Contents

[Groovy Overview 6](#_Toc102745112)

[Features of Groovy 6](#_Toc102745113)

[Groovy Installation 7](#_Toc102745114)

[Running a Groovy Script 7](#_Toc102745115)

[Using the Groovy Shell 7](#_Toc102745116)

[Recognized Commands 8](#_Toc102745117)

[Using the Groovy Console 11](#_Toc102745118)

[Running a Script 11](#_Toc102745119)

[Interruption 12](#_Toc102745120)

[Running Groovy Scripts in Visual Studio Code 12](#_Toc102745121)

[Groovy - Basic Syntax 14](#_Toc102745122)

[Import Statement in Groovy 14](#_Toc102745123)

[Tokens in Groovy 14](#_Toc102745124)

[Comments in Groovy 15](#_Toc102745125)

[Semicolons 15](#_Toc102745126)

[Identifiers 15](#_Toc102745127)

[Keywords 16](#_Toc102745128)

[Whitespaces 16](#_Toc102745129)

[Literals 17](#_Toc102745130)

[Groovy – Data Types 18](#_Toc102745131)

[Built-in Data Types 18](#_Toc102745132)

[Bound values 18](#_Toc102745133)

[Class Numeric 18](#_Toc102745134)

[Groovy – Variables 20](#_Toc102745135)

[Variable Declarations 20](#_Toc102745136)

[Naming Variables 20](#_Toc102745137)

[Printing Variables 21](#_Toc102745138)

[Groovy – Operators 22](#_Toc102745139)

[Arithmetic Operators 22](#_Toc102745140)

[Relational operators 23](#_Toc102745141)

[Logical Operators 25](#_Toc102745142)

[Bitwise Operators 25](#_Toc102745143)

[Assignment operators 27](#_Toc102745144)

[Range Operators 28](#_Toc102745145)

[Operator Precedence 28](#_Toc102745146)

[Groovy – Loops 30](#_Toc102745147)

[The while Statement 30](#_Toc102745148)

[The for Statement 31](#_Toc102745149)

[The for-in Statement 32](#_Toc102745150)

[Loop Control Statements 34](#_Toc102745151)

[The break Statement 34](#_Toc102745152)

[The continue Statement 35](#_Toc102745153)

[Groovy – Decision Making 37](#_Toc102745154)

[The if Statement 37](#_Toc102745155)

[The if-else Statement 38](#_Toc102745156)

[The Nested if Statement 39](#_Toc102745157)

[The switch Statement 40](#_Toc102745158)

[The Nested switch Statement 42](#_Toc102745159)

[Groovy – Methods 44](#_Toc102745160)

[Method Parameters 44](#_Toc102745161)

[Default Parameters 45](#_Toc102745162)

[Method Return Values 46](#_Toc102745163)

[Instance methods 46](#_Toc102745164)

[Local and External Parameter Names 47](#_Toc102745165)

[this method for Properties 47](#_Toc102745166)

[Groovy – File I/O 49](#_Toc102745167)

[Reading files 49](#_Toc102745168)

[Reading the Contents of a File as an Entire String 49](#_Toc102745169)

[Writing to Files 50](#_Toc102745170)

[Getting the Size of a File 50](#_Toc102745171)

[Testing if a File is a Directory 50](#_Toc102745172)

[Creating a Directory 50](#_Toc102745173)

[Deleting a File 51](#_Toc102745174)

[Copying files 51](#_Toc102745175)

[Getting Directory Contents 51](#_Toc102745176)

[Groovy – Optionals 53](#_Toc102745177)

[Groovy – Numbers 55](#_Toc102745178)

[Some Number Methods 56](#_Toc102745179)

[xxxValue() 56](#_Toc102745180)

[compareTo() 57](#_Toc102745181)

[equals() 58](#_Toc102745182)

[toString() 58](#_Toc102745183)

[parseInt() 59](#_Toc102745184)

[random() 60](#_Toc102745185)

[Other Examples of using the Random class 60](#_Toc102745186)

[Other Number Methods 61](#_Toc102745187)

[Groovy – Strings 62](#_Toc102745188)

[String Indexing 62](#_Toc102745189)

[Basic String Operations 63](#_Toc102745190)

[Concatenation of Two Strings 63](#_Toc102745191)

[String Repetition 63](#_Toc102745192)

[String Length 64](#_Toc102745193)

[Other String Methods 64](#_Toc102745194)

[Groovy – Ranges 67](#_Toc102745195)

[contains() 67](#_Toc102745196)

[get() 67](#_Toc102745197)

[getFrom() 68](#_Toc102745198)

[getTo() 68](#_Toc102745199)

[isReverse() 69](#_Toc102745200)

[size() 69](#_Toc102745201)

[subList() 70](#_Toc102745202)

[Groovy – Lists 71](#_Toc102745203)

[Iterate a List 71](#_Toc102745204)

[Iterate with index# 71](#_Toc102745205)

[add() 71](#_Toc102745206)

[contains() 72](#_Toc102745207)

[get() 73](#_Toc102745208)

[isEmpty() 73](#_Toc102745209)

[minus() 74](#_Toc102745210)

[plus() 75](#_Toc102745211)

[pop() 75](#_Toc102745212)

[remove() 76](#_Toc102745213)

[reverse() 76](#_Toc102745214)

[size() 77](#_Toc102745215)

[sort() 77](#_Toc102745216)

[Groovy – Maps 79](#_Toc102745217)

[Iterate a Map 79](#_Toc102745218)

[containsKey() 79](#_Toc102745219)

[get() 80](#_Toc102745220)

[keyset() 80](#_Toc102745221)

[values() 81](#_Toc102745222)

[put() 81](#_Toc102745223)

[size() 82](#_Toc102745224)

[Groovy – Dates and Times 83](#_Toc102745225)

[Date() 83](#_Toc102745226)

[Date (long millisec) 83](#_Toc102745227)

[Exception Handling 87](#_Toc102745228)

[Catching Exceptions 88](#_Toc102745229)

[Multiple Catch Blocks 89](#_Toc102745230)

[Finally Block 89](#_Toc102745231)

[public String getMessage() 90](#_Toc102745232)

[public Throwable getCause() 90](#_Toc102745233)

[public String toString() 90](#_Toc102745234)

[public void printStackTrace() 90](#_Toc102745235)

[public StackTraceElement [] getStackTrace() 91](#_Toc102745236)

[public Throwable fillInStackTrace() 91](#_Toc102745237)

[Groovy – Object Oriented 92](#_Toc102745238)

[getter and setter Methods 92](#_Toc102745239)

[Instance Methods 93](#_Toc102745240)

[Creating Multiple Objects 94](#_Toc102745241)

[Inheritance 95](#_Toc102745242)

[Extends 95](#_Toc102745243)

[Inner Classes 96](#_Toc102745244)

[Abstract Classes 97](#_Toc102745245)

[Interfaces 98](#_Toc102745246)

[Groovy - Meta Object Programming 99](#_Toc102745247)

[Missing Properties 99](#_Toc102745248)

[Another Better Example for getProperty() and setProperty() 100](#_Toc102745249)

[Missing methods 101](#_Toc102745250)

[Method Missing 102](#_Toc102745251)

[Metaclass 103](#_Toc102745252)

[Groovy – Closures 104](#_Toc102745253)

[Formal parameters in closures 104](#_Toc102745254)

[Closures and Variables 105](#_Toc102745255)

[Using Closures in Methods 105](#_Toc102745256)

[Closures in Collections and String 106](#_Toc102745257)

[Using Closures with Lists 106](#_Toc102745258)

[Using Closures with Maps 106](#_Toc102745259)

[Methods used with Closures 107](#_Toc102745260)

[Groovy – Builder Pattern 110](#_Toc102745261)

[Domain Model 110](#_Toc102745262)

[Company 110](#_Toc102745263)

[Department 111](#_Toc102745264)

[Employee 112](#_Toc102745265)

[CompanyBuilder Implementation 112](#_Toc102745266)

[CompanyBuilder Usage / Syntax 114](#_Toc102745267)

[References 116](#_Toc102745268)

# Groovy Overview

Groovy is an object-oriented language which is based on Java platform. Groovy 1.0 was released in January 2, 2007 with Groovy 2.4 as the current major release. Groovy is distributed via the Apache License v 2.0.

## Features of Groovy

Groovy has the following features:

* Support for both static and dynamic typing.
* Support for operator overloading.
* Native syntax for lists and associative arrays.
* Native support for regular expressions.
* Native support for various markup languages such as XML and HTML.
* Groovy is simple for Java developers since the syntax for Java and Groovy are very similar.
* You can use existing Java libraries.
* Groovy extends the java.lang.Object.

The official website for Groovy is <http://www.groovy-lang.org/>

# Groovy Installation

There are a variety of ways to get the Groovy environment setup.

**Binary download and installation** − Go to the link [www.groovy-lang.org/download.html](http://www.groovy-lang.org/download.html) to get the Windows Installer section. Select the latest stable distribution of Groovy and click on the “Windows Installer” option to start the download of the Groovy installer.

Current distribution version is 4.0.2.

After the installer is downloaded, run, and complete the installer.

**Note**: Make sure you select the options to create the GROOVY\_HOME env variable and add the path to the Groovy home’s bin folder to the PATH env variable.

## Running a Groovy Script

After the installation is done, check the version from the command prompt by running:

groovy -v

groovy --version

## Using the Groovy Shell

From the command prompt, run the following command to open the Groovy shell:

groovysh

Then from the shell, execute and check the following statements:

class Foo {

def bar() {

println "baz"

}

}

foo = new Foo()

foo.bar()

Shell variables are **all** untyped (i.e. no def or other type information).

This **will** set a shell variable:

foo = "bar"

println(foo)

But, this will evaluate a local variable and will **not** be saved to the shell’s environment:

def foo = "bar"

println(foo)

Unknown property: bar

Functions can be defined in the shell and will be saved for later use.

Defining a function is easy:

groovy:000> def hello(name) {

groovy:001> println("Hello $name")

groovy:002> }

And then using it is as one might expect:

hello("Jason")

Internally the shell creates a closure to encapsulate the function and then binds the closure to a variable. So, variables and functions share the same namespace.

### Recognized Commands

#### help

Display the list of commands (and aliases) or the help text for specific command.

groovy:000> :help

Help for a Command

While in the interactive shell, you can ask for help for any command to get more details about its syntax or function. Here is an example of what happens when you ask for help for the help command:

groovy:000> :help :help

#### [exit](https://groovy-lang.org/groovysh.html#GroovyShell-exit)

Exit the shell.

This is the **only** way to exit the shell. Well, you can still CTRL-C, but the shell will complain about an abnormal shutdown of the JVM.

#### [import](https://groovy-lang.org/groovysh.html#GroovyShell-import)

Add a custom import which will be included for all shell evaluations.

This command can be given at any time to add new imports.

#### [grab](https://groovy-lang.org/groovysh.html#GroovyShell-grab)

Grab a dependency (Maven, Ivy, etc.) from Internet sources or cache, and add it to the Groovy Shell environment.

groovy:000> :grab 'com.google.guava:guava:19.0'

groovy:000> import com.google.common.collect.BiMap

===\> com.google.common.collect.BiMap

This command can be given at any time to add new dependencies.

#### [display](https://groovy-lang.org/groovysh.html#GroovyShell-display)

Display the contents of the current buffer.

This only displays the buffer of an incomplete expression. Once the expression is complete, the buffer is rest. The prompt will update to show the size of the current buffer as well.

**Example**

groovy:000> class Foo {

groovy:001> def bar

groovy:002> def baz() {

groovy:003> :display

001> class Foo {

002> def bar

003> def baz() {

#### [clear](https://groovy-lang.org/groovysh.html#GroovyShell-clear)

Clears the current buffer, resetting the prompt counter to 000. Can be used to recover from compilation errors.

#### [show](https://groovy-lang.org/groovysh.html#GroovyShell-show)

Show variables, classes or preferences or imports.

#### show variables

groovy:000> :show variables

Variables:

\_ = true

show classes

show imports

show preferences

show all

#### [inspect](https://groovy-lang.org/groovysh.html#GroovyShell-inspect)

Opens the GUI object browser to inspect a variable or the result of the last evaluation.

#### [purge](https://groovy-lang.org/groovysh.html#GroovyShell-purge)

Purges objects from the shell.

purge variables

purge classes

purge imports

purge preferences

purge all

#### [edit](https://groovy-lang.org/groovysh.html#GroovyShell-edit)

Edit the current buffer in an external editor.

Currently only works on UNIX systems which have the EDITOR environment variable set, or have configured the editor preference.

#### [load](https://groovy-lang.org/groovysh.html#GroovyShell-load)

Load one or more files (or urls) into the buffer.

Create a file named test.groovy with these lines:

class Test {

def show() {

println("Hi there")

}

}

In the shell, run these commands:

:load test.groovy

tst = new Test()

tst.show()

#### [save](https://groovy-lang.org/groovysh.html#GroovyShell-save)

Saves the buffer’s contents to a file.

#### [record](https://groovy-lang.org/groovysh.html#GroovyShell-record)

Record the current session to a file.

record start

record stop

record status

#### [history](https://groovy-lang.org/groovysh.html#GroovyShell-history)

Display, manage and recall edit-line history.

history show

history recall

history flush

history clear

#### [alias](https://groovy-lang.org/groovysh.html#GroovyShell-alias)

Create an alias.

#### [doc](https://groovy-lang.org/groovysh.html#GroovyShell-doc)

Opens a browser with documentation for the provided class. For example:

groovy:000> :doc java.util.List

http://docs.oracle.com/javase/7/docs/api/java/util/List.html

http://docs.groovy-lang.org/2.4.2-SNAPSHOT/html/groovy-jdk/java/util/List.html

Will open two windows (or tabs, depending on your browser):

* one for the JDK documentation
* one for the GDK documentation

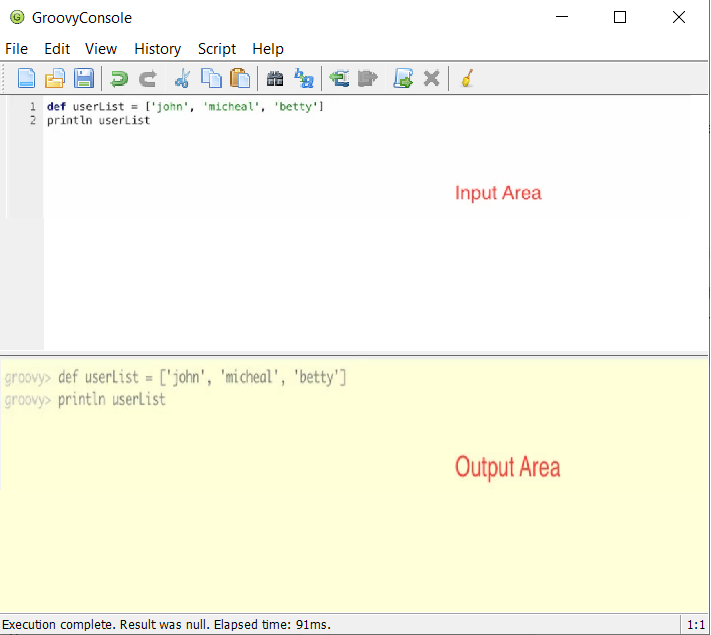
#### [set](https://groovy-lang.org/groovysh.html#GroovyShell-set)

Set or list preferences.

