
- groovy.util.*
- java.lang.*
- java.util.*
- java.net.*
- java.io.*
- java.math.BigInteger
- java.math.BigDecimal

Groovy allows that an import is shortened for later access, e.g., `import javax.swing.WindowConstants as WC`.

1.6. IDE support for Groovy

Pivotal, the company behind the popular Spring framework and the Groovy programming language provides the *GROOVY/GRAILS TOOL SUITE* which is a complete Eclipse-based development environment optimized for developing, debugging and executing Groovy and Grails applications.

Other IDEs like Netbean and IntelliJ also provide great Groovy development support.

2. Installation of Groovy

2.1. Install Groovy for the command line

To be able to run Groovy code from the command line download the latest version from Groovy from the Groovy download website.

Download at least the binary zip file and extract it to a directory on your hard disk. Afterwards set the GROOVY_HOME environment variable and %GROOVY_HOME%/bin to your path.

If you are using MS Windows you can use the Windows installer. This installer configured the environment variables automatically for you.

2.2. Download the GROOVY/GRAILS TOOL SUITE

Download the GROOVY/GRAILS TOOL SUITE from the Spring tools download side.

2.3. Optional: Installing the Groovy plug-in into an existing Eclipse IDE

You can also install the Groovy tooling into an existing Eclipse installation. The update site is Eclipse version dependent, see Groovy/Grails Tool Suite™ Downloads if you use a different release than Eclipse 4.4.

Open the Eclipse Update manager via the Help → Install New Software... menu entry to install the Groovy Eclipse plug-in. Enter the following URL in this dialog:

```
http://dist.springsource.com/release/TOOLS/update/e4.4/
```

2.4. More information about Groovy programming with Eclipse

The Groovy plug-in has a FAQ. The Eclipse Groovy bug tracker is located under the following URL GRECLIPSE bugtacker .

3. Exercise: Create and run first Groovy program

3.1. Create a new Groovy project

The following example assumes you have Groovy and the Eclipse IDE installed and configured.

Create a new Groovy project called *com.vogella.groovy.first* the File → New → Other →

Groovy → Groovy Project. After entering the project name, press the Finish button. This creates a new Groovy project similar to a new Java project but with the required Groovy

libraries.

Right click on the source folder and New → Package from the context menu. Create a new package called `first`.

Create a new Groovy class called `FirstGroovy` via File → New → Other → Groovy → Groovy Class.

Create the following code.

```
package first

class FirstGroovy {

  static void main(def args){
    def mylist= [1,2,"Lars","4"]
    mylist.each{ println it }
  }
}
```

3.2. Run the Groovy class

Right-click the Groovy class, and select Run As → Groovy Script from the context menu.

3.3. Validate the Groovy class

The above menu entry triggers the execution of the main method in your Groovy class and prints output to *Console* view of the Eclipse IDE.

Congratulation! You created and ran your first Groovy class.

3.4. Decompile the Groovy class file

Note

This exercise is optional.

If you are familiar with a Java decompiler, like the JadClipse plug-in, decompile the class file generated from your Groovy source file and review it.

See Use javap to decompile for a command line option.

4. Groovy classes, objects and methods

4.1. A Groovy class and default access modifier

A Groovy class is defined with the `class` keyword, similar to Java. All Groovy classes and methods are by default public.

The following is an example Groovy class called `Task.groovy`.

```
package com.vogella.groovy.first

class Task {
  String summary
  String description
  Date dueDate
}
```

4.2. Groovy objects (Plain Old Groovy Objects) and fields

In Groovy all fields of a class have by default the `private` access modifier. Groovy creates automatically getter and setter methods for the fields. Therefore all Groovy classes are per default *Java beans*.

Groovy objects are frequently referred to as *Plain Old Groovy Objects (POGO)*.

You can use the getter and setter directly or use the name of the field for access. Groovy also supports the array subscript acessor (object[property]). Groovy uses the getter or setter method, even if you directly use the name of the field. If a field should not be changeable define it as `final`, in this case Groovy will not provide a setter.

4.3. Constructors

Groovy provides *constructors with named parameters* in which you can specify the element you would like to set during construction. This constructor is also called *map based constructor*, as it uses the `property:value` map syntax.

If such a constructor is used, Groovy calls the default constructor and then calls the setter methods for the attributes. This "constructor with named parameters" works also if you call a Java class from Groovy code as Groovy uses again the default constructor of the Java class and then the methods to set the properties.

The usage of the constructors with named parameters is demonstrated by the following example.

```
package com.vogella.groovy.first

public class Person{
  String firstName
  String lastName
  int age
  def address

  static void main(def args) {
    Person p = new Person()
    // use the generated access methods
    p.setFirstName("Lars")
    // this will still use the generated access method, it is not a direct
access!
    p.lastName = "Vogel"
    p.address = ("Homestreet 3");
    println(p.firstName + " " + p.lastName);
    // use the generated constructor
    p = new Person(firstName: "Peter", lastName:"Mueller");
    println(p.firstName + " " + p.lastName);
  }

}
```

4.4. Equals, == and the method is()

One difference between Java and Groovy is that the == operator will check for equality and not for identity. Java checks if both variables points to the same object while Groovy checks if both variables are equals. To check for identify you can use in Groovy the is() method.

In Groovy null == null returns true. If two references point to the same object it is also true. If an object implements the `compareTo` method, Comparable this method is used, otherwise the `equals` method.

4.5. Optional parameters in methods

Groovy allows to have optional parameter values. Optional parameter values are indicated by =0.

```
class Hello {

static main(args){
  println sum(1,5)
  println sum(1,2,5)
  }

static sum(a,b,c=0){
  a+b+c;
  }
}
```

4.6. Default parameters in methods

In Groovy you assign default values to parameters in a method. If a default value for a parameter is defined, Groovy offers two method signatures: one with all parameters and one where the parameter with a default value is omitted. If you use multiple parameters with default values then the right most parameter with a default value is first eliminated then the next, etc.

5. GPath

GPath is a path expression language integrated into Groovy which allows parts of nested structured data to be identified. In this sense, it has similar aims and scope as XPath does for XML. The two main places where you use GPath expressions is when dealing with nested POJOs or when dealing with XML.

For example the `a.b.c` statement is equivalent to `a.getB().getC()`.

GPath navigation works also in complex structures like XML or JSON.

6. Groovy data types

6.1. Optional typed variables

Variables and fields can be typed as in Java or your can use the `def` keyword to define a variable. As a rule of thumb, use the type if it adds clarity to your code otherwise use `def`.

```
// valid variable definitions

// typed
String name
int x
Integer y

// untyped
def list
def map
def todo
```

At runtime variables and fields are always typed, Groovy infers the type based on your source code. This means that at runtime you receive an error if you try to assign a non fitting type to a variable.

Variables which are not declared can be used in Groovy scripts to indicate that they can be set from outside. Such declarations are only valid in scripts and become part of the scripts binding.

6.2. All types are objects

All variables in Groovy are objects (reference variables), Groovy does not use primitive variables. Groovy still allows to use the primitive's types as a short form for the variable declaration but the compiler translates this into the object.

6.3. Numbers

Numbers are objects in Groovy, as well as variables defined as int, float, double, etc. If you use numbers in your code Groovy assigns a type to it and performs automatically the down- and up casting for you.

As numbers are object they have also methods for example the `times` method which executes a block of code the number of times defined by the number.

Create the following class called `TypesTest` to play with numbers.

```
package example

int i = 1 // Short form for Integer i = new Integer(1)
int j = i +3
int k = i.plus(3) // Same as above
// Make sure this worked
assert(k==4)
println i.class
println j.class
println k.class

// Automatic type assignement
def value = 1.0F
println value.class
def value2 = 1
```

```
println value2.class
// this would be zero in Java
value2 = value2 / 2
println value2
// value was upcasted
println value2.class

10.times {println "Test"}
```

The operators, like + or - are also mapped to methods by Groovy.

Table 1.

Operator	Name	Method
a+b	plus	a.plus(b)
a-b	minus	a.minus(b)
a*b	star	a.multiply(b)
a/b	divide	a.div(b)
a%b	modulo	a.mod(b)
a--, --a	decrement	a.previous()
a++, ++a	increment	a.next()
a**b	power	a.power(b)
a-b	minus	a.minus(b)
a-b	minus	a.minus(b)

6.4. Ranges

Groovy supports the `Range` data type is a `Collection`. Ranges consists of two values separated by two dots. Ranges can for example be used to define a loop statement.

```
package de.vogella.groovy.datatypes

for (i in 0..9) {
  println ("Hello $i")
}
assert 'B'..'E' == ['B', 'C', 'D', 'E']
```

You can use Strings for the definition of ranges, these ranges follow the alphabet.

Every object can be used as `Range` long as it implements the `previous()` and `next()` methods and the `java.util.Comparable` interface. The methods map to the ++ and -- operators.

7. Operator overloading

Groovy supports that you can use the standard operations in your own classes. For example if you want to use the operation a+b where a and b are from class Z then you have to implement the method plus(Z name) in class Z.

Groovy will map the operations to the following classes.

Table 2.

Operator	Name	Method
a+b	plus	a.plus(b)
a-b	minus	a.minus(b)
a*b	star	a.multiply(b)
a/b	divide	a.div(b)
a%b	modulo	a.mod(b)
a--, --a	decrement	a.previous()
a++, ++a	increment	a.next()
a**b	power	a.power(b)
a-b	minus	a.minus(b)
a-b	minus	a.minus(b)

8. Strings in Groovy

8.1. Strings and GStrings

Groovy allows to use two different types of String, the `java.lang.String` and the `groovy.lang.GString` class. You can also define a single line or a multi-line string in Groovy.

Strings which are quoted in by "" are of type `GString` (short for Groovy Strings). In GStrings you can directly use variables or call Groovy code. The Groovy runtime evaluates the variables and method calls. An instance of `GString` is automatically converted to a `java.lang.String` whenever needed.

```
package com.vogella.groovy.strings

def name = "John"
def s1 = "Hello $name" // $name will be replaced
def s2 = 'Hello $name' // $name will not be replaced
println s1
println s2
```

```
println s1.class
println s2.class

// demonstrates object references and method calls
def date = new Date()
println "We met at $date"
println "We met at ${date.format('MM/dd/yy')}"
```

The definition of these different types of Strings is demonstrated in the following table.

Table 3. Define Strings in Groovy

String example	Description
'This is a String'	Standard Java String
"This is a GString"	Groovy GString, allows variable substitution and method calls
''' Multiline string (with line breaks)'''	A multi line string
""" Multiline string (with line breaks)"""	A multi line GString
/regularexpression/	Forward Slash – Escape backslashes ignored, makes Regular Expressions more readable

The `tokenize()` method tokenize the String into a list of String with a whitespace as the delimiter.

The Groovy JDK adds the `toURL()` method to String, which allows to convert a String to a URL.

The `trim` method removes is applied to remove leading and trailing whitespace.

8.2. Operator overloading in Strings

String support operator overloading. You can use + to concatenate strings, - to substract strings and the *left-shift operator* to add to a String.

9.1. Regular expressions

Groovy is based on Java regular expression support and add the following operators to make the usage of regular expressions easier:

Table 4.