## Using the Groovy Console

From the command prompt, run groovyconsole.

It will open the following window:



In the top half of the windows, enter the following:

class Example {

static void main(String[] args) {

// Using a simple println statement to print output to the console

println('Hello World');

}

}

### Running a Script

To run the Groovy script, do either of the following:

1. From the menu, select Script -> Run.
2. Press CTRL+R.
3. Click on the Execute Groovy Script button  from the Ribbon.

There are only codes in the output area, because results are printed in the command line console where you started Groovy Console from. If you want to see results also in the output area, you need to enable **View ->** **Capture Standard Output**.

You can also open Groovy script file by using **File > Open** menu and the script will loaded in the input area. Then you can run loaded script.

Additionally, you can run only selected portion of the code in Input Area. If you select a section in the Input Area and click **Run Selection (or Ctrl + Shift + R)** in the **Script** menu, only the selected code will be executed.

### Interruption

You can interrupt current running script in Groovy Console after enabling interruption by selecting **Script > Allow Interruption** menu. What I mean by interruption here is interrupting current thread in running script. For example, to interrupt following script.

try {

while(1) {

println "something"

Thread.currentThread().sleep(500)

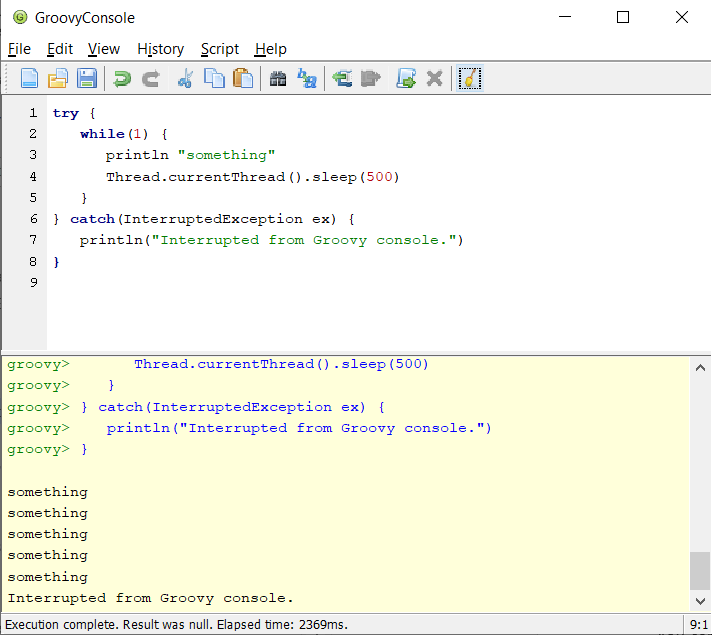
}

} catch(InterruptedException ex) {

println("Interrupted from Groovy console.")

}

You can first Run script and then you can click **Interrupt** button on the right side of the **Run** button. You will see an output like below.



## Running Groovy Scripts in Visual Studio Code

1. Install Visual Studio Code (code.visualstudio.com).
2. Install extension code-groovy.
3. Open folder that has groovy scripts.
4. In the Terminal window, run the script with groovy <script\_name>.
5. Install extension Code Runner.
6. Right-click the groovy script file (or even anywhere in the editor window) and select Run Code.

# Groovy - Basic Syntax

Sample program:

class Example {

static void main(String[] args) {

// Using a simple println statement to print output to the console

println('Hello World');

}

}

Save the above to a file named hello.groovy.

Open a command prompt terminal window and navigate to the folder where you created the file.

Run the following command to execute the program:

groovy hello.groovy

## Import Statement in Groovy

The import statement can be used to import the functionality of other libraries which can be used in your code. This is done by using the **import** keyword.

The following example shows how to use a simple import of the MarkupBuilder class which is probably one of the most used classes for creating HTML or XML markup.

import groovy.xml.MarkupBuilder

def xml = new MarkupBuilder()

By default, Groovy includes the following libraries in your code, so you don’t need to explicitly import them.

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

## Tokens in Groovy

A token is either a keyword, an identifier, a constant, a string literal, or a symbol.

println(“Hello World”);

In the above code line, there are two tokens, the first is the keyword println and the next is the string literal of “Hello World”.

## Comments in Groovy

Comments are used to document your code. Comments in Groovy can be single line or multiline.

Single line comments are identified by using the // at any position in the line. An example is shown below:

class Example {

static void main(String[] args) {

// Using a simple println statement to print output to the console

println('Hello World');

}

}

Multiline comments are identified with /\* in the beginning and \*/ to identify the end of the multiline comment.

class Example {

static void main(String[] args) {

/\* This program is the first program

This program shows how to display hello world \*/

println('Hello World');

}

}

## Semicolons

Unlike in the Java programming language, it is not mandatory to have semicolons after the end of every statement, It is optional.

class Example {

static void main(String[] args) {

def x = 5

println('Hello World');

}

}

If you execute the above program, both statements in the main method don't generate any error.

## Identifiers

Identifiers are used to define variables, functions or other user defined variables. Identifiers start with a letter, a dollar or an underscore. They cannot start with a number. Here are some examples of valid identifiers:

def employeename

def student1

def student\_name

where **def** is a keyword used in Groovy to define an identifier.

Here is a code example of how an identifier can be used in our Hello World program.

class Example {

static void main(String[] args) {

// One can see the use of a semi-colon after each statement

def x = 5;

println('Hello World');

}

}

In the above example, the variable **x** is used as an identifier.

## Keywords

Keywords as the name suggest are special words which are reserved in the Groovy Programming language. The following table lists the keywords which are defined in Groovy.

|  |  |  |  |
| --- | --- | --- | --- |
| as | assert | break | case |
| catch | class | const | continue |
| def | default | do | else |
| enum | extends | false | Finally |
| for | goto | if | implements |
| import | in | instanceof | interface |
| new | pull | package | return |
| super | switch | this | throw |
| throws | trait | true | try |
| while |  |  |  |

## Whitespaces

Whitespace is the term used in a programming language such as Java and Groovy to describe blanks, tabs, newline characters and comments. Whitespace separates one part of a statement from another and enables the compiler to identify where one element in a statement.

For example, in the following code example, there is a white space between the keyword **def** and the variable x. This is so that the compiler knows that **def** is the keyword which needs to be used and that x should be the variable name that needs to be defined.

def x = 5;

## Literals

A literal is a notation for representing a fixed value in groovy. The groovy language has notations for integers, floating-point numbers, characters, and strings. Here are some of the examples of literals in the Groovy programming language:

12

1.45

‘a’

“aa”

# Groovy – Data Types

In any programming language, you need to use various variables to store various types of information. Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory to store the value associated with the variable.

You may like to store information of various data types like string, character, wide character, integer, floating point, Boolean, etc. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

## Built-in Data Types

Groovy offers a wide variety of built-in data types. Following is a list of data types which are defined in Groovy −

* **byte** − This is used to represent a byte value. An example is 2.
* **short** − This is used to represent a short number. An example is 10.
* **int** − This is used to represent whole numbers. An example is 1234.
* **long** − This is used to represent a long number. An example is 10000090.
* **float** − This is used to represent 32-bit floating point numbers. An example is 12.34.
* **double** − This is used to represent 64-bit floating point numbers which are longer decimal number representations which may be required at times. An example is 12.3456565.
* **char** − This defines a single character literal. An example is ‘a’.
* **Boolean** − This represents a Boolean value which can either be true or false.
* **String** − These are text literals which are represented in **the form** of chain of characters. For example “Hello World”.

## Bound values

The following table shows the maximum allowed values for the numerical and decimal literals.

|  |  |
| --- | --- |
| byte | -128 to 127 |
| short | -32,768 to 32,767 |
| int | -2,147,483,648 to 2,147,483,647 |
| long | -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 |
| float | 1.40129846432481707e-45 to 3.40282346638528860e+38 |
| double | 4.94065645841246544e-324d to 1.79769313486231570e+308d |

## Class Numeric

Types In addition to the primitive types, the following object types (sometimes referred to as wrapper types) are allowed −

* java.lang.Byte
* java.lang.Short
* java.lang.Integer
* java.lang.Long
* java.lang.Float
* java.lang.Double

In addition, the following classes can be used for supporting arbitrary precision arithmetic −

|  |  |  |
| --- | --- | --- |
| **Name** | **Description** | **Example** |
| java.math.BigInteger | Immutable arbitrary-precision signed integral numbers | 30g |
| java.math.BigDecimal | Immutable arbitrary-precision signed decimal numbers | 3.5g |

The following code example showcases how the different built-in data types can be used −

class Example {

static void main(String[] args) {

//Example of a int datatype

int x = 5;

//Example of a long datatype

long y = 100L;

//Example of a floating point datatype

float a = 10.56f;

//Example of a double datatype

double b = 10.5e40;

//Example of a BigInteger datatype

BigInteger bi = 30g;

//Example of a BigDecimal datatype

BigDecimal bd = 3.5g;

println(x);

println(y);

println(a);

println(b);

println(bi);

println(bd);

}

}

When we run the above program, we will get the following result −

5

100

10.56

1.05E41

30

3.5

# Groovy – Variables

Variables in Groovy can be defined in two ways − using the **native syntax** for the data type or the next is **by using the def keyword**. For variable definitions it is mandatory to either provide a type name explicitly or to use "def" in replacement. This is required by the Groovy parser.

There are following basic types of variable in Groovy as explained in the previous chapter −

* **byte** − This is used to represent a byte value. An example is 2.
* **short** − This is used to represent a short number. An example is 10.
* **int** − This is used to represent whole numbers. An example is 1234.
* **long** − This is used to represent a long number. An example is 10000090.
* **float** − This is used to represent 32-bit floating point numbers. An example is 12.34.
* **double** − This is used to represent 64-bit floating point numbers which are longer decimal number representations which may be required at times. An example is 12.3456565.
* **char** − This defines a single character literal. An example is ‘a’.
* **Boolean** − This represents a Boolean value which can either be true or false.
* **String** − These are text literals which are represented in **the form** of chain of characters. For example “Hello World”.

Groovy also allows for additional types of variables such as arrays, structures and classes which we will see in the subsequent chapters.

## Variable Declarations

A variable declaration tells the compiler where and how much to create the storage for the variable.

Following is an example of variable declaration −

class Example {

static void main(String[] args) {

// x is defined as a variable

String x = "Hello";

// The value of the variable is printed to the console

println(x);

}

}

When we run the above program, we will get the following result −

Hello

## Naming Variables

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because Groovy, just like Java is a case-sensitive programming language.

class Example {

static void main(String[] args) {

// Defining a variable in lowercase

int x = 5;

// Defining a variable in uppercase

int X = 6;

// Defining a variable with the underscore in it's name

def \_Name = "Joe";

println(x);

println(X);

println(\_Name);

}

}

When we run the above program, we will get the following result −

5

6

Joe

We can see that **x** and **X** are two different variables because of case sensitivity and in the third case, we can see that \_Name begins with an underscore.

## Printing Variables

You can print the current value of a variable with the println function. The following example shows how this can be achieved.

class Example {

static void main(String[] args) {

//Initializing 2 variables

int x = 5;

int X = 6;

//Printing the value of the variables to the console

println("The value of x is " + x + "The value of X is " + X);

}

}

When we run the above program, we will get the following result −

The value of x is 5 The value of X is 6

# Groovy – Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations.

Groovy has the following types of operators −

* Arithmetic operators
* Relational operators
* Logical operators
* Bitwise operators
* Assignment operators

## Arithmetic Operators

The Groovy language supports the normal Arithmetic operators as any the language. Following are the Arithmetic operators available in Groovy −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Addition of two operands | 1 + 2 will give 3 |
| − | Subtracts second operand from the first | 2 − 1 will give 1 |
| \* | Multiplication of both operands | 2 \* 2 will give 4 |
| / | Division of numerator by denominator | 3 / 2 will give 1.5 |
| % | Modulus Operator and remainder of after an integer/float division | 3 % 2 will give 1 |
| ++ | Incremental operators used to increment the value of an operand by 1 | int x = 5;  x++;  x will give 6 |
| -- | Incremental operators used to decrement the value of an operand by 1 | int x = 5;  x--;  x will give 4 |

The following code snippet shows how the various operators can be used.

class Example {

static void main(String[] args) {

// Initializing 3 variables

def x = 5;

def y = 10;

def z = 8;

//Performing addition of 2 operands

println(x+y);

//Subtracts second operand from the first

println(x-y);

//Multiplication of both operands

println(x\*y);

//Division of numerator by denominator

println(z/x);

//Modulus Operator and remainder of after an integer/float division

println(z%x);

//Incremental operator

println(x++);

//Decrementing operator

println(x--);

}

}

When we run the above program, we will get the following result. It can be seen that the results are as expected from the description of the operators as shown above.