Construct	Description
=~	Find: True if the pattern is contained in a text
==~	Match: True if the complete string matches the pattern
~String	Turns a string into a regular expression

If you use the ~ operator such a string turns into a regular expression which can be used for pattern matching. You can use special sign (escape characters) in Strings if you put them between slashes.

```
package de.vogella.groovy.datatypes

public class RegularExpressionTest{
  public static void main(String[] args) {
    // Defines a string with special signs
    def text = "John Jimbo jingeled happily ever after"

    // Every word must be followed by a nonword character
    // Match
    if (text==~/(\w*\W+)*/){
      println "Match was successful"
    } else {
      println "Match was not successful"
    }
    // Every word must be followed by a nonword character
    // Find
    if (text=~/(\w*\W+)*/){
      println "Find was successful"
    } else {
      println "Find was not successful"
    }

    if (text==~/^J.*/){
      println "There was a match"
    } else {
      println "No match found"
    }
    def newText = text.replaceAll(/\w+/, "hubba")
    println newText
  }


}
```

10. Lists

10.1. Defining and accessing lists

Groovy treads lists as first class constructs in the language. You define a list via `List list = new List[]`. You can also use generics. To access element i in a list you can either use list.get(i) or list[i].

```
package de.vogella.groovy.datatypes

public class Person{
  String firstName;
  String lastName;
  Person(String firstName, String lastName){
    this.firstName = firstName
    this.lastName= lastName
  }
}
package de.vogella.groovy.datatypes

public class ListMapTest{

  public static void main(args){
    List<Integer> list = [1,2,3,4]
    println list[0]
    println list[1]
    println list[2]
    List<Person> persons = list[]
    Person p = new Person("Jim", "Knopf")
    persons[0] = p
    println persons.size()
    println persons[0].firstName
    println persons.get(0).firstName
  }
}
```

Groovy allows direct property access for a list of items. This is demonstrated by the following snippet.

```
package de.vogella.groovy.datatypes

public class ListMapTest{

  public static void main(args){
    List<Person> persons = list[]
    persons[0] = new Person("Jim", "Knopf")
    persons[1] = new Person("Test", "Test")
    println persons.firstName
  }
}
```

10.2. List methods

The following lists the most useful methods on List.

- reverse()
- sort()
- remove(index)
- findAll{closure} - returns all list elements for which the closure validates to true
- first()
- last()
- max()
- min()
- join("string") - combines all list elements, calling the toString method and using the string for concatenation.
- << e - appends element e to the list

The `grep` method can be used to filter elements in a collection.

10.3. Operator overloading in Lists

List support operator overloading. You can use + to concatenate strings, - to substract lists and the *left-shift operator* to add elements to a list.

10.4. Spread-dot operator

The spread-dot operator `*.` is used to invoke a method on all members of a Collection. The result of this operation is another Collection object.

```
def list = ["Hello", "Test", "Lars"]

// calculate the lenght of every String in the list
def sizeList = list*.size()
assert sizeList = [5, 4, 4]
```

10.5. Searching in a list with find, findall and grp

You can search in a list.

- findAll{closure} - returns all list elements for which the closure validates to true
- find{closure} - returns the list element for which the closure validates to true
- grep(Object filter) - Iterates over the collection of items and returns each item that matches the given filter - calling the Object#isCase. This method can be used with different kinds of filters like regular expressions, classes, ranges etc.

```
package list

def l1 = ['test', 12, 20, true]
```

```
// check with grep that one element is a Boolean
assert [true] == l1.grep(Boolean)

// grep for all elements which start with a pattern
assert ['Groovy'] == ['test', 'Groovy', 'Java'].grep(~/^G.*/)

// grep for if the list contains b and c
assert ['b', 'c'] == ['a', 'b', 'c', 'd'].grep(['b', 'c'])

// grep for elements which are contained in the range
assert [14, 16] == [5, 14, 16, 75, 12].grep(13..17)

// grep for elements which are equal to 42.031
assert [42.031] == [15, 'Peter', 42.031, 42.032].grep(42.031)

// grep for elements which are larger than 40 based on the closure
assert [50, 100, 300] == [10, 12, 30, 50, 100, 300].grep({ it > 40 })
```

11. Maps in Groovy

11.1. Map declaration and access

Groovy treads also maps as first class constructs in the language.

The items of maps are key–value pairs that are delimited by colons. An empty map can be created via [:]. By default a map is of the `java.util.HashMap` type. If the keys are of type String, you can avoid the single or double quotes in the map declaration.

```
package com.vogella.groovy.maps

class Main {

  static main(args) {
    // create map
    def map = ["Jim":"Knopf", "Thomas":"Edison"]

    // create map without quotes for the keys
    def anotherMap = [Jim:"Knopf", Thomas:"Edison"]
    // size is used to determine the number of elements
    assert create.size() == 2

    // if key should be evaluated put it into brackets
    def x ="a"
    // not true, as x is interpreted as "x"
    println ([a:1]==[x:1])
    // force Groovy to see x as expression
    println ([a:1]==[(x):1])

    // create empty map
    def emptyMap = [:]

     }
}
```

The values of a mapped value can get accessed via map[key]. Assignment can be done via map[key]=value. You can also call `get(key)` or `get(key,default)`. In the second case, if the key is not found and the default is returned, the (key,default) pair is added to the map.

```
package com.vogella.groovy.maps

class Main {

  static main(args) {
    // create map
    def map = ["Jim":"Knopf", "Thomas":"Edison"]

    // create map without quotes for the keys
    def anotherMap = [Jim:"Knopf", Thomas:"Edison"]
    // size is used to determine the number of elements
    assert create.size() == 2

    // if key should be evaluated put it into brackets
    def x ="a"
    // not true, as x is interpreted as "x"
    println ([a:1]==[x:1])
    // force Groovy to see x as expression
    println ([a:1]==[(x):1])

    // create empty map
    def emptyMap = [:]



  }

}
```

The `keySet()` method returns a set of keys, a collection without duplicate entries and no guaranteed ordering.