15

-5

50

1.6

3

5

6

## Relational operators

Relational operators allow of the comparison of objects. Following are the relational operators available in Groovy −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | Tests the equality between two objects | 2 == 2 will give true |
| != | Tests the difference between two objects | 3 != 2 will give true |
| < | Checks to see if the left objects is less than the right operand. | 2 < 3 will give true |
| <= | Checks to see if the left objects is less than or equal to the right operand. | 2 <= 3 will give true |
| > | Checks to see if the left objects is greater than the right operand. | 3 > 2 will give true |
| >= | Checks to see if the left objects is greater than or equal to the right operand. | 3 >= 2 will give true |

The following code snippet shows how the various operators can be used.

class Example {

static void main(String[] args) {

def x = 5;

def y = 10;

def z = 8;

if(x == y) {

println("x is equal to y");

} else

println("x is not equal to y");

if(z != y) {

println("z is not equal to y");

} else

println("z is equal to y");

if(z != y) {

println("z is not equal to y");

} else

println("z is equal to y");

if(z<y) {

println("z is less than y");

} else

println("z is greater than y");

if(x<=y) {

println("x is less than y");

} else

println("x is greater than y");

if(x>y) {

println("x is greater than y");

} else

println("x is less than y");

if(x>=y) {

println("x is greater or equal to y");

} else

println("x is less than y");

}

}

When we run the above program, we will get the following result. It can be seen that the results are as expected from the description of the operators as shown above.

x is not equal to y

z is not equal to y

z is not equal to y

z is less than y

x is less than y

x is less than y

x is less than y

## Logical Operators

Logical operators are used to evaluate Boolean expressions. Following are the logical operators available in Groovy −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| && | This is the logical “and” operator | true && true will give true |
| || | This is the logical “or” operator | true || true will give true |
| ! | This is the logical “not” operator | !false will give true |

The following code snippet shows how the various operators can be used.

class Example {

static void main(String[] args) {

boolean x = true;

boolean y = false;

boolean z = true;

println(x&&y);

println(x&&z);

println(x||z);

println(x||y);

println(!x);

}

}

When we run the above program, we will get the following result. It can be seen that the results are as expected from the description of the operators as shown above.

false

true

true

true

false

## Bitwise Operators

Groovy provides four bitwise operators. Following are the bitwise operators available in Groovy −

|  |  |
| --- | --- |
| **Sr.No** | **Operator & Description** |
| 1 | & This is the bitwise “and” operator |
| 2 | | This is the bitwise “or” operator |
| 3 | ^ This is the bitwise “xor” or Exclusive or operator |
| 4 | ~ This is the bitwise negation operator |

Here is the truth table showcasing these operators.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **q** | **p & q** | **p | q** | **p ^ q** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

The following code snippet shows how the various operators can be used.

class Example {

static void main(String[] args) {

int a = 00111100;

int b = 00001101;

int x;

println(Integer.toBinaryString(a&b));

println(Integer.toBinaryString(a|b));

println(Integer.toBinaryString(a^b));

a=~a;

println(Integer.toBinaryString(a));

}

}

When we run the above program, we will get the following result. It can be seen that the results are as expected from the description of the operators as shown above.

1001000000

1001001001000001

1001000000000001

100100100100000

## Assignment operators

The Groovy language also provides assignment operators. Following are the assignment operators available in Groovy −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| += | This adds right operand to the left operand and assigns the result to left operand. | def A = 5  A+=3  Output will be 8 |
| -= | This subtracts right operand from the left operand and assigns the result to left operand | def A = 5  A-=3  Output will be 2 |
| \*= | This multiplies right operand with the left operand and assigns the result to left operand | def A = 5  A\*=3  Output will be 15 |
| /= | This divides left operand with the right operand and assigns the result to left operand | def A = 6  A/=3  Output will be 2 |
| %= | This takes modulus using two operands and assigns the result to left operand | def A = 5  A%=3  Output will be 2 |

class Example {

static void main(String[] args) {

int x = 5;

println(x+=3);

println(x-=3);

println(x\*=3);

println(x/=3);

println(x%=3);

}

}

When we run the above program, we will get the following result. It can be seen that the results are as expected from the description of the operators as shown above.

8

5

15

5

2

## Range Operators

Groovy supports the concept of ranges and provides a notation of range operators with the help of the .. notation. A simple example of the range operator is given below.

def range = 0..5

This just defines a simple range of integers, stored into a local variable called range with a lower bound of 0 and an upper bound of 5.

The following code snippet shows how the various operators can be used.

class Example {

static void main(String[] args) {

def range = 5..10;

println(range);

println(range.get(2));

}

}

When we run the above program, we will get the following result −

From the **println** statement, you can see that the entire range of numbers which are defined in the range statement are displayed.

The get statement is used to get an object from the range defined which takes in an index value as the parameter.

[5, 6, 7, 8, 9, 10]

7

## Operator Precedence

The following table lists all groovy operators in order of precedence.

|  |  |
| --- | --- |
| **Sr.No** | **Operators & Names** |
| 1 | ++ -- + - pre increment/decrement, unary plus, unary minus |
| 2 | \* / % multiply, div, modulo |
| 3 | + - addition, subtraction |
| 4 | == != <=> equals, not equals, compare to |
| 5 | & binary/bitwise and |
| 6 | ^ binary/bitwise xor |
| 7 | | binary/bitwise or |
| 8 | && logical and |
| 9 | || logical or |
| 10 | = \*\*= \*= /= %= += -= <<= >>= >>>= &= ^= |= Various assignment operators |

# Groovy – Loops

So far, we have seen statements which have been executed one after the other in a sequential manner. Additionally, statements are provided in Groovy to alter the flow of control in a program’s logic. They are then classified into flow of control statements which we will see in detail.

|  |  |
| --- | --- |
| **S.No.** | **Statements & Description** |
| 1 | while Statement: The while statement is executed by first evaluating the condition expression (a Boolean value), and if the result is true, then the statements in the while loop are executed. |
| 2 | for Statement: The for statement is used to iterate through a set of values. |
| 3 | for-in Statement: The for-in statement is used to iterate through a set of values. |

## The while Statement

The syntax of the while statement is shown below −

while(condition) {

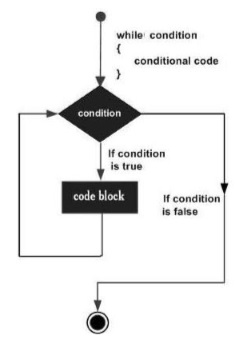
statement #1

statement #2

...

}

The **while** statement is executed by first evaluating the condition expression (a Boolean value), and if the result is true, then the statements in the while loop are executed. The process is repeated starting from the evaluation of the condition in the while statement. This loop continues until the condition evaluates to false. When the condition becomes false, the loop terminates. The program logic then continues with the statement immediately following the while statement. The following diagram shows the diagrammatic explanation of this loop.



Following is an example of a while loop statement −

class Example {

static void main(String[] args) {

int count = 0;

while(count<5) {

println(count);

count++;

}

}

}

In the above example, we are first initializing the value of a count integer variable to 0. Then our condition in the while loop is that we are evaluating the condition of the expression to be that count should be less than 5. Till the value of count is less than 5, we will print the value of count and then increment the value of count. The output of the above code would be −

0

1

2

3

4

## The for Statement

The **for** statement is used to iterate through a set of values. The **for** statement is generally used in the following way.

for(variable declaration;expression;Increment) {

statement #1

statement #2

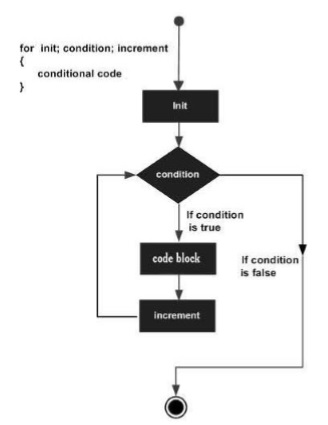
…

}

The classic for statement consists of the following parts −

* **Variable declaration** − This step is executed only once for the entire loop and used to declare any variables which will be used within the loop.
* **Expression** − This will consists of an expression which will be evaluated for each iteration of the loop.
* The increment section will contain the logic needed increment the variable declared in the **for** statement.

The following diagram shows the diagrammatic explanation of this loop.



Following is an example of the classic for statement −

class Example {

static void main(String[] args) {

for(int i = 0;i<5;i++) {

println(i);

}

}

}

In the above example, we are in our **for** loop doing three things −

* Declaring a variable **i** and Initializing the value of **i** to 0
* Putting a conditional expression that **the for** loop should execute till the value of i is less than 5.
* Increment the value of **i** by 1 for each iteration.

The output of the above code would be −

0

1

2

3

4

## The for-in Statement

The **for-in** statement is used to iterate through a set of values. The **for-in** statement is generally used in the following way.

for(variable in range) {

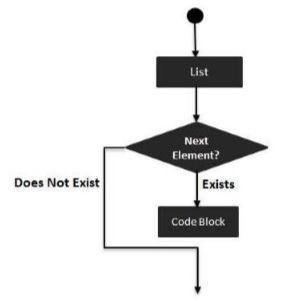
statement #1

statement #2

…

}

The following diagram shows the diagrammatic explanation of this loop.



Following is an example of a for-in statement −

class Example {

static void main(String[] args) {

int[] array = [0,1,2,3];

for(int i in array) {

println(i);

}

}

}

In the above example, we are first initializing an array of integers with 4 values of 0,1,2 and 3. We are then using our for loop statement to first define a variable i which then iterates through all of the integers in the array and prints the values accordingly. The output of the above code would be −

0

1

2

3

The **for-in** statement can also be used to loop through ranges. The following example shows how this can be accomplished.

class Example {

static void main(String[] args) {

for(int i in 1..5) {

println(i);

}

}

}

In the above example, we are actually looping through a range which is defined from 1 to 5 and printing the each value in the range. The output of the above code would be −

1

2

3

4

5

The **for-in** statement can also be used to loop through Map’s. The following example shows how this can be accomplished.

class Example {

static void main(String[] args) {

def employee = ["Ken" : 21, "John" : 25, "Sally" : 22];

for(emp in employee) {

println(emp);

}

}

}

In the above example, we are actually looping through a map which has a defined set of key value entries. The output of the above code would be −

Ken = 21

John = 25

Sally = 22

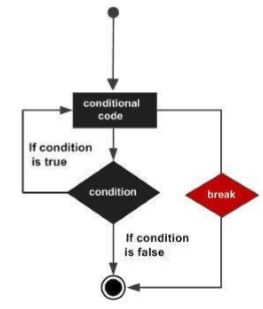
## Loop Control Statements

|  |  |
| --- | --- |
| **S.No.** | **Statements & Description** |
| 1 | Break Statement:  The break statement is used to alter the flow of control inside loops and switch statements. |
| 2 | Continue Statement  The continue statement complements the break statement. Its use is restricted to while and for loops. |

## The break Statement

The **break** statement is used to alter the flow of control inside loops and switch statements. We have already seen the break statement in action in conjunction with the switch statement. The break statement can also be used with while and for statements. Executing a **break** statement with any of these looping constructs causes immediate termination of the innermost enclosing loop.

The following diagram shows the diagrammatic explanation of the **break** statement.



Following is an example of the break statement −

class Example {

static void main(String[] args) {

int[] array = [0,1,2,3];

for(int i in array) {

println(i);

if(i == 2)

break;

}

}

}

The output of the above code would be −

0

1

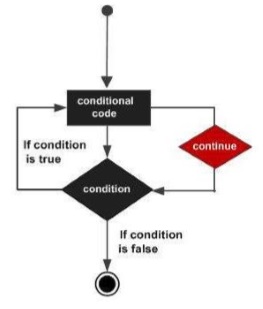
2

As expected, since there is a condition put saying that if the value of **i** is 2 then break from the loop that is why the last element of the array which is 3 is not printed.

## The continue Statement

The continue statement complements the break statement. Its use is restricted to while and for loops. When a continue statement is executed, control is immediately passed to the test condition of the nearest enclosing loop to determine whether the loop should continue. All subsequent statements in the body of the loop are ignored for that particular loop iteration.

The following diagram shows the diagrammatic explanation of the continue statement −



Following is an example of the **continue** statement −

class Example {

static void main(String[] args) {

int[] array = [0,1,2,3];

for(int i in array) {

if(i == 2)

continue;

println(i);

}

}

}

The output of the above code would be −

0

1

3

# Groovy – Decision Making

Decision-making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be **true**, and optionally, other statements to be executed if the condition is determined to be **false**.

|  |  |
| --- | --- |
| **Sr.No.** | **Statements & Description** |
| 1 | if Statement  The general working of this statement is that first a condition is evaluated in the if statement. If the condition is true, it then executes the statements. |
| 2 | if/else Statement  The general working of this statement is that first a condition is evaluated in the if statement. If the condition is true it then executes the statements thereafter and stops before the else condition and exits out of the loop. If the condition is false it then executes the statements in the else statement block and then exits the loop. |
| 3 | Nested If Statement  Sometimes there is a requirement to have multiple if statement embedded inside of each other. |
| 4 | Switch Statement  Sometimes the nested if-else statement is so common and is used so often that an easier statement was designed called the switch statement. |
| 5 | Nested Switch Statement  It is also possible to have a nested set of switch statements. |

## The if Statement

The first decision making statement is the **if** statement. The general form of this statement is −

if(condition) {

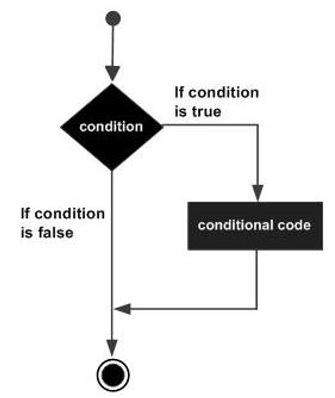
statement #1

statement #2

...

}

The general working of this statement is that first a condition is evaluated in the **if** statement. If the condition is true, it then executes the statements. The following diagram shows the flow of the **if** statement.



Following is an example of a if/else statement −

class Example {

static void main(String[] args) {

// Initializing a local variable

int a = 2

//Check for the boolean condition

if (a<100) {

//If the condition is true print the following statement

println("The value is less than 100");

}

}

}

In the above example, we are first initializing a variable to a value of 2. We are then evaluating the value of the variable and then deciding whether the **println** statement should be executed. The output of the above code would be −

The value is less than 100

## The if-else Statement

The next decision-making statement we will see is the **if/else** statement. The general form of this statement is −

if(condition) {

statement #1

statement #2

...

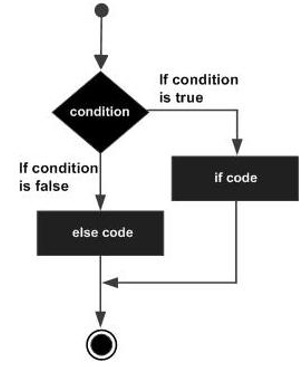
} else{

statement #3

statement #4

}

The general working of this statement is that first a condition is evaluated in the **if** statement. If the condition is true it then executes the statements thereafter and stops before the else condition and exits out of the loop. If the condition is false it then executes the statements in the else statement block and then exits the loop. The following diagram shows the flow of the **if** statement.



Following is an example of a if/else statement −

class Example {

static void main(String[] args) {

// Initializing a local variable

int a = 2

//Check for the boolean condition

if (a<100) {

//If the condition is true print the following statement

println("The value is less than 100");

} else {

//If the condition is false print the following statement

println("The value is greater than 100");

}

}

}

In the above example, we are first initializing a variable to a value of 2. We are then evaluating the value of the variable and then deciding on which **println** statement should be executed. The output of the above code would be

The value is less than 100.

## The Nested if Statement

Sometimes there is a requirement to have multiple if statement embedded inside of each other.

The general form of this statement is −

if(condition) {

statement #1

statement #2

...

} else if(condition) {

statement #3

statement #4

} else {

statement #5

statement #6

}

Following is an example of a nested if/else statement −

class Example {

static void main(String[] args) {

// Initializing a local variable

int a = 12

//Check for the boolean condition

if (a>100) {

//If the condition is true print the following statement

println("The value is less than 100");

} else

// Check if the value of a is greater than 5

if (a>5) {

//If the condition is true print the following statement

println("The value is greater than 5 and greater than 100");

} else {

//If the condition is false print the following statement

println("The value of a is less than 5");

}

}

}

In the above example, we are first initializing a variable to a value of 12. In the first **if** statement, we are seeing if the value of **a** is greater than 100. If not, then we enter our second for loop to see if the value of **a** is greater than 5 or less than 5. The output of the above code would be −

The value is greater than 5 and greater than 100

## The switch Statement

Sometimes the nested if-else statement is so common and is used so often that an easier statement was designed called the **switch** statement.

switch(expression) {

case expression #1:

statement #1

...

case expression #2:

statement #2

...

case expression #N:

statement #N

...

default:

statement #Default

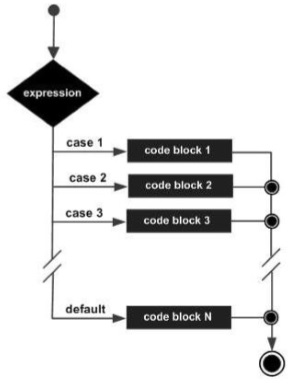
...

}

The general working of this statement is as follows −

* The expression to be evaluated is placed in the switch statement.
* There will be multiple case expressions defined to decide which set of statements should be executed based on the evaluation of the expression.
* A **break** statement is added to each case section of statements at the end. This is to ensure that the loop is exited as soon as the relevant set of statements gets executed.
* There is also a **default case** statement which gets executed if none of the prior case expressions evaluate to true.

The following diagram shows the flow of the **switch-case** statement.



Following is an example of the switch statement −

class Example {

static void main(String[] args) {

//initializing a local variable

int a = 2

//Evaluating the expression value

switch(a) {

//There is case statement defined for 4 cases

// Each case statement section has a break condition to exit the loop

case 1:

println("The value of a is One");

break;

case 2:

println("The value of a is Two");

break;

case 3:

println("The value of a is Three");

break;

case 4:

println("The value of a is Four");

break;

default:

println("The value is unknown");

break;

}

}

}

In the above example, we are first initializing a variable to a value of 2. We then have a switch statement which evaluates the value of the variable a. Based on the value of the variable it will execute the relevant case set of statements. The output of the above code would be −

The value of a is Two

## The Nested switch Statement

It is also possible to have a nested set of **switch** statements.

Following is an example of the nested switch statement −

class Example {

static void main(String[] args) {

//Initializing 2 variables i and j

int i = 0;

int j = 1;

// First evaluating the value of variable i

switch(i) {

case 0:

// Next evaluating the value of variable j

switch(j) {

case 0:

println("i is 0, j is 0");

break;

case 1:

println("i is 0, j is 1");

break;

// The default condition for the inner switch statement

default:

println("nested default case!!");

}

break;

// The default condition for the outer switch statement

default:

println("No matching case found!!");

}

}

}

In the above example, we are first initializing a variable to the a to a value of 2. We then have a **switch** statement which evaluates the value of the variable **a**. Based on the value of the variable it will execute the relevant case set of statements. The output of the above code would be −

i is 0, j is 1

# Groovy – Methods

A method is in Groovy is defined with a return type or with the **def** keyword. Methods can receive any number of arguments. It’s not necessary that the types are explicitly defined when defining the arguments. Modifiers such as public, private and protected can be added. By default, if no visibility modifier is provided, the method is public.

The simplest type of a method is one with no parameters as the one shown below −

def methodName() {

//Method code

}

Following is an example of simple method

class Example {

static def DisplayName() {

println("This is how methods work in groovy");

println("This is an example of a simple method");

}

static void main(String[] args) {

DisplayName();

}

}

In the above example, DisplayName is a simple method which consists of two println statements which are used to output some text to the console. In our static main method, we are just calling the DisplayName method. The output of the above method would be −

This is how methods work in groovy

This is an example of a simple method

## Method Parameters

A method is more generally useful if its behavior is determined by the value of one or more parameters. We can transfer values to the called method using method parameters. Note that the parameter names must differ from each other.

The simplest type of a method with parameters as the one shown below −

def methodName(parameter1, parameter2, parameter3) {

// Method code goes here

}

Following is an example of simple method with parameters

class Example {

static void sum(int a,int b) {

int c = a+b;

println(c);

}

static void main(String[] args) {

sum(10,5);

}

}

In this example, we are creating a sum method with 2 parameters, **a** and **b**. Both parameters are of type **int**. We are then calling the sum method from our main method and passing the values to the variables **a** and **b**.

The output of the above method would be the value 15.

## Default Parameters

There is also a provision in Groovy to specify default values for parameters within methods. If no values are passed to the method for the parameters, the default ones are used. If both nondefault and default parameters are used, then it has to be noted that the default parameters should be defined at the end of the parameter list.

Following is an example of simple method with parameters −

def someMethod(parameter1, parameter2 = 0, parameter3 = 0) {

// Method code goes here

}

Let’s look at the same example we looked at before for the addition of two numbers and create a method which has one default and another non-default parameter −

class Example {

static void sum(int a,int b = 5) {

int c = a+b;

println(c);

}

static void main(String[] args) {

sum(6);

}

}

In this example, we are creating a sum method with two parameters, **a** and **b**. Both parameters are of type int. The difference between this example and the previous example is that in this case we are specifying a default value for **b** as 5. So when we call the sum method from our main method, we have the option of just passing one value which is 6 and this will be assigned to the parameter **a** within the **sum** method.

The output of the above method would be the value 11.

class Example {

static void sum(int a,int b = 5) {

int c = a+b;

println(c);

}

static void main(String[] args) {

sum(6,6);

}

}

We can also call the sum method by passing 2 values, in our example above we are passing 2 values of 6. The second value of 6 will actually replace the default value which is assigned to the parameter **b**.

The output of the above method would be the value 12.

## Method Return Values

Methods can also return values back to the calling program. This is required in modern-day programming language wherein a method does some sort of computation and then returns the desired value to the calling method.

Following is an example of simple method with a return value.

class Example {

static int sum(int a,int b = 5) {

int c = a+b;

return c;

}

static void main(String[] args) {

println(sum(6));

}

}

In our above example, note that this time we are specifying a return type for our method sum which is of the type int. In the method we are using the return statement to send the sum value to the calling main program. Since the value of the method is now available to the main method, we are using the **println** function to display the value in the console.

The output of the above method would be the value 11.

## Instance methods

Methods are normally implemented inside classes within Groovy just like the Java language. A class is nothing but a blueprint or a template for creating different objects which defines its properties and behaviors. The class objects exhibit the properties and behaviors defined by its class. So the behaviors are defined by creating methods inside of the class.

We will see classes in more detail in a later chapter but following is an example of a method implementation in a class. In our previous examples we defined our method as static methods which meant that we could access those methods directly from the class. The next example of methods is instance methods wherein the methods are accessed by creating objects of the class. Again we will see classes in a later chapter, for now we will demonstrate how to use methods.

Following is an example of how methods can be implemented.

class Example {

int x;

public int getX() {

return x;

}

public void setX(int pX) {

x = pX;

}

static void main(String[] args) {

Example ex = new Example();

ex.setX(100);

println(ex.getX());

}

}

In our above example, note that this time we are specifying no static attribute for our class methods. In our main function we are actually creating an instance of the Example class and then invoking the method of the ‘ex’ object.

The output of the above method would be the value 100.

## Local and External Parameter Names

Groovy provides the facility just like java to have local and global parameters. In the following example, **lx** is a local parameter which has a scope only within the function of **getX()** and **x** is a global property which can be accessed inside the entire Example class. If we try to access the variable **lx** outside of the **getX()** function, we will get an error.

class Example {

static int x = 100;

public static int getX() {

int lx = 200;

println(lx);

return x;

}

static void main(String[] args) {

println(getX());

}

}

When we run the above program, we will get the following result.

200

100

## this method for Properties

Just like in Java, groovy can access its instance members using the **this** keyword. The following example shows how when we use the statement **this.x**, it refers to its instance and sets the value of **x** accordingly.

class Example {

int x = 100;

public int getX() {

this.x = 200;

return x;

}

static void main(String[] args) {

Example ex = new Example();

println(ex.getX());

}

}

When we run the above program, we will get the result of 200 printed on the console.

# Groovy – File I/O

Groovy provides a number of helper methods when working with I/O. Groovy provides easier classes to provide the following functionalities for files.

* Reading files
* Writing to files
* Traversing file trees
* Reading and writing data objects to files

In addition to this, you can always use the normal Java classes listed below for File I/O operations.

* java.io.File
* java.io.InputStream
* java.io.OutputStream
* java.io.Reader
* java.io.Writer

## Reading files

The following example will output all the lines of a text file in Groovy. The method **eachLine** is in-built in the File class in Groovy for the purpose of ensuring that each line of the text file is read.

import java.io.File

class Example {

static void main(String[] args) {

new File("E:/Example.txt").eachLine {

line -> println "line : $line";

}

}

}

The File class is used to instantiate a new object which takes the file name as the parameter. It then takes the function of eachLine, puts it to a variable called line and prints it accordingly.

If the file contains the following lines, they will be printed.

line : Example1

line : Example2

## Reading the Contents of a File as an Entire String

If you want to get the entire contents of the file as a string, you can use the text property of the file class. The following example shows how this can be done.

class Example {

static void main(String[] args) {

File file = new File("E:/Example.txt")

println file.text

}

}

If the file contains the following lines, they will be printed.

line : Example1

line : Example2

## Writing to Files

If you want to write to files, you need to use the writer class to output text to a file. The following example shows how this can be done.

import java.io.File

class Example {

static void main(String[] args) {

new File('E:/','Example.txt').withWriter('utf-8') {

writer -> writer.writeLine 'Hello World'

}

}

}

If you open the file Example.txt, you will see the words “Hello World” printed to the file.

## Getting the Size of a File

If you want to get the size of the file one can use the length property of the file class to get the size of the file. The following example shows how this can be done.

class Example {

static void main(String[] args) {

File file = new File("E:/Example.txt")

println "The file ${file.absolutePath} has ${file.length()} bytes"

}

}

The above code would show the size of the file in bytes.

## Testing if a File is a Directory

If you want to see if a path is a file or a directory, one can use the **isFile** and **isDirectory** option of the File class. The following example shows how this can be done.

class Example {

static void main(String[] args) {

def file = new File('E:/')

println "File? ${file.isFile()}"

println "Directory? ${file.isDirectory()}"

}

}

The above code would show the following output −

File? false

Directory? True

## Creating a Directory

If you want to create a new directory you can use the **mkdir** function of the File class. The following example shows how this can be done.

class Example {

static void main(String[] args) {

def file = new File('E:/Directory')

file.mkdir()

}

}

The directory E:\Directory will be created if it does not exist.

## Deleting a File

If you want to delete a file you can use the delete function of the File class. The following example shows how this can be done.

class Example {

static void main(String[] args) {

def file = new File('E:/Example.txt')

file.delete()

}

}

The file will be deleted if it exists.

## Copying files

Groovy also provides the functionality to copy the contents from one file to another. The following example shows how this can be done.

class Example {

static void main(String[] args) {

def src = new File("E:/Example.txt")

def dst = new File("E:/Example1.txt")

dst << src.text

}

}

The file Example1.txt will be created and all of the contents of the file Example.txt will be copied to this file.

## Getting Directory Contents

Groovy also provides the functionality to list the drives and files in a drive.

The following example shows how the drives on a machine can be displayed by using the **listRoots** function of the File class.

class Example {

static void main(String[] args) {

def rootFiles = new File("test").listRoots()

rootFiles.each {

file -> println file.absolutePath

}

}

}

Depending on the drives available on your machine, the output could vary. On a standard machine the output would be similar to the following one −

C:\

D:\

The following example shows how to list the files in a particular directory by using the **eachFile** function of the File class.

class Example {

static void main(String[] args) {

new File("E:/Temp").eachFile() {

file->println file.getAbsolutePath()

}

}

}

The output would display all of the files in the directory E:\Temp

If you want to recursively display all of files in a directory and its subdirectories, then you would use the **eachFileRecurse** function of the File class. The following example shows how this can be done.

class Example {

static void main(String[] args) {

new File("E:/temp").eachFileRecurse() {

file -> println file.getAbsolutePath()

}

}

}

The output would display all of the files in the directory E:\Temp and in its subdirectories if they exist.

# Groovy – Optionals

Groovy is an “optionally” typed language, and that distinction is an important one when understanding the fundamentals of the language. When compared to Java, which is a “strongly” typed language, whereby the compiler knows all of the types for every variable and can understand and honor contracts at compile time. This means that method calls are able to be determined at compile time.

When writing code in Groovy, developers are given the flexibility to provide a type or not. This can offer some simplicity in implementation and, when leveraged properly, can service your application in a robust and dynamic way.

In Groovy, optional typing is done via the ‘def’ keyword. Following is an example of the usage of the **def** method −

class Example {

static void main(String[] args) {

// Example of an Integer using def

def a = 100;

println(a);

// Example of an float using def

def b = 100.10;

println(b);

// Example of an Double using def

def c = 100.101;

println(c);

// Example of an String using def

def d = "HelloWorld";

println(d);

}

}

From the above program, we can see that we have not declared the individual variables as Integer, float, double, or string even though they contain these types of values.

When we run the above program, we will get the following result −

100

100.10

100.101

HelloWorld

Optional typing can be a powerful utility during development but can lead to problems in maintainability during the later stages of development when the code becomes too vast and complex.

To get a handle on how you can utilize optional typing in Groovy without getting your codebase into an unmaintainable mess, it is best to embrace the philosophy of “duck typing” in your applications.

The name comes from the phrase "If it looks like a duck and quacks like a duck, it's a duck".

If we re-write the above code using duck typing, it would look like the one given below. The variable names are given names which resemble more often than not the type they represent which makes the code more understandable.

class Example {

static void main(String[] args) {

// Example of an Integer using def

def aint = 100;

println(aint);

// Example of an float using def

def bfloat = 100.10;

println(bfloat);

// Example of an Double using def

def cDouble = 100.101;

println(cDouble);

// Example of an String using def

def dString = "HelloWorld";

println(dString);

}

}

# Groovy – Numbers

In Groovy, Numbers are actually represented as object’s, all of them being an instance of the class Integer. To make an object do something, we need to invoke one of the methods declared in its class.

Groovy supports integer and floating point numbers.