11.2. Each, any and the every method

You can call closures on the elements, via the `each()`, `any()` and `every()` method. The `any()` and `every()` methods return a boolean depending whether any or every entry in the map satisfies a condition defined by a closure.

```
package com.vogella.groovy.maps

class CallMethods {

  static main(args) {
    def mymap = [1:"Jim Knopf", 2:"Thomas Edison", 3:"Lars Vogel"]
    mymap.each {entry -> println (entry.key > 1)}
    mymap.each {entry -> println (entry.value.contains("o"))}
    println "Lars contained:" + mymap.any {entry ->
entry.value.contains("Lars")}
    println "Every key small than 4:" + mymap.every {entry -> entry.key < 4}
```

```
    def result =''
    for (key in mymap.keySet()) {
      result += key
    }
    println result

    mymap.each { key, value ->
      print key + " "
      println value
    }

    mymap.each { entry ->
      print entry.key + " "
      println entry.value
    }
  }
}
```

As you can see in the above example you can iterate in different ways through a map. The parameter for each can by one parameter and than it is the map entry or two in which case it is the key, value combination.

11.3. Searching in a map

You can also use the following methods:

- findAll(closure) - Finds all entries satisfying the condition defined by the closure
- find(closure) - Find the first entry satisfying the condition defined by the closure
- collect(closure) - Returns a list based on the map with the values returned by the closure
- submap('key1', 'key2',) - returns a map based on the entries of the listed keys

11.4. Getting and adding defaults values via the get method

The `get(key, default_value)` allows to add the "default_value" to the map and return it to the caller, if the element identified by "key" is not found in the map. The `get(key)` method, does not add automatically to the map.

11.5. Named arguments for method invocation

It is possible to use named arguments in method invocation.

```
package namedarguments

def address = new Address(street: 'Reeperbahn', city: 'Hamburg')
def p = new Person(name: 'Lars', address: address, phoneNumber: '123456789')

// Groovy translates the following call to:
// p.move([street: 'Saselbeck', city: 'Hamburg'], '23456789')
p.moveToNewPlace(street: 'Saselbeck', '23456789', city: 'Hamburg')
```

```
assert 'Lars' == p.name
assert 'Hamburg' == p.address.city
assert 'Saselbeck' == p.address.street
assert '23456789' == p.phoneNumber
```

All named arguments are used are converted by Groovy them a map and passed into the method as first parameter. All other parameters are passed in afterwards. The method can now extract the parameter from the map and perform its setup.

```
package namedarguments

class Address {
    String street, city
}

class Person {
    String name
    Address address
    String phoneNumber

    def moveToNewPlace(inputAsMap, newPhoneNumber) {
        address.street = inputAsMap.street
        address.city   = inputAsMap.city
        phoneNumber = newPhoneNumber

    }
}
```

11.6. Convert a list to a map

To convert a list to a map you can use the `collectEntries` method.

```
package com.vogella.groovy.maps

def words = ['Ubuntu', 'Android', 'Mac OS X', 'Windows']

// simple conversion
def result = words.collectEntries {
  [(it):0]
}

assert result.size() == 4
assert result.Ubuntu == 0

// now calculate value with a closure, true if word contains "n"
def map = words.collectEntries {
  [(it): it.contains('n')]
}

println map
assert map.Ubuntu && map.Windows && map.Android && !map.'Mac OS X'
```

12. Control structures

12.1. The Groovy truth for evaluating conditions

Groovy evaluates a condition defined in a control statement differently from Java. A boolean expression is evaluated the same as in Java, but empty collections or null evaluates to `false`. The number "0" evaluates to `true`, all other numbers evaluates to `true`.

```
package example

map = [:]
assert !map

list = ["Ubuntu", "Android"]
assert list
assert !0
assert 1
assert -1
assert !""
assert "Hello"
def test = null
assert !test
```

Note

This evaluation is commonly known in the Groovy worlds as *the Groovy truth*.

12.2. if statements

The `if` and `switch` are supported, the if statement supports the Groovy truth, e.g., you can use for example a list as parameter in `if` and Groovy will evaluate this Groovy truth value.

12.3. switch statement and the isCase method

The switch statement is very flexible, everything which implements the `isCase` method can be used as classifier.Groovy provides an implementation of the `isCase()` method to `Class` (using isInstance), `Object` (using (equals), `Collections` (using contains) and regular expressions (using matches). You can also specify a closure, which is evaluated to a boolean value.

```
def testingSwitch(input) {
    def result
    switch (input) {
        case 51:
            result = 'Object equals'
            break
        case ~/^Regular.*matching/:
            result = 'Pattern match'
            break
        case 10..50:
            result = 'Range contains'
```

```
            break
        case ["Ubuntu", 'Android', 5, 9.12]:
            result = 'List contains'
            break
        case { it instanceof Integer && it < 50 }:
            result = 'Closure boolean'
            break
        case String:
            result = 'Class isInstance'
            break
        default:
            result = 'Default'
            break
    }
    result
}

assert 'Object equals' == testingSwitch(51)
assert 'Pattern match' == testingSwitch("Regular pattern matching")
assert 'Range contains' == testingSwitch(13)
assert 'List contains' == testingSwitch('Ubuntu')
assert 'Closure boolean' == testingSwitch(9)
assert 'Class isInstance' == testingSwitch('This is an instance of String')
assert 'Default' == testingSwitch(200)
```

If several conditions fit, the first `case` statement is selected.