* An integer is a value that does not include a fraction.
* A floating-point number is a decimal value that includes a decimal fraction.

An Example of numbers in Groovy is shown below −

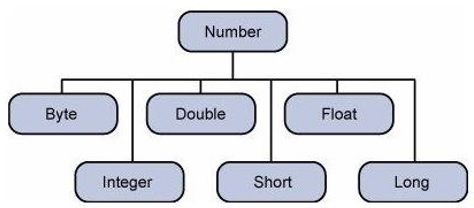
Integer x = 5;

Float y = 1.25;

Where **x** is of the type Integer and **y** is the float.

The reason why numbers in groovy are defined as objects is generally because there are requirements to perform operations on numbers. The concept of providing a class over primitive types is known as wrapper classes.

By default, the following wrapper classes are provided in Groovy.



The object of the wrapper class contains or wraps its respective primitive data type. The process of converting a primitive data types into object is called boxing, and this is taken care by the compiler. The process of converting the object back to its corresponding primitive type is called unboxing.

**Example**

Following is an example of boxing and unboxing −

class Example {

static void main(String[] args) {

Integer x = 5,y = 10,z = 0;

// The the values of 5,10 and 0 are boxed into Integer types

// The values of x and y are unboxed and the addition is performed

z = x+y;

println(z);

}

}

The output of the above program would be 15. In the above example, the values of 5, 10, and 0 are first boxed into the Integer variables x, y and z accordingly. And then the when the addition of x and y is performed the values are unboxed from their Integer types.

## Some Number Methods

Since the Numbers in Groovy are represented as classes, following are the list of some methods available:

### xxxValue()

This method takes on the Number as the parameter and returns a primitive type based on the method which is invoked. Following are the list of methods available −

byte byteValue()

short shortValue()

int intValue()

long longValue()

float floatValue()

double doubleValue()

**Parameters** − No parameters required.

**Return Value** − The return value is the primitive type returned depending on the value function which is called.

Following is an example of the usage of the method values.

class Example {

static void main(String[] args) {

Integer x = 5;

// Converting the number to double primitive type

println(x.doubleValue());

// Converting the number to byte primitive type

println(x.byteValue());

// Converting the number to float primitive type

println(x.floatValue());

// Converting the number to long primitive type

println(x.longValue());

// Converting the number to short primitive type

println(x.shortValue());

// Converting the number to int primitive type

println(x.intValue());

}

}

When we run the above program, we will get the following result −

5.0

5

5.0

5

5

5

### compareTo()

The compareTo method is to use compare one number against another. This is useful if you want to compare the value of numbers.

**Syntax**

public int compareTo( NumberSubClass referenceName )

**Parameters**

referenceName - This could be a Byte, Double, Integer, Float, Long or Short.

**Return Value**

* If the Integer is equal to the argument then 0 is returned.
* If the Integer is less than the argument then -1 is returned.
* If the Integer is greater than the argument then 1 is returned.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

Integer x = 5;

//Comparison against a Integer of lower value

System.out.println(x.compareTo(3));

//Comparison against a Integer of equal value

System.out.println(x.compareTo(5));

//Comparison against a Integer of higher value

System.out.println(x.compareTo(8));

}

}

When we run the above program, we will get the following result −

1

0

-1

### equals()

The method determines whether the Number object that invokes the method is equal to the object that is passed as argument.

**Syntax**

public boolean equals(Object o)

**Parameters**

* o - Any object.

**Return Value**

The method returns True if the argument is not null and is an object of the same type and with the same numeric value.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

Integer x = 5;

Integer y = 10;

Integer z = 5;

//Comparison against an Integer of different value

System.out.println(x.equals(y));

//Comparison against an Integer of same value

System.out.println(x.equals(z));

}

}

When we run the above program, we will get the following result −

false

true

### toString()

The method is used to get a String object representing the value of the Number Object.

If the method takes a primitive data type as an argument, then the String object representing the primitive data type value is returned.

If the method takes two arguments, then a String representation of the first argument in the radix specified by the second argument will be returned.

**Syntax**

* String toString()
* static String toString(int i)

**Parameters**

* i − An int for which string representation would be returned.

**Return Value**

* toString() − This returns a String object representing the value of this Integer.
* toString(int i) − This returns a String object representing the specified integer.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

Integer x = 5;

System.out.println(x.toString());

System.out.println(Integer.toString(12));

}

}

When we run the above program, we will get the following result −

5

12

### parseInt()

This method is used to get the primitive data type of a certain String. parseXxx() is a static method and can have one argument or two.

**Syntax**

static int parseInt(String s)

static int parseInt(String s, int radix)

**Parameters**

* **s** − This is a string representation of decimal.
* **radix** − This would be used to convert String s into integer.

**Return Value**

* **parseInt(String s)** − This returns an integer (decimal only).
* **parseInt(int i)** − This returns an integer, given a string representation of decimal, binary, octal, or hexadecimal (radix equals 10, 2, 8, or 16 respectively) numbers as input.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

int x = Integer.parseInt("9");

double y = Double.parseDouble("5");

System.out.println(x);

System.out.println(y);

}

}

When we run the above program, we will get the following result −

9

5.0

### random()

The method is used to generate a random number between 0.0 and 1.0 (excluding 0 and 1). The range is: 0.0 =< Math.random < 1.0. Different ranges can be achieved by using arithmetic.

**Syntax**

static double random()

**Parameters**

This is a default method and accepts no parameter.

**Return Value**

This method returns a double.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

System.out.println( Math.random() );

System.out.println( Math.random() );

}

}

When we run the above program, we will get the following result −

0.0543333676591804

0.3223824169137166

### Other Examples of using the Random class

Random rnd = new Random()

println(rnd.next(2)) // 2 bits of random number that is, one of the following: 0,1,2,3

println(rnd.nextInt(3)) // random integer in the range of 0, 3 (so one of 0,1, 2)

In the next example we have a list of one-letter strings and we would like to pick one of the elements randomly. So we need an integer between 0 and the size of the array.

We run it in a loop so we can see more values picked.

def z = ["a", "b", "c", "d", "e"]

Random rnd = new Random()

for (i=0; i < 10; i++) {

println(z[rnd.nextInt(z.size)])

}

### Other Number Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| valueOf() | abs() | round() | max() | min() |

# Groovy – Strings

A String literal is constructed in Groovy by enclosing the string text in quotations.

Groovy offers a variety of ways to denote a String literal. Strings in Groovy can be enclosed in single quotes (’), double quotes (“), or triple quotes (“””). Further, a Groovy String enclosed by triple quotes may span multiple lines.

Following is an example of the usage of strings in Groovy −

class Example {

static void main(String[] args) {

String a = 'Hello Single';

String b = "Hello Double";

String c = "'Hello Triple" + "Multiple lines'";

println(a);

println(b);

println(c);

}

}

When we run the above program, we will get the following result −

Hello Single

Hello Double

'Hello TripleMultiple lines'

## String Indexing

Strings in Groovy are an ordered sequences of characters. The individual character in a string can be accessed by its position. This is given by an index position.

String indices start at zero and end at one less than the length of the string. Groovy also permits negative indices to count back from the end of the string.

Following is an example of the usage of string indexing in Groovy −

class Example {

static void main(String[] args) {

String sample = "Hello world";

println(sample[4]); // Print the 5 character in the string

//Print the 1st character in the string starting from the back

println(sample[-1]);

println(sample[1..2]);//Prints a string starting from Index 1 to 2

println(sample[4..2]);//Prints a string starting from Index 4 back to 2

}

}

When we run the above program, we will get the following result −

o

d

el

oll

## Basic String Operations

### Concatenation of Two Strings

The concatenation of strings can be done by the simple ‘+’ operator.

String+String

**Parameters** − The parameters will be 2 strings as the left and right operand for the + operator.

**Return Value** − The return value is a string

Following is an example of the string concatenation in Groovy.

class Example {

static void main(String[] args) {

String a = "Hello";

String b = "World";

println("Hello" + "World");

println(a + b);

}

}

When we run the above program, we will get the following result −

HelloWorld

HelloWorld

### String Repetition

The repetition of strings can be done by the simple ‘\*’ operator.

String\*number

**Parameters**

The parameters will be

* A string as the left operand for the \* operator
* A number at the right side of the operator to indicate the number of times the strings needs to be repeated.

**Return Value**

The return value is a string.

Following is an example of the usage of strings in Groovy −

class Example {

static void main(String[] args) {

String a = "Hello";

println("Hello"\*3);

println(a\*3);

}

}

When we run the above program, we will get the following result −

HelloHelloHello

HelloHelloHello

### String Length

**Syntax** − The length of the string determined by the length() method of the string.

**Parameters** − No parameters.

**Return Value** − An Integer showing the length of the string.

Following is an example of the usage of strings in Groovy −

class Example {

static void main(String[] args) {

String a = "Hello";

println(a.length());

}

}

When we run the above program, we will get the following result −

5

## Other String Methods

|  |  |
| --- | --- |
| **S.No.** | **Methods & Description** |
| 1 | [center()](https://www.tutorialspoint.com/groovy/groovy_center.htm)  Returns a new String of length numberOfChars consisting of the recipient padded on the left and right with space characters. |
| 2 | [compareToIgnoreCase()](https://www.tutorialspoint.com/groovy/groovy_comparetoignorecase.htm)  Compares two strings lexicographically, ignoring case differences. |
| 3 | [concat()](https://www.tutorialspoint.com/groovy/groovy_concat.htm)  Concatenates the specified String to the end of this String. |
| 4 | [eachMatch()](https://www.tutorialspoint.com/groovy/groovy_eachmatch.htm)  Processes each regex group (see next section) matched substring of the given String. |
| 5 | [endsWith()](https://www.tutorialspoint.com/groovy/groovy_endswith.htm)  Tests whether this string ends with the specified suffix. |
| 6 | [equalsIgnoreCase()](https://www.tutorialspoint.com/groovy/groovy_equalsignorecase.htm)  Compares this String to another String, ignoring case considerations. |
| 7 | [getAt()](https://www.tutorialspoint.com/groovy/groovy_getat.htm)  It returns string value at the index position |
| 8 | [indexOf()](https://www.tutorialspoint.com/groovy/groovy_indexof.htm)  Returns the index within this String of the first occurrence of the specified substring. |
| 9 | [matches()](https://www.tutorialspoint.com/groovy/groovy_matches.htm)  It outputs whether a String matches the given regular expression. |
| 10 | [minus()](https://www.tutorialspoint.com/groovy/groovy_minus.htm)  Removes the value part of the String. |
| 11 | [next()](https://www.tutorialspoint.com/groovy/groovy_next.htm)  This method is called by the ++ operator for the class String. It increments the last character in the given String. |
| 12 | [padLeft()](https://www.tutorialspoint.com/groovy/groovy_padleft.htm)  Pad the String with the spaces appended to the left. |
| 13 | [padRight()](https://www.tutorialspoint.com/groovy/groovy_padright.htm)  Pad the String with the spaces appended to the right. |
| 14 | [plus()](https://www.tutorialspoint.com/groovy/groovy_plus.htm)  Appends a String |
| 15 | [previous()](https://www.tutorialspoint.com/groovy/groovy_previous.htm)  This method is called by the -- operator for the CharSequence. |
| 16 | [replaceAll()](https://www.tutorialspoint.com/groovy/groovy_replaceall.htm)  Replaces all occurrences of a captured group by the result of a closure on that text. |
| 17 | [reverse()](https://www.tutorialspoint.com/groovy/groovy_strings_reverse.htm)  Creates a new String which is the reverse of this String. |
| 18 | [split()](https://www.tutorialspoint.com/groovy/groovy_split.htm)  Splits this String around matches of the given regular expression. |
| 19 | [subString()](https://www.tutorialspoint.com/groovy/groovy_substring.htm)  Returns a new String that is a substring of this String. |
| 20 | [toUpperCase()](https://www.tutorialspoint.com/groovy/groovy_touppercase.htm)  Converts all of the characters in this String to upper case. |
| 21 | [toLowerCase()](https://www.tutorialspoint.com/groovy/groovy_tolowercase.htm)  Converts all of the characters in this String to lower case. |

# Groovy – Ranges

A range is shorthand for specifying a sequence of values. A Range is denoted by the first and last values in the sequence, and Range can be inclusive or exclusive. An inclusive Range includes all the values from the first to the last, while an exclusive Range includes all values except the last. Here are some examples of Range literals −

* 1..10 - An example of an inclusive Range
* 1..<10 - An example of an exclusive Range
* ‘a’..’x’ – Ranges can also consist of characters
* 10..1 – Ranges can also be in descending order
* ‘x’..’a’ – Ranges can also consist of characters and be in descending order.

Following are the various methods available for ranges.

## contains()

Checks if a range contains a specific value.

**Syntax**

boolean contains(Object obj)

**Parameters**

**Obj** − The value to check in the range list.

**Return Value**

Returns true if this Range contains the specified element.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

// Example of an Integer using def

def rint = 1..10;

println(rint.contains(2));

println(rint.contains(11));

}

}

When we run the above program, we will get the following result −

true

false

## get()

Returns the element at the specified position in this Range.

**Syntax**

Object get(int index)

**Parameters**

Index – The index value to get from the range.

**Return Value**

The range value at the particular index.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

// Example of an Integer using def

def rint = 1..10;

println(rint.get(2));

println(rint.get(4));

}

}

When we run the above program, we will get the following result −

3

5

## getFrom()

Get the lower value of this Range.

**Syntax**

Comparable getFrom()

**Parameters**

None

**Return Value**

The lower value of the range.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

// Example of an Integer using def

def rint = 1..10;

println(rint.getFrom());

}

}

When we run the above program, we will get the following result −

1

## getTo()

Get the upper value of this Range.

**Syntax**

Comparable getTo()

**Parameters**

None

**Return Value**

The upper value of the range.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

// Example of an Integer using def

def rint = 1..10;

println(rint.getTo());

}

}

When we run the above program, we will get the following result −

10

## isReverse()

Is this a reversed Range, iterating backwards?

**Syntax**

boolean isReverse()

**Parameters**

None

**Return Value**

Boolean value of true or false on whether the range is reversed.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

// Example of an Integer using def

def rint = 1..10;

println(rint.isReverse());

}

}

When we run the above program, we will get the following result −

false

## size()

Returns the number of elements in this Range.

**Syntax**

int size()

**Parameters**

None

**Return Value**

Returns the size of the range.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

// Example of an Integer using def

def rint = 1..10;

println(rint.size());

}

}

When we run the above program, we will get the following result −

10

## subList()

Returns a view of the portion of this Range between the specified fromIndex, inclusive, and toIndex, exclusive

**Syntax**

List subList(int fromIndex, int toIndex)

**Parameters**

* fromIndex – Starting index of the range
* toIndex – End Index of the range

**Return Value**

The list of range values from specified starting to ending index.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

def rint = 1..10;

println(rint.subList(1,4));

println(rint.subList(4,8));

}

}

When we run the above program, we will get the following result −

[2, 3, 4]

[5, 6, 7, 8]

# Groovy – Lists

The List is a structure used to store a collection of data items. In Groovy, the List holds a sequence of object references. Object references in a List occupy a position in the sequence and are distinguished by an integer index. A List literal is presented as a series of objects separated by commas and enclosed in square brackets.

To process the data in a list, we must be able to access individual elements. Groovy Lists are indexed using the indexing operator []. List indices start at zero, which refers to the first element.

Following are some example of lists −

* [11, 12, 13, 14] – A list of integer values
* [‘Angular’, ‘Groovy’, ‘Java’] – A list of Strings
* [1, 2, [3, 4], 5] – A nested list
* [‘Groovy’, 21, 2.11] – A heterogeneous list of object references
* [ ] – An empty list

## Iterate a List

def lst = ['foo', 'bar', 'baz']

// using implicit argument

lst.each { println it }

// using explicit argument

lst.each { val -> println val }

## Iterate with index[#](https://riptutorial.com/groovy/example/18003/iterate-over-a-collection#undefined)

def lst = ['foo', 'bar', 'baz']

// explicit arguments are required

lst.eachWithIndex { val, idx -> println "$val in position $idx" }​​​​​​​​​​​​​​

// prints:

// foo in position 0

// bar in position 1

// baz in position 2

## add()

Append the new value to the end of this List. This method has 2 different variants.

* **boolean add(Object value)** − Append the new value to the end of this List.

**Syntax**

boolean add(Object value)

**Parameters**

* value – Value to be appended to the list.

**Return Value − A Boolean value on whether the value was added.**

* **void add(int index, Object value)** − Append the new value to a particular position in the List.

**Syntax**

void add(int index, Object value)

**Parameters**

* value – Value to be appended to the list.
* Index – the index where the value needs to be added.

**Return Value − None**

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

def lst = [11, 12, 13, 14];

println(lst);

lst.add(15);

println(lst);

lst.add(2,20);

println(lst);

}

}

When we run the above program, we will get the following result −

[11, 12, 13, 14]

[11, 12, 13, 14, 15]

[11, 12, 20, 13, 14, 15]

## contains()

Returns true if this List contains the specified value.

**Syntax**

boolean contains(Object value)

**Parameters**

**Value** − The value to find in the list.

**Return Value**

True or false depending on if the value is present in the list.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

def lst = [11, 12, 13, 14];

println(lst.contains(12));

println(lst.contains(18));

}

}

When we run the above program, we will get the following result −

true

false

## get()

Returns the element at the specified position in this List.

**Syntax**

Object get(int index)

**Parameters**

Index – The index at which the value needs to be returned.

**Return Value**

The value at the index position in the list.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

def lst = [11, 12, 13, 14];

println(lst.get(0));

println(lst.get(2));

}

}

When we run the above program, we will get the following result −

11

13

## isEmpty()

Returns true if this List contains no elements.

**Syntax**

boolean isEmpty()

**Parameters**

None

**Return Value**

True or false depending on whether the list is empty or not.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

def lst = [11, 12, 13, 14];

def emptylst = [];

println(lst.isEmpty());

println(emptylst.isEmpty());

}

}

When we run the above program, we will get the following result −

false

true

## minus()

Creates a new List composed of the elements of the original without those specified in the collection.

**Syntax**

List minus(Collection collection)

**Parameters**

Collection – The collection of values to minus from the list.

**Return Value**

New list of values.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

def lst = [11, 12, 13, 14];

def newlst = [];

newlst = lst.minus([12,13]);

println(newlst);

}

}

When we run the above program, we will get the following result −

[11, 14]

## plus()

Creates a new List composed of the elements of the original together with those specified in the collection.

**Syntax**

List plus(Collection collection)

**Parameters**

Collection – The collection of values to add to the list.

**Return Value**

New list of values.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

def lst = [11, 12, 13, 14];

def newlst = [];

newlst = lst.plus([15,16]);

println(newlst);

}

}

When we run the above program, we will get the following result −

[11, 12, 13, 14, 15, 16]

## pop()

Removes the last item from this List.

**Syntax**

Object pop()

**Parameters**

None

**Return Value**

The popped value from the list.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

def lst = [11, 12, 13, 14];

println(lst.pop());

println(lst);

}

}

When we run the above program, we will get the following result −

14

[11, 12, 13]

## remove()

Removes the element at the specified position in this List.

**Syntax**

Object remove(int index)

**Parameters**

Index – Index at which the value needs to be removed.

**Return Value**

The removed value.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

def lst = [11, 12, 13, 14];

println(lst.remove(2));

println(lst);

}

}

When we run the above program, we will get the following result −

13

[11, 12, 14]

## reverse()

Create a new List that is the reverse the elements of the original List.

**Syntax**

List reverse()

**Parameters**

None

**Return Value**

The reversed list.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

def lst = [11, 12, 13, 14];

def revlst = lst.reverse();

println(revlst);

}

}

When we run the above program, we will get the following result −

[14, 13, 12, 11]

## size()

Obtains the number of elements in this List.

**Syntax**

int size()

**Parameters**

None

**Return Value**

The size of the list.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

def lst = [11, 12, 13, 14];

println(lst.size);

}

}

When we run the above program, we will get the following result −

4

## sort()

Returns a sorted copy of the original List.

**Syntax**

List sort()

**Parameters**

None

**Return Value**

The sorted list.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

def lst = [13, 12, 15, 14];

def newlst = lst.sort();

println(newlst);

}

}

When we run the above program, we will get the following result −

[12, 13, 14, 15]

# Groovy – Maps

A Map (also known as an associative array, dictionary, table, and hash) is an unordered collection of object references. The elements in a Map collection are accessed by a key value. The keys used in a Map can be of any class. When we insert into a Map collection, two values are required: the key and the value.

Following are some examples of maps −

* [‘TopicName’ : ‘Lists’, ‘Author’ : ‘Raghav’] – Collections of key value pairs which has TopicName as the key and their respective values.
* [ : ] – An Empty map.

## Iterate a Map

def map = [foo: 'FOO', bar: 'BAR', baz: 'BAZ']

// using implicit argument

map.each { println "key: ${it.key}, value: ${it.value}"}

// using explicit arguments

map.each { k, v -> println "key: $k, value: $v"}

## containsKey()

Does this Map contain this key?

**Syntax**

boolean containsKey(Object key)

**Parameters**

**Key** − The key used to search for.

**Return Value**

True or false depending on whether the key value is there or not.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

def mp = ["TopicName" : "Maps", "TopicDescription" : "Methods in Maps"]

println(mp.containsKey("TopicName"));

println(mp.containsKey("Topic"));

}

}

When we run the above program, we will get the following result −

true

false

## get()

Look up the key in this Map and return the corresponding value. If there is no entry in this Map for the key, then return null.

**Syntax**

Object get(Object key)

**Parameters**

**Key** − Key to search for retrieval.

**Return Value**

The key-value pair or NULL if it does not exist.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

def mp = ["TopicName" : "Maps", "TopicDescription" : "Methods in Maps"]

println(mp.get("TopicName"));

println(mp.get("Topic"));

}

}

When we run the above program, we will get the following result −

Maps

Null

## keyset()

Obtain a Set of the keys in this Map.

**Syntax**

Set keySet()

**Parameters**

None.

**Return Value**

Set of Keys.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

def mp = ["TopicName" : "Maps", "TopicDescription" : "Methods in Maps"]

println(mp.keySet());

}

}

When we run the above program, we will get the following result −

[TopicName, TopicDescription]

## values()

Returns a collection view of the values contained in this Map.

**Syntax**

Collection values()

**Parameters**

None.

**Return Value**

Collection of values.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

def mp = ["TopicName" : "Maps", "TopicDescription" : "Methods in Maps"]

println(mp.values());

}

}

When we run the above program, we will get the following result −

[Maps, Methods in Maps]

## put()

Associates the specified value with the specified key in this Map. If this Map previously contained a mapping for this key, the old value is replaced by the specified value.

**Syntax**

Object put(Object key, Object value)

**Parameters**

* Key – The key to be put in the map
* Value – The associated value for the key

**Return Value**

The returned key-value pair which is inserted.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

def mp = ["TopicName" : "Maps", "TopicDescription" : "Methods in Maps"]

mp.put("TopicID","1");

println(mp);

}

}

When we run the above program, we will get the following result −

[TopicName:Maps, TopicDescription:Methods in Maps, TopicID:1]

## size()

Returns the number of key-value mappings in this Map.

**Syntax**

int size()

**Parameters**

None.

**Return Value**

The size of the map.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

def mp = ["TopicName" : "Maps", "TopicDescription" : "Methods in Maps"]

println(mp.size());

mp.put("TopicID","1");

println(mp.size());

}

}

When we run the above program, we will get the following result −

2

3

# Groovy – Dates and Times

The class Date represents a specific instant in time, with millisecond precision. The Date class has two constructors as shown below.

## Date()

**Syntax**

public Date()

**Parameters** − None.

**Return Value**

Allocates a Date object and initializes it so that it represents the time at which it was allocated, measured to the nearest millisecond.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

Date date = new Date();

// display time and date using toString()

System.out.println(date.toString());

}

}

When we run the above program, we will get the following result. The following output will give you the current date and time −

Thu Dec 10 21:31:15 GST 2015

## Date (long millisec)

**Syntax**

public Date(long millisec)

**Parameters**

Millisec – The number of millisecconds to specify since the standard base time.

**Return Value**

Allocates a **Date** object and initializes it to represent the specified number of milliseconds since the standard base time known as "the epoch", namely January 1, 1970, 00:00:00 GMT.

Following is an example of the usage of this method −

class Example {

static void main(String[] args) {

Date date = new Date(100);

// display time and date using toString()

System.out.println(date.toString());

}

}

When we run the above program, we will get the following result −

Thu Jan 01 04:00:00 GST 1970

Following are the given methods of the Date class. In all methods of class Date that accept or return year, month, date, hours, minutes, and seconds values, the following representations are used −

* A year y is represented by the integer y - 1900.
* A month is represented by an integer from 0 to 11; 0 is January, 1 is February, and so forth; thus 11 is December.
* A date (day of month) is represented by an integer from 1 to 31 in the usual manner.
* An hour is represented by an integer from 0 to 23. Thus, the hour from midnight to 1 a.m. is hour 0, and the hour from noon to 1 p.m. is hour 12.
* A minute is represented by an integer from 0 to 59 in the usual manner.
* A second is represented by an integer from 0 to 61.

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | [after()](https://www.tutorialspoint.com/groovy/groovy_after.htm) Tests if this date is after the specified date.  public boolean after(Date when)  class Example {  static void main(String[] args) {  Date olddate = new Date("05/11/2015");  Date newdate = new Date("05/12/2015");  Date latestdate = new Date();    System.out.println(olddate.after(newdate));  System.out.println(latestdate.after(newdate));  }  } |
| 2 | [equals()](https://www.tutorialspoint.com/groovy/groovy_dates_times_equals.htm)  Compares two dates for equality. The result is true if and only if the argument is not null and is a Date object that represents the same point in time, to the millisecond, as this object.  public boolean equals(Object obj)  class Example {  static void main(String[] args) {  Date olddate = new Date("05/11/2015");  Date newdate = new Date("05/11/2015");  Date latestdate = new Date();    System.out.println(olddate.equals(newdate));  System.out.println(latestdate.equals(newdate));  }  } |
| 3 | [compareTo()](https://www.tutorialspoint.com/groovy/groovy_dates_times_compareto.htm) Compares two Dates for ordering.  public int compareTo(Date anotherDate)  class Example {  static void main(String[] args) {  Date olddate = new Date("05/11/2015");  Date newdate = new Date("05/11/2015");  Date latestdate = new Date();    System.out.println(olddate.compareTo(newdate));  System.out.println(latestdate.compareTo(newdate));  }  } |
| 4 | [toString()](https://www.tutorialspoint.com/groovy/groovy_dates_times_tostring.htm) Converts this Date object to a String of the form dow mon dd hh:mm:ss zzz yyyy  public String toString()  class Example {  static void main(String[] args) {  Date olddate = new Date("05/11/2015");  Date newdate = new Date("05/11/2015");  Date latestdate = new Date();    System.out.println(olddate.toString());  System.out.println(newdate.toString());  System.out.println(latestdate.toString());  }  } |
| 5 | [before()](https://www.tutorialspoint.com/groovy/groovy_before.htm) Tests if this date is before the specified date.  public boolean before(Date when)  class Example {  static void main(String[] args) {  Date olddate = new Date("05/11/2015");  Date newdate = new Date("05/11/2015");  Date latestdate = new Date();    System.out.println(olddate.before(newdate));  System.out.println(olddate.before(latestdate));  }  } |
| 6 | [getTime()](https://www.tutorialspoint.com/groovy/groovy_gettime.htm) Returns the number of milliseconds since January 1, 1970, 00:00:00 GMT represented by this Date object.  public long getTime()  class Example {  static void main(String[] args) {  Date olddate = new Date("05/11/2015");  Date newdate = new Date("05/11/2015");  Date latestdate = new Date();    System.out.println(olddate.getTime());  System.out.println(newdate.getTime());  System.out.println(latestdate.getTime());  }  } |
| 7 | [setTime()](https://www.tutorialspoint.com/groovy/groovy_settime.htm) Sets this Date object to represent a point in time that is time milliseconds after January 1, 1970 00:00:00 GMT.  public void setTime(long time)  class Example {  static void main(String[] args) {  Date olddate = new Date("05/11/2015");  Date newdate = new Date("05/12/2015");  Date latestdate = new Date();    olddate.setTime(10000);  newdate.setTime(10000);  latestdate.setTime(10000);    System.out.println(olddate.toString());  System.out.println(newdate.toString());  System.out.println(latestdate.toString());  }  } |

# Exception Handling

Exception handling is required in any programming language to handle the runtime errors so that normal flow of the application can be maintained.

Exception normally disrupts the normal flow of the application, which is the reason why we need to use Exception handling in our application.

Exceptions are broadly classified into the following categories −

* **Checked Exception** − The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g.IOException, SQLException etc. Checked exceptions are checked at compile-time.

One classical case is the FileNotFoundException. Suppose you had the following codein your application which reads from a file in E drive.

class Example {

static void main(String[] args) {

File file = new File("E://file.txt");

FileReader fr = new FileReader(file);

}

}

If the File (file.txt) is not there in the E drive then the following exception will be raised.