To use your custom class in a switch statement implement the `isCase` method.

12.4. Safe Navigation Operator

You can use safe navigation operator to check safety for null via the `?.` operator. This will avoid a NullPointerException if you access properties of a variable which is actually null.

```
// firstName will be null if user is null
// no NPE
def firstName = user?.firstName
```

12.5. Elvis operator

The `?:` (called the Elvis operator) is a short form for the Java ternary operator. You can use this to set a default if an expression resolves to false or null.

```
// if user exists, return it, otherwise create a new User

// Java version
String user = null;
String result1 = user!=null ? user : new String();

// Groovy
String test = null
String result2 = test ?: new String()
```

13. Loops

13.1. For and while loops

Groovy supports the standard Java `for`, the `for-each` and the `while` loop. Groovy does not support the `do while` loop.

13.2. Using the each method

While for and while loops are supported the Groovy way of iterating throw a list is using the `each()` method. Groovy provides this method on several objects include lists, maps and ranges.

This method takes as argument a closure, a block of code which can directly get executed. You can either directly define the name of the variable which the value of each iteration should get assigned to or using the implicit available variable "it".

```
package de.vogella.groovy.loops


public class PrintLoop{

  public static void main(def args){

    def list = ["Lars", "Ben", "Jack"]

    // using a variable assignment

    list.each{firstName->

      println firstName

    }

    // using the it variable

    list.each{println it}

  }

}
```

Groovy provides also the `eachWithIndex` method which provides two parameters, the first is the element and the second it the index.

13.3. Iterative with numbers

In additional your have the methods `upto()`, `downto()`, `times()` on number variables. Also you can use ranges (this is an additional datatype) to execute certain things from a number to another number. This is demonstrated by the following example.

```
package de.vogella.groovy.loops

public class LoopTest{
  public static void main(args){
       5.times {println "Times + $it "}
       1.upto(3) {println "Up + $it "}
       4.downto(1) {print "Down + $it "}
       def sum = 0
       1.upto(100) {sum += it}
       print sum
       (1..6).each {print "Range $it"}
  }
}
```

14. Closures

14.1. What are closures?

Closures are *code fragments* or *code blocks* which can be used without being a method or a class.

A closure is defined via `{list of parameters-> closure body}`. The values before the `->` sign define the parameters of the closure. For the case that only one parameter is used you can use the implicit defined variable `it`.

The last statement of a closure is returned as return value.

14.2. Defining default values in a closure

If you define a closure you can also define default values for its parameters.

```
package closures

def multiply = {int a, int  b = 10 -> a * b}

assert multiply(2) == 20
assert multiply(2,5) == 10
```

14.3. Example: Using closures in the each method

```
// return the input, using the implicit variable it
def returnInput = {it}

assert 'Test' = returnInput('Test')

// return the input without implicit variable
def returnInput2 = {s-> s}

assert 'Test' = returnInput2('Test')
```

The Groovy collections have several methods which accept a closure as parameter, for example the each method.

```
List<Integer> list = [5,6,7,8]
list.each({line -> println line})
list.each({println it})


// calculate the sum of the number up to 10

def total = 0
(1..10).each {total+=it}
```

14.4. Using the with method

Every Groovy object has a `with` method which allows to group method and property calls to an object. The `with` method gets a closure as parameter and every method call or property access in this closure is applied to the object.

```
package withmethod

class WithTestClass {
  String property1
  String property2
  List<String> list = []
  def addElement(value) {
    list << value
  }
  def returnProperties () {
    "Property 1: $property1, Property 2: $property2"
  }
}

def sample = new WithTestClass()
def result= sample.with {
  property1 = 'Input 1'
  property2 = 'This is cool'
  addElement 'Ubuntu'
  addElement 'Android'
  addElement 'Linux'
  returnProperties()
```

```
}
println result
assert 3 == sample.list.size()
assert 'Input 1' == sample.property1
assert 'This is cool' == sample.property2
assert 'Linux' == sample.list[2]


def sb = new StringBuilder()
sb.with {
  append 'Just another way to add '
  append 'strings to the StringBuilder '
  append 'object.'
}
```

15. Groovy and concurrency

GPars adds support for parallel processing to Groovy. It supports Actors, Map/Reduce, Dataflow, Fork/Join. You find more information on the GPars website.

16. Groovy and processing files and HTTP get requests

16.1. Groovy and processing files

Processing files with Groovy is simple. The following example will first print out every line to the console and then print again every line to the console with a leading line number.

```
// write the content of the file to the console
File file = new File("./input/test.txt")
file.eachLine{ line -> println line }

// adds a line number in front of each line to the console
def lineNumber = 0
file = new File("./input/test.txt")
file.eachLine{ line ->
  lineNumber++
  println "$lineNumber: $line"
}

// read the file into a String
String s = new File("./input/test.txt").text
println s
```

The `File` object provides also methods like `eachFile`, `eachDir` and `earchFileRecursively` which takes an closure as argument.

16.2. Writing to files

You can also easily write to a file and append to a file.

```
// write the content of the file to the console
File file = new File("output.txt")
file.write "Hello\n"
file.append "Testing\n"
file << "More appending...\n"
File result = new File("output.txt")
println (result.text)
// clean-up
file.delete()
```

16.3. Groovy and processing HTTP get requests

Reading an HTTP page is similar simple.

```
def data = new URL(http://www.vogella.com).text
```

17. Using template engines in Groovy

A template is some text with predefined places for modificatoins. This template can contain variable reference and Groovy code. The templates engines from Groovy provide `createTemplate` methods for Strings, Files, Readers or URL and create a `Template` object based on their input.

Template objects are used to create the final text. A map of key values is passed to the `make` method of the template which return a `Writable`.

```
package template

import groovy.text.SimpleTemplateEngine


 String templateText = '''Project report:

We have currently ${tasks.size} number of items with a total duration of
$duration.
<% tasks.each { %>- $it.summary
<% } %>

'''

def list = [
  new Task(summary:"Learn Groovy", duration:4),
  new Task(summary:"Learn Grails", duration:12)]
def totalDuration = 0
list.each {totalDuration += it.duration}
def engine = new SimpleTemplateEngine()
def template = engine.createTemplate(templateText)
def binding = [
duration: "$totalDuration",
tasks: list]

println template.make(binding).toString()
```

18. Groovy builders

Groovy supports the builder pattern to create tree-like data structures. The base class for the builder support is `BuilderSupport` and its subclasses are `NodeBuilder`, `MarkupBuilder`, `AntBuilder` and `SwingBuilder`.

19. Groovy and Markup like XML or HTML

19.1. Parsing XML with XmlSlurper

Groovy allows to process XML very easily. Groovy provide the `XmlSlurper` class for this purpose. There are other options but the `XmlSlurper` is usually considered to be the more efficient in terms of speed and flexibility. XmlSlurper can also be used to transform the XML white parsing it.

`XmlSlurper` allows to parse an XML document and returns an `GPathResult` object. You can use `GPath` expressions to access nodes in the XML tree.

XMLParser allows to parse an XML document and returns an `groovy.util.Node` object. You can use `GPath` expressions to access nodes in the XML tree. Dots traverse from parent elements to children, and @ signs represent attribute values.

```
package mypackage

public class XmlSluperTest{
    static void main(args){
        def xmldocument = '''
        <persons>
            <person age="3">
        <name>
          <firstname>Jim</firstname>
          <lastname>Knopf </lastname></name>
            </person>
            <person age="4">
        <name>
          <firstname>Ernie</firstname>
          <lastname>Bernd</lastname></name>
            </person>
        </persons>
        '''

        // in case you want to read a file
        // def persons = new XmlSlurper().parse(new File('data/plan.xml'))
        def persons = new XmlSlurper().parseText(xmldocument)
        def allRecords = persons.person.size()

        // create some output
        println("Number of persons in the XML documents is: $allRecords")
        def person = persons.person[0]
        println("Name of the person tag is: ${person.name}")
```

```
        // Lets print out all important information
        for (p in persons.person){
            println "${p.name.firstname.text()}  ${p.name.lastname.text()} is
${p.@age} old"
        }
    }
}
```

19.2. Using the MarkupTemplateEngine to generated Markup

Introduce in Groovy 2.3 the `MarkupTemplateEngine` which supports generating XML-like markup (XML, XHTML, HTML5, etc), but it can be used to generate any text based content.

It is compiled statically to be very fast and supports internationalization. It also supports templates as input.

```
package mypackage

import groovy.text.markup.MarkupTemplateEngine
import groovy.text.markup.TemplateConfiguration

String xml_template = '''xmlDeclaration()
tasks {
  tasks.each {
    task (summary: it.summary, duration: it.duration)
  }
}'''
String html_template ='''
yieldUnescaped '<!DOCTYPE html>'
html(lang:'en') {
  head {
    meta('http-equiv':'"Content-Type" content="text/html; charset=utf-8"')
    title('My page')
  }
  body {
    p('This is an example of HTML contents')
  }
}'''

values = [tasks:[
  new Task(summary:"Doit1", duration:4),
  new Task(summary:"Doit2", duration:12)
  ]]
TemplateConfiguration config = new TemplateConfiguration()
def engine = new MarkupTemplateEngine(config)
def template1 = engine.createTemplate(xml_template)
def template2 = engine.createTemplate(html_template)
println template1.make(values)
println template2.make(values)
```

Templates support includes.

19.3. Creating Markup (XML) files with the MarkupBuilder

The usage of the `MarkupBuilder` as "old" builder is demonstrated by the following snippet.

```
package com.vogella.groovy.builder.markup

import groovy.xml.MarkupBuilder

class TestMarkupWriter {
  static main (args) {
    def date = new Date()
    StringWriter writer = new StringWriter()
    MarkupBuilder builder = new MarkupBuilder(writer)
    builder.tasks {
      for (i in 1..10) {
        task {
          summary (value: "Test $i")
          description (value: "Description $i")
          dueDate(value: "${date.format('MM/dd/yy')}")
        }
      }
    }
    print writer.toString()
  }
}
```

The builder in Groovy uses the method names to construct the node and node names. These methods are not defined on the MarkupBuilder class but constructed at runtime.

It is possible to use maps MarkupBuilderin the builder

```
package com.vogella.groovy.builder.markup

import groovy.xml.MarkupBuilder



class TestMarkupWriterMap {
  static main (args) {
    Map map = [Jim:"Knopf", Thomas:"Edison"]
    def date = new Date()
    StringWriter writer = new StringWriter()
    MarkupBuilder builder = new MarkupBuilder(writer)
    builder.tasks {
      map.each { key, myvalue ->
        person {
        firstname (value : "$key")
        lastname(value : "$myvalue")
        }
      }
    }
    print writer.toString()
  }
}
```

You can also use the builder to create valid HTML.

```
package com.vogella.groovy.builder.markup

import groovy.xml.MarkupBuilder



class TestMarkupHtml {
  static main (args) {
    Map map = [Jim:"Knopf", Thomas:"Edison"]
    def date = new Date()
    StringWriter writer = new StringWriter()
    MarkupBuilder builder = new MarkupBuilder(writer)
    builder.html {
      head { title "vogella.com" }
      body {
        dev (class:"strike") {
          p "This is a line"
        }
      }
      print writer.toString()
    }
  }
}
```

20. Groovy and JSON

Similar to the `XmlSlurper` class for parsing XML, Groovy provides the `JsonSlurper` for parsing JSON.

```
import groovy.json.JsonOutput
import groovy.json.JsonSlurper

def a = new JsonSlurper().parse(new File("./input/tasks.json"))
JsonOutput.prettyPrint(a.toString())
```

You can use the `setType(LAX)` method to parse partially invalid JSON files. With this mode the JSON file can contain // comments, Strings can use " for quotes can be forgotten.