Caught: java.io.FileNotFoundException: E:\file.txt (The system cannot find the file specified).

java.io.FileNotFoundException: E:\file.txt (The system cannot find the file specified).

* **Unchecked Exception** − The classes that extend RuntimeException are known as unchecked exceptions, e.g., ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

One classical case is the ArrayIndexOutOfBoundsException which happens when you try to access an index of an array which is greater than the length of the array. Following is a typical example of this sort of mistake.

class Example {

static void main(String[] args) {

def arr = new int[3];

arr[5] = 5;

}

}

When the above code is executed the following exception will be raised.

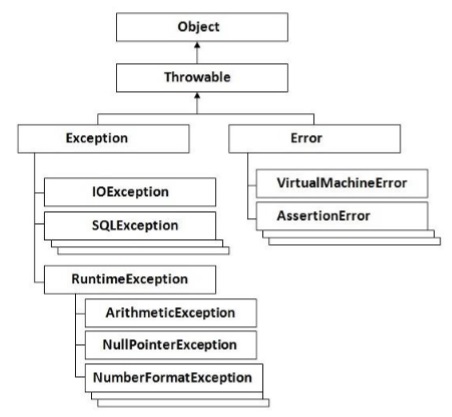
Caught: java.lang.ArrayIndexOutOfBoundsException: 5

java.lang.ArrayIndexOutOfBoundsException: 5

* **Error** − Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

These are errors which the program can never recover from and will cause the program to crash.

The following diagram shows how the hierarchy of exceptions in Groovy is organized. It’s all based on the hierarchy defined in Java.



## Catching Exceptions

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception.

try {

//Protected code

} catch(ExceptionName e1) {

//Catch block

}

All of your code which could raise an exception is placed in the Protected code block.

In the catch block, you can write custom code to handle your exception so that the application can recover from the exception.

Let’s look at an example of the similar code we saw above for accessing an array with an index value which is greater than the size of the array. But this time let’s wrap our code in a try/catch block.

class Example {

static void main(String[] args) {

try {

def arr = new int[3];

arr[5] = 5;

} catch(Exception ex) {

println("Catching the exception");

}

println("Let's move on after the exception");

}

}

When we run the above program, we will get the following result −

Catching the exception

Let's move on after the exception

From the above code, we wrap out faulty code in the try block. In the catch block we are just catching our exception and outputting a message that an exception has occurred.

## Multiple Catch Blocks

One can have multiple catch blocks to handle multiple types of exceptions. For each catch block, depending on the type of exception raised you would write code to handle it accordingly.

Let’s modify our above code to catch the ArrayIndexOutOfBoundsException specifically. Following is the code snippet.

class Example {

static void main(String[] args) {

try {

def arr = new int[3];

arr[5] = 5;

}catch(ArrayIndexOutOfBoundsException ex) {

println("Catching the Array out of Bounds exception");

}catch(Exception ex) {

println("Catching the exception");

}

println("Let's move on after the exception");

}

}

When we run the above program, we will get the following result −

Catching the Aray out of Bounds exception

Let's move on after the exception

From the above code you can see that the ArrayIndexOutOfBoundsException catch block is caught first because it means the criteria of the exception.

## Finally Block

The **finally** block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code. The syntax for this block is given below.

try {

//Protected code

} catch(ExceptionType1 e1) {

//Catch block

} catch(ExceptionType2 e2) {

//Catch block

} catch(ExceptionType3 e3) {

//Catch block

} finally {

//The finally block always executes.

}

Let’s modify our above code and add the finally block of code. Following is the code snippet.

class Example {

static void main(String[] args) {

try {

def arr = new int[3];

arr[5] = 5;

} catch(ArrayIndexOutOfBoundsException ex) {

println("Catching the Array out of Bounds exception");

}catch(Exception ex) {

println("Catching the exception");

} finally {

println("The final block");

}

println("Let's move on after the exception");

}

}

When we run the above program, we will get the following result −

Catching the Array out of Bounds exception

The final block

Let's move on after the exception

Following are the Exception methods available in Groovy −

### public String getMessage()

Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.

### public Throwable getCause()

Returns the cause of the exception as represented by a Throwable object.

### public String toString()

Returns the name of the class concatenated with the result of getMessage()

### public void printStackTrace()

Prints the result of toString() along with the stack trace to System.err, the error output stream.

### public StackTraceElement [] getStackTrace()

Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.

### public Throwable fillInStackTrace()

Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

Following is the code example using some of the methods given above −

class Example {

static void main(String[] args) {

try {

def arr = new int[3];

arr[5] = 5;

}catch(ArrayIndexOutOfBoundsException ex) {

println(ex.toString());

println(ex.getMessage());

println(ex.getStackTrace());

} catch(Exception ex) {

println("Catching the exception");

}finally {

println("The final block");

}

println("Let's move on after the exception");

}

}

# Groovy – Object Oriented

In Groovy, as in any other Object-Oriented language, there is the concept of classes and objects to represent the objected oriented nature of the programming language. A Groovy class is a collection of data and the methods that operate on that data. Together, the data and methods of a class are used to represent some real world object from the problem domain.

A class in Groovy declares the state (data) and the behavior of objects defined by that class. Hence, a Groovy class describes both the instance fields and methods for that class.

Following is an example of a class in Groovy. The name of the class is Student which has two fields – **StudentID** and **StudentName**. In the main function, we are creating an object of this class and assigning values to the **StudentID** and **StudentName** of the object.

class Student {

int StudentID;

String StudentName;

static void main(String[] args) {

Student st = new Student();

st.StudentID = 1;

st.StudentName = "Joe"

}

}

## getter and setter Methods

In any programming language, it always a practice to hide the instance members with the private keyword and instead provide getter and setter methods to set and get the values of the instance variables accordingly. The following example shows how this can be done.

class Student {

private int StudentID;

private String StudentName;

void setStudentID(int pID) {

StudentID = pID;

}

void setStudentName(String pName) {

StudentName = pName;

}

int getStudentID() {

return this.StudentID;

}

String getStudentName() {

return this.StudentName;

}

static void main(String[] args) {

Student st = new Student();

st.setStudentID(1);

st.setStudentName("Joe");

println(st.getStudentID());

println(st.getStudentName());

}

}

When we run the above program, we will get the following result −

1

Joe

Note the following key points about the above program −

* In the class both the studentID and studentName are marked as private which means that they cannot be accessed from outside of the class.
* Each instance member has its own getter and setter method. The getter method returns the value of the instance variable, for example the method int getStudentID() and the setter method sets the value of the instance ID, for example the method - void setStudentName(String pName)

## Instance Methods

It’s normally a natural to include more methods inside of the class which actually does some sort of functionality for the class. In our student example, let’s add instance members of Marks1, Marks2 and Marks3 to denote the marks of the student in 3 subjects. We will then add a new instance method which will calculate the total marks of the student. Following is how the code would look like.

In the following example, the method Total is an additional Instance method which has some logic built into it.

class Student {

int StudentID;

String StudentName;

int Marks1;

int Marks2;

int Marks3;

int Total() {

return Marks1+Marks2+Marks3;

}

static void main(String[] args) {

Student st = new Student();

st.StudentID = 1;

st.StudentName="Joe";

st.Marks1 = 10;

st.Marks2 = 20;

st.Marks3 = 30;

println(st.Total());

}

}

When we run the above program, we will get the following result −

60

## Creating Multiple Objects

One can also create multiple objects of a class. Following is the example of how this can be achieved. In here we are creating 3 objects (st, st1 and st2) and calling their instance members and instance methods accordingly.

class Student {

int StudentID;

String StudentName;

int Marks1;

int Marks2;

int Marks3;

int Total() {

return Marks1+Marks2+Marks3;

}

static void main(String[] args) {

Student st = new Student();

st.StudentID = 1;

st.StudentName = "Joe";

st.Marks1 = 10;

st.Marks2 = 20;

st.Marks3 = 30;

println(st.Total());

Student st1 = new Student();

st.StudentID = 1;

st.StudentName = "Joe";

st.Marks1 = 10;

st.Marks2 = 20;

st.Marks3 = 40;

println(st.Total());

Student st3 = new Student();

st.StudentID = 1;

st.StudentName = "Joe";

st.Marks1 = 10;

st.Marks2 = 20;

st.Marks3 = 50;

println(st.Total());

}

}

When we run the above program, we will get the following result −

60

70

80

## Inheritance

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance, the information is made manageable in a hierarchical order.

The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

## Extends

**extends** is the keyword used to inherit the properties of a class. Given below is the syntax of extends keyword. In the following example we are doing the following things −

* Creating a class called Person. This class has one instance member called name.
* Creating a class called Student which extends from the Person class. Note that the name instance member which is defined in the Person class gets inherited in the Student class.
* In the Student class constructor, we are calling the base class constructor.
* In our Student class, we are adding 2 additional instance members of StudentID and Marks1.

class Example {

static void main(String[] args) {

Student st = new Student();

st.StudentID = 1;

st.Marks1 = 10;

st.name = "Joe";

println(st.name);

}

}

class Person {

public String name;

public Person() {}

}

class Student extends Person {

int StudentID

int Marks1;

public Student() {

super();

}

}

When we run the above program, we will get the following result −

Joe

## Inner Classes

Inner classes are defined within another classes. The enclosing class can use the inner class as usual. On the other side, an inner class can access members of its enclosing class, even if they are private. Classes other than the enclosing class are not allowed to access inner classes.

Following is an example of an Outer and Inner class. In the following example we are doing the following things −

* Creating a class called Outer which will be our outer class.
* Defining a string called name in our Outer class.
* Creating an Inner or nested class inside of our Outer class.
* Note that in the inner class we can access the name instance member defined in the Outer class.

class Example {

static void main(String[] args) {

Outer outobj = new Outer();

outobj.name = "Joe";

outobj.callInnerMethod()

}

}

class Outer {

String name;

def callInnerMethod() {

new Inner().methodA()

}

class Inner {

def methodA() {

println(name);

}

}

}

When we run the above program, we will get the following result −

Joe

## Abstract Classes

Abstract classes represent generic concepts, thus, they cannot be instantiated, being created to be subclassed. Their members include fields/properties and abstract or concrete methods. Abstract methods do not have implementation, and must be implemented by concrete subclasses. Abstract classes must be declared with abstract keyword. Abstract methods must also be declared with abstract keyword.

In the following example, note that the Person class is now made into an abstract class and cannot be instantiated. Also note that there is an abstract method called DisplayMarks in the abstract class which has no implementation details. In the student class it is mandatory to add the implementation details.

class Example {

static void main(String[] args) {

Student st = new Student();

st.StudentID = 1;

st.Marks1 = 10;

st.name="Joe";

println(st.name);

println(st.DisplayMarks());

}

}

abstract class Person {

public String name;

public Person() { }

abstract void DisplayMarks();

}

class Student extends Person {

int StudentID

int Marks1;

public Student() {

super();

}

void DisplayMarks() {

println(Marks1);

}

}

When we run the above program, we will get the following result −

Joe

10

null

## Interfaces

An interface defines a contract that a class needs to conform to. An interface only defines a list of methods that need to be implemented but does not define the methods implementation. An interface needs to be declared using the interface keyword. An interface only defines method signatures. Methods of an interface are always **public**. It is an error to use protected or private methods in interfaces.

Following is an example of an interface in groovy. In the following example we are doing the following things −

* Creating an interface called Marks and creating an interface method called DisplayMarks.
* In the class definition, we are using the implements keyword to implement the interface.
* Because we are implementing the interface we have to provide the implementation for the DisplayMarks method.

class Example {

static void main(String[] args) {

Student st = new Student();

st.StudentID = 1;

st.Marks1 = 10;

println(st.DisplayMarks());

}

}

interface Marks {

void DisplayMarks();

}

class Student implements Marks {

int StudentID

int Marks1;

void DisplayMarks() {

println(Marks1);

}

}

When we run the above program, we will get the following result −

10

null

# Groovy - Meta Object Programming

Meta object programming or MOP can be used to invoke methods dynamically and also create classes and methods on the fly.

So what does this mean? Let’s consider a class called Student, which is kind of an empty class with no member variables or methods. Suppose if you had to invoke the following statements on this class.

def myStudent = new Student()

myStudent.Name = ”Joe”;

myStudent.Display()

Now in meta object programming, even though the class does not have the member variable Name or the method Display(), the above code will still work.

How can this work? Well, for this to work out, one has to implement the GroovyInterceptable interface to hook into the execution process of Groovy. Following are the methods available for this interface.

public interface GroovyInterceptable {

public object invokeMethod(String methodName, Object args)

public object getproperty(String propertyName)

public object setProperty(String propertyName, Object newValue)

public MetaClass getMetaClass()

public void setMetaClass(MetaClass metaClass)

}

So in the above interface description, suppose if you had to implement the invokeMethod(), it would be called for every method which either exists or does not exist.

## Missing Properties

So let’s look an example of how we can implement Meta Object Programming for missing Properties. The following keys things should be noted about the following code.

* The class Student has no member variable called Name or ID defined.
* The class Student implements the GroovyInterceptable interface.
* There is a parameter called dynamicProps which will be used to hold the value of the member variables which are created on the fly.
* The methods getproperty and setproperty have been implemented to get and set the values of the property’s of the class at runtime.

class Example {

static void main(String[] args) {

Student mst = new Student();

mst.Name = "Joe";

mst.ID = 1;

println(mst.Name);

println(mst.ID);

}

}

class Student implements GroovyInterceptable {

protected dynamicProps=[:]

void setProperty(String pName,val) {

dynamicProps[pName] = val

}

def getProperty(String pName) {

dynamicProps[pName]

}

}

The output of the following code would be −

Joe

1

## Another Better Example for getProperty() and setProperty()

Here, we do not implement the GroovyInterceptable interface.

class Student {

def property1 = 'ha'

def field2 = 'ho'

def field4 = 'hu'

def getField1() {

return 'getHa'

}

def getProperty(String name) {

if (name != 'field3')

return metaClass.getProperty(this, name)

else

return 'field3'

}

String property;

void setProperty(String name, Object value) {

this.@"$name" = 'overridden'

}

}

class Example {

static void main(String[] args) {

Student mst = new Student();

println(mst.field1); // == 'getHa'

println(mst.field2); // == 'ho'

println(mst.field3); // == 'field3'

println(mst.field4); // == 'hu'

mst.property = "Hello";

println(mst.property); // == 'overridden'

}

}

## Missing methods

So let’s look an example of how we can implement Meta Object Programming for missing Properties. The following keys things should be noted about the following code −

* The class Student now implememts the invokeMethod method which gets called irrespective of whether the method exists or not.

class Example {

static void main(String[] args) {

Student mst = new Student();

mst.Name = "Joe";

mst.ID = 1;

println(mst.Name);

println(mst.ID);

println(mst.AddMarks());

}

}

class Student implements GroovyInterceptable {

protected dynamicProps = [:]

void setProperty(String pName, val) {

dynamicProps[pName] = val

}

def getProperty(String pName) {

dynamicProps[pName]

}

def invokeMethod(String name, Object args) {

return "called invokeMethod $name $args"

}

}

The output of the following codewould be shown below. Note that there is no error of missing Method Exception even though the method Display does not exist.

Joe

1

called invokeMethod Addmarks []

However, the use of invokeMethod to intercept missing methods is discouraged. In cases where the intent is to only intercept method calls in the case of a failed method dispatch use methodMissing instead.

## Method Missing

Groovy supports the concept of methodMissing. This method differs from invokeMethod in that it is only invoked in case of a failed method dispatch, when no method can be found for the given name and/or the given arguments. The following example shows how the methodMissing can be used.

class Example {

static void main(String[] args) {

Student mst = new Student();

mst.Name = "Joe";

mst.ID = 1;

println(mst.Name);

println(mst.ID);

mst.AddMarks();

mst.AddMarks("ABC");

}

}

class Student implements GroovyInterceptable {

protected dynamicProps = [:]

void setProperty(String pName, val) {

dynamicProps[pName] = val

}

def getProperty(String pName) {

dynamicProps[pName]

}

def methodMissing(String name, def args) {

println "Missing method"

}

}

The output of the following code would be −

Joe

1

Missing method AddMarks []

Missing method AddMarks [ABC]

## Metaclass

This functionality is related to the MetaClass implementation. In the default implementation you can access fields without invoking their getters and setters. The following example shows how by using the metaClass function we are able to change and retrieve the value of the private variables in the class.

class Example {

static void main(String[] args) {

Student mst = new Student();

println mst.getName()

mst.metaClass.setAttribute(mst, 'name', 'Mark')

println mst.getName()

println(mst.metaClass.getAttribute(mst, 'name'))

}

}

class Student {

private String name = "Joe";

public String getName() {

return this.name;

}

}

The output of the following code would be −

Joe

Mark

Mark

# Groovy – Closures

A closure is a short anonymous block of code. It just normally spans a few lines of code. A method can even take the block of code as a parameter. They are anonymous in nature.

Following is an example of a simple closure and what it looks like.

class Example {

static void main(String[] args) {

def clos = {println "Hello World"};

clos.call();

}

}

In the above example, the code line - {println "Hello World"} is known as a closure. The code block referenced by this identifier can be executed with the call statement.

When we run the above program, we will get the following result −

Hello World

## Formal parameters in closures

Closures can also contain formal parameters to make them more useful just like methods in Groovy.

class Example {

static void main(String[] args) {

def clos = {param->println "Hello ${param}"};

clos.call("World");

}

}

In the above code example, notice the use of the ${param } which causes the closure to take a parameter. When calling the closure via the clos.call statement we now have the option to pass a parameter to the closure.

When we run the above program, we will get the following result −

Hello World

The next illustration repeats the previous example and produces the same result, but shows that an implicit single parameter referred to as it can be used. Here ‘it’ is a keyword in Groovy.

class Example {

static void main(String[] args) {

def clos = {println "Hello ${it}"};

clos.call("World");

}

}

When we run the above program, we will get the following result −

Hello World

## Closures and Variables

More formally, closures can refer to variables at the time the closure is defined. Following is an example of how this can be achieved.

class Example {

static void main(String[] args) {

def str1 = "Hello";

def clos = {param -> println "${str1} ${param}"}

clos.call("World");

// We are now changing the value of the String str1 which is referenced in the closure

str1 = "Welcome";

clos.call("World");

}

}

In the above example, in addition to passing a parameter to the closure, we are also defining a variable called str1. The closure also takes on the variable along with the parameter.

When we run the above program, we will get the following result −

Hello World

Welcome World

## Using Closures in Methods

Closures can also be used as parameters to methods. In Groovy, a lot of the inbuilt methods for data types such as Lists and collections have closures as a parameter type.

The following example shows how a closure can be sent to a method as a parameter.

class Example {

def static Display(clo) {

// This time the $param parameter gets replaced by the string "Inner"

clo.call("Inner");

}

static void main(String[] args) {

def str1 = "Hello";

def clos = { param -> println "${str1} ${param}" }

clos.call("World");

// We are now changing the value of the String str1 which is referenced in the closure

str1 = "Welcome";

clos.call("World");

// Passing our closure to a method

Example.Display(clos);

}

}

In the above example,

* We are defining a static method called Display which takes a closure as an argument.
* We are then defining a closure in our main method and passing it to our Display method as a parameter.

When we run the above program, we will get the following result −

Hello World

Welcome World

Welcome Inner

## Closures in Collections and String

Several List, Map, and String methods accept a closure as an argument. Let’s look at example of how closures can be used in these data types.

### Using Closures with Lists

The following example shows how closures can be used with Lists. In the following example we are first defining a simple list of values. The list collection type then defines a function called .**each**. This function takes on a closure as a parameter and applies the closure to each element of the list.

class Example {

static void main(String[] args) {

def lst = [11, 12, 13, 14];

lst.each {println it}

}

}

When we run the above program, we will get the following result −

11

12

13

14

### Using Closures with Maps

The following example shows how closures can be used with Maps. In the following example we are first defining a simple Map of key value items. The map collection type then defines a function called .each. This function takes on a closure as a parameter and applies the closure to each key-value pair of the map.

class Example {

static void main(String[] args) {

def mp = ["TopicName" : "Maps", "TopicDescription" : "Methods in Maps"]

mp.each {println it}

mp.each {println "${it.key} maps to: ${it.value}"}

}

}

When we run the above program, we will get the following result −

TopicName = Maps

TopicDescription = Methods in Maps

TopicName maps to: Maps

TopicDescription maps to: Methods in Maps

Often, we may wish to iterate across the members of a collection and apply some logic only when the element meets some criterion. This is readily handled with a conditional statement in the closure.

class Example {

static void main(String[] args) {

def lst = [1,2,3,4];

lst.each {println it}

println("The list will only display those numbers which are divisible by 2")

lst.each{num -> if(num % 2 == 0) println num}

}

}

The above example shows the conditional if(num % 2 == 0) expression being used in the closure which is used to check if each item in the list is divisible by 2.

When we run the above program, we will get the following result −

1

2

3

4

The list will only display those numbers which are divisible by 2.

2

4

## Methods used with Closures

The closures themselves provide some methods.

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | [find()](https://www.tutorialspoint.com/groovy/groovy_find.htm) The find method finds the first value in a collection that matches some criterion.  Object find(Closure closure)  class Example {  static void main(String[] args) {  def lst = [1,2,3,4];  def value;    value = lst.find {element -> element > 2}  println(value);  }  } |
| 2 | [findAll()](https://www.tutorialspoint.com/groovy/groovy_findall.htm) It finds all values in the receiving object matching the closure condition.  List findAll(Closure closure)  class Example {  static void main(String[] args) {  def lst = [1,2,3,4];  def value;    value = lst.findAll{element -> element > 2}  value.each {println it}  }  } |
| 3 | [any() & every()](https://www.tutorialspoint.com/groovy/groovy_any_every.htm) Method any iterates through each element of a collection checking whether a Boolean predicate is valid for at least one element.  boolean any(Closure closure)  boolean every(Closure closure)  class Example {  static void main(String[] args) {  def lst = [1,2,3,4];  def value;    // Is there any value above 2  value = lst.any{element -> element > 2}  println(value);    // Is there any value above 4  value = lst.any{element -> element > 4}  println(value);  }  }  class Example {  static void main(String[] args) {  def lst = [1,2,3,4];  def value;    // Are all value above 2  value = lst.every{element -> element > 2}  println(value);    // Are all value above 4  value = lst.every{element -> element > 4}  println(value);  def largelst = [4,5,6];    // Are all value above 2  value = largelst.every{element -> element > 2}  println(value);  }  } |
| 4 | [collect()](https://www.tutorialspoint.com/groovy/groovy_collect.htm) The method collect iterates through a collection, converting each element into a new value using the closure as the transformer.  List collect(Closure closure)  class Example {  static void main(String[] args) {  def lst = [1,2,3,4];  def newlst = [];  newlst = lst.collect {element -> return element \* element}  println(newlst);  }  } |

# Groovy – Builder Pattern

<https://dzone.com/articles/a-basic-builder-with-groovy>

A builder, in its most simple form, can be thought of as a hierarchically nested constructor which may take a closure as argument. Within the closure, further constructors may be called.

You, as implementer of a builder, decide what the constructors do, how they're called and which arguments they take. Normally, you will already have an existing data structure that you would like to construct or modify with builder syntax. These data structures are, for example, an XML document, a Swing UI or a JSON document. In these cases, Groovy provides builders out-of-the-box. Maybe you have a domain model, or a very specific document or message type that you would like to create and you want something more specialized. You need a domain specific language and the Groovy Builder syntax is just what you're looking for.

Here, we'll create a builder for a domain model consisting of a Company with Departments and Employees. A Department may have zero or more Departments. Each object has an ID that is passed through the constructor.

## Domain Model

The domain model is as follows:

### Company

|  |
| --- |
| // Company.groovy  package builder.basic.model; |
| import java.util.ArrayList; |
| import java.util.List; |
|  |
| public class Company { |
| private List<Department> departments; |
| private final String id; |
|  |
| public Company(String id) { |
| this.id = id; |
| this.departments = new ArrayList<>(); |
| } |
| public List<Department> getDepartments() { |
| return departments; |
| } |
| public void setDepartments(List<Department> departments) { |
| this.departments = departments; |
| } |
| public String getId() { |
| return id; |
| } |
| } |

### Department

|  |
| --- |
| // Department.groovy  package builder.basic.model; |
| import java.util.ArrayList; |
| import java.util.List; |
|  |
| public class Department { |
| private List<Employee> employees; |
| private List<Department> departments; |
| private final String id; |
|  |
| public Department(String id) { |
| this.id = id; |
| this.employees = new ArrayList<>(); |
| this.departments = new ArrayList<>(); |
| } |
|  |
| public List<Employee> getEmployees() { |
| return employees; |
| } |
|  |
| public void setEmployees(List<Employee> employees) { |
| this.employees = employees; |
| } |
|  |
| public List<Department> getDepartments() { |
| return departments; |
| } |
|  |
| public void setDepartments(List<Department> departments) { |
| this.departments = departments; |
| } |
|  |
| public String getId() { |
| return id; |
| } |
| } |

### Employee

|  |
| --- |
| // Employee.groovy  package builder.basic.model; |
| public class Employee { |
| private String name; |
| private String role; |
| private final String id; |
|  |
| public Employee(String id) { |
| this.id = id; |
| } |
|  |
| public String getName() { |
| return name; |
| } |
| public void setName(String name) { |
| this.name = name; |
| } |
| public String getRole() { |
| return role; |
| } |
| public void setRole(String role) { |
| this.role = role; |
| } |
|  |
| public String getId() { |
| return id; |
| } |
| } |

## CompanyBuilder Implementation

|  |
| --- |
| // CompanyBuilder.groovy  package builder.basic |
| import builder.basic.model.Company |
| import builder.basic.model.Department |
| import builder.basic.model.Employee |
|  |
| class CompanyBuilder extends BuilderSupport { |
| @Override |
| protected Object createNode(Object name, Object id) { |
| switch(name) { |
| case 'company': return createCompany(id) |
| case 'department': return createDepartment(id) |
| case 'employee': return createEmployee(id) |
| case 'name': return setEmployeeName(id) |
| case 'role': return setEmployeeRole(id) |
| default: throw new IllegalArgumentException("Invalid keyword $name") |
| } |
| } |
| @Override |
| protected Object createNode(Object arg0) { |
| // TODO Auto-generated method stub |
| return null; |
| } |
| @Override |
| protected Object createNode(Object arg0, Map arg1) { |
| // TODO Auto-generated method stub |
| return null; |
| } |
| @Override |
| protected Object createNode(Object arg0, Map arg1, Object arg2) { |
| // TODO Auto-generated method stub |
| return null; |
| } |
| @Override |
| protected void setParent(Object parent, Object child) { |
|  |
| } |
| private Employee setEmployeeName(String name) { |
| if(current instanceof Employee) { |
| Employee employee = (Employee) current |
| employee.setName(name) |
| } else { |
| throw new IllegalArgumentException("Invalid keyword 'name'") |
| } |
| } |
| private Employee setEmployeeRole(String role) { |
| if(current instanceof Employee) { |
| Employee employee = (Employee) current |
| employee.setRole(role) |
| } else { |
| throw new IllegalArgumentException("Invalid keyword 'role'") |
| } |
| } |
| private Company createCompany(String id) { |
| Company company = new Company(id) |
| return company |
| } |
| private Department createDepartment(String id) { |
| Department department = new Department(id) |
| if(current instanceof Company) { |
| Company company = (Company) current |
| company.getDepartments().add(department) |
| } else if(current instanceof Department) { |
| Department parentDep = (Department) current |
| parentDep.getDepartments().add(department) |
| } |
| return department |
| } |
| private Employee createEmployee(String id) { |
| Employee employee = new Employee(id) |
| if(current instanceof Department) { |
| Department department = (Department) current |
| department.getEmployees().add(employee) |
| } |
| return employee |
| } |
| } |

## CompanyBuilder Usage / Syntax

// BuilderExample.groovy

package builder.basic

import builder.basic.CompanyBuilder

import builder.basic.model.Company

import builder.basic.model.Department

import builder.basic.model.Employee

class BuilderExample {

static void main(String[] args) {

CompanyBuilder builder = new CompanyBuilder()

Company company = builder.company('ABC') {

department('XYZ') {

employee('emp12345') {

name('John')

role('Administrator')

}

}

department('123') {

employee('emp987') {

name('Karen')

role('Project Manager')

}

department('456') {

employee('emp456') {

name('Mary')

role('Developer')

}

}

}

}

println("Comapny Id: ${company.getId()}")

println(company.getDepartments().each { dept -> println("Department Id: ${dept.getId()}") })

println(company.getDepartments().each { dept ->

println("Department Id: ${dept.getId()}");

dept.getEmployees().each {

emp -> println("\tEmployee Id: ${emp.getId()}. Name: ${emp.getName()}. Role: ${emp.getRole()}")

};

dept.getDepartments().each {

dpt -> println("\tDept Id: ${dpt.getId()}");

dpt.getEmployees().each {e -> println("\t\tEmployee Id: ${e.getId()}. Name: ${e.getName()}. Role: ${e.getRole()}") }

}

})

}

}

# References

<https://www.tutorialspoint.com/groovy/index.htm>