21. Compile-time meta programming and AST transformations

21.1. What are AST transformations?

An *Abstract Syntax Tree (AST)* is a in memory representation of code as data. An ADT transformation allows to modify this representation during compile time. This is sometimes called compile-time metaprogramming.

Groovy provides several AST transformations which allows you to reduce the amount of code you have to write.

21.2. @TupleConstructor

If a class is annotated with `@TupleConstructor` Groovy generates a constructor using all fields.

21.3. @EqualsAndHashCode

The `@EqualsAndHashCode` annotation can be applied to a class, creates the `equals` and `hashCode` method. Includes fields can be customized.

```
package asttransformations

import groovy.transform.EqualsAndHashCode


@EqualsAndHashCode (excludes=["summary","description"])
public class Task {
  private final long id;
  private String summary;
  private String description;
}
```

21.4. @ToString for beans

The `@ToString` annotation can be applied to a class, generates a toString method, support boolean flags, like includePackage, includeNames, allows to exclude a list of fields.

This annotation typically only considers properties (non-private fields) but you can include them via `@ToString(includeFields=true)`. Via `@ToString(excludes=list)` we can exclude a list of fields and properties.

```
package asttransformations

import groovy.transform.ToString


@ToString(includeFields=true)
public class Task {
  private final long id;
  private String summary;
  private String description;
}
```

21.5. @Canonical

Combines @ToString, @EqualsAndHashCode and @TupleConstructor.

21.6. @Sortable for beans

You can automatically created a `Comparator` for a Groovy bean by annotating it with `@Sortable`. Fields are used in the order of declaration.

You can also include/exclude fields.

```
package asttransformations

import groovy.transform.Sortable

@Sortable(excludes = ['duration'])
class Task {
  String summary
  String description
  int duration
}
```

21.7. @Memoize for methods

If the `@Memoize` annotation is to a method the Groovy runtime caches the result for invocations with the same parameters. If the annotated method is called the first time with a certain set of parameter, it is executed and the result is cached. If the method is called again with the same parameters, the result is returned from the cache.

```
package asttransformations

import groovy.transform.Memoized

class MemoizedExample {
  @Memoized
  int complexCalculation (int input){
    println "called"
    // image something really time consuming here
    return input + 1;
  }
}
package asttransformations

def m = new MemoizedExample()

// prints "called"
m.complexCalculation(1)

// no output as value is returned from cache
m.complexCalculation(1)
```

21.8. @AnnotationCollector for combining AST Transformations annotations

The `AnnotationCollector` allows to combine other AST Transformations annotations.

```
package asttransformations;
```

```
import groovy.transform.AnnotationCollector
import groovy.transform.EqualsAndHashCode
import groovy.transform.ToString

@ToString(includeNames=true)
@EqualsAndHashCode
@AnnotationCollector
public @interface Pojo {}
```

You can use this annotation, it is also possible to override parameters in them.

```
package asttransformations


@Pojo
class Person {
  String firstName
  String lastName
}

@Pojo(includeNames=false)
class Person2 {
  String firstName
  String lastName
}
package asttransformations


def p = new Person(firstName:"Lars" ,lastName:"Vogel")
println p
// output: asttransformations.Person(firstName:Lars, lastName:Vogel)

p = new Person2(firstName:"Lars" ,lastName:"Vogel")
println p
// output: asttransformations.Person2(Lars, Vogel)
```

21.9. @Bindable observable Properties

The Java beans specification requires that Java beans support `PropertyChangeSupport` for all fields.

The `@groovy.beans.Bindable` annotation can be applied to a whole class or a method. If the property is applied to a class, all methods will be treated as having the `@Bindable` annotation. This will trigger Groovy to generated a `java.beans.PropertyChangeSupport` property in the class and generate methods so that listeners can register and deregister. Also all setter methods will notfiy the property change listener.

The following listing shows the Java version of a Java Bean with one property.

21.10. @Builder

The `@Builder` can be applied to a class and generates transparently a builder for this class.

```
package asttransformations

import groovy.transform.ToString
import groovy.transform.builder.Builder

@Builder
@ToString(includeNames=true)
class TaskWithBuilder {
  String summary
  String description
  int duration
}
package asttransformations

TaskWithBuilder test = TaskWithBuilder.builder().
            summary("Help").
            description("testing").
            duration(5).
            build();

print test;
```

21.11. @Grab for dependency management

Groovy allows to add Maven dependencies to your Groovy script or Groovy class using the `@Grab` annotation. Before a Groovy program is executed it reads the `@Grab` annotation, resolves the Maven dependencies, downloads them and adds them to the classpath of the program.

```
@Grab(group='org.eclipse.jetty.aggregate', module='jetty-all',
version='7.6.15.v20140411')
```

Using the @GrabResolver annotation you can specify the Maven repository you want to use. For example `@GrabResolver(name='myrepo', root='http://myrepo.my-company.com/') .`

21.12. Other AST Transformations

The following table lists other commonly used annotations in Groovy code:

Table 5. AST transformation annotations

Annotation	Description
@Singleton	Makes annotated class a Singleton, access via ClassName.instance.
@Immutable	Makes all fields as final and do not generate setters.
@PackageScope	Defines fields, methods or class as package scope, which is the default access modifier in Java.

21.13. Writing custom transformations

You can also define you custom local or global transformations. For a local transformation you would write your own annotation and write a processors for that and use the annotation on an element in your Groovy class. The Groovy compiler calls your processors to transform the input into something else.

Global transformations are applied to every single source unit in the compilation without the need for additional customization.

22. Meta Object Protocol

22.1. Meta Object Protocol

The *Meta-Object Protocol (MOP)* allows to add at runtime dynamically methods and properties.

22.2. Calling methods or accessing properties on a Groovy object

If a method is called or a property is accessed in a class and this class does not define this method / property then pre-defined methods are called which can be used to handle this call.

- def methodMissing (String name, args) - Called for missing method
- void setProperty (String property, Object o) - called for non existing setter of a property
- Object getProperty (String property) - called for non existing getter of a property

Instances of Groovy object have default implementations of these methods, but an Groovy object can override these methods. The Groovy framework calls the methods at runtime if a method or property cannot be found. This approach is for example used by the Groovy builder pattern, it pretends to have certain method.

23. Exercise: Meta Object Protocol

23.1. Target

In this exercise you learn how to extend a Groovy class using the Meta Object Protocol.

23.2. Making a Groovy object responding to all methods and property calls

Create the following Groovy class. This class returns a fixed value for every property asked and it fakes method calls.

```
package mop


public class AnyMethodExecutor{
  // Should get ignored
```

```
  String value ="Lars"

  // always return 5 no matter which property is called
  Object getProperty (String property){
    return 5;
  }

  void setProperty (String property, Object o){
    // ignore setting
  }

  def methodMissing (String name, args){
    def s = name.toLowerCase();
    if (!s.contains("hello")) {
      return "This method is just fake"
    } else {
      return "Still a fake method but 'hello' back to you."

    }
  }

}
```

Test this method via the following Groovy script.

```
package mop

def test = new AnyMethodExecutor ();

// you can call any method you like
// on this class
assert "This method is just fake" == test.hall();
assert "This method is just fake" == test.Hallo();
assert "Still a fake method but 'hello' back to you." == test.helloMethod();

// setting is basically ignored
test.test= 5;
test.superDuperCool= 100

// all properties return 5
assert test.superDuperCool == 5
assert test.value == 5;
```

23.3. Exercise: Adding JSON output to Groovy class, the ugly and the smart way

Create the following Groovy class.

```
package mop;

import groovy.json.JsonBuilder
import groovy.json.JsonOutput

public class Task {
  String summary
  String description
```

```
  def methodMissing (String name, args){
    if (name=="toJson") {
      JsonBuilder b1 = new JsonBuilder(this)
      return JsonOutput.prettyPrint(b1.toString())
    }
  }
}
```

Is uses the `methodMissing` to respond to a `toJson` method call. This implementation is a bit ugly as it "pollutes" our domain model with "framework" code.

This script trigger the JSON generation.

```
package mop

def t = new Task(summary: "Mop",description:"Learn all about Mop");
println t.toJson()
```

Groovy allows to created an instance of `MetaClass` and register it automatic for a certain class. This registration is based on a package naming conversion:

```
// define an instance of Metaclass in such a package
// Groovy will register it a MetaClass
groovy.runtime.metaclass.[thePackage].[theClassName]MetaClass
```

Create the following class in the listed package to register it as `MetaClass` for your class.

```
package groovy.runtime.metaclass.mop;

import groovy.json.JsonBuilder
import groovy.json.JsonOutput

class TaskMetaClass extends DelegatingMetaClass {

  TaskMetaClass(MetaClass meta) {
    super(meta)
  }

  @Override
  def invokeMethod(Object object, String method, Object[] args) {
    println method
    if (method == "toJson") {
      JsonBuilder b1 = new JsonBuilder(object)
      return JsonOutput.prettyPrint(b1.toString())
    }
    super.invokeMethod(object, method, args)
  }
}
```

This allows you to clean up your domain model.

```
package mop;
```

```
import groovy.json.JsonBuilder
import groovy.json.JsonOutput

public class Task {
  String summary
  String description
}
```

Run your small test script again and validate that the conversion to JSON still works.

```
package mop

def t = new Task(summary: "Mop",description:"Learn all about Mop");
println t.toJson()
```

24. Using Groovy classes in Java

24.1. Calling Groovy classes directly

To use Groovy classes in Java classes you need to add the Groovy runtime to the Java classpath.

Create a new Java project "de.vogella.groovy.java". Create package "de.vogella.groovy.java"

Create the following Groovy class.

```
package de.vogella.groovy.java


public class Person{
  String firstName
  String lastName
  int age
  def address
}
```

Create the following Java class.

```
package de.vogella.groovy.java;

public class Main {
  public static void main(String[] args) {
    Person p = new Person();
    p.setFirstName("Lars");
    p.setLastName("Vogel");
    System.out.println(p.getFirstName() + " " + p.getLastName());
  }
}
```

You should be able to run this Java program.

Right-click your project, select "Properties" and check that the build path includes the Groovy libraries.

24.2. Calling a script

```
import java.io.FileNotFoundException;
import java.io.FileReader;

import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class ExecuteGroovyViaJSR223 {
  public static void main(String[] args) {
    ScriptEngine engine = new ScriptEngineManager()
        .getEngineByName("groovy");
    try {
      engine.put("street", "Haindaalwisch 17a");
      engine.eval("println 'Hello, Groovy!'");
      engine.eval(new FileReader("src/hello.groovy"));
    } catch (ScriptException e) {
      e.printStackTrace();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    }
  }
}
println "hello"
// if not defined it becomes part of the binding
println street
```