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 −

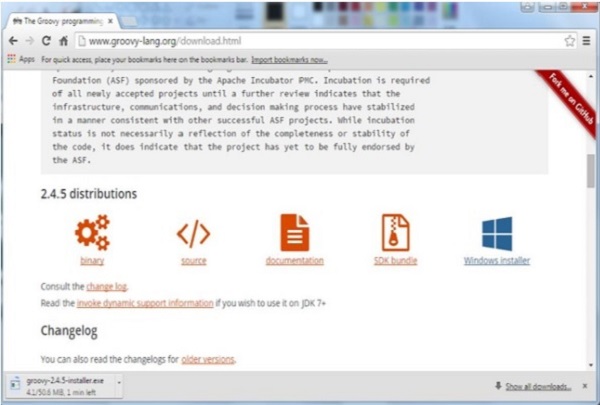
* Support for both static and dynamic typing.
* Support for operator overloading.
* Native syntax for lists and associative arrays.
* Native support for regular expressions.
* Native support for various markup languages such as XML and HTML.
* Groovy is simple for Java developers since the syntax for 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/>



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. Click on this option to start the download of the Groovy installer.



Once you launch the installer, follow the steps given below to complete the installation.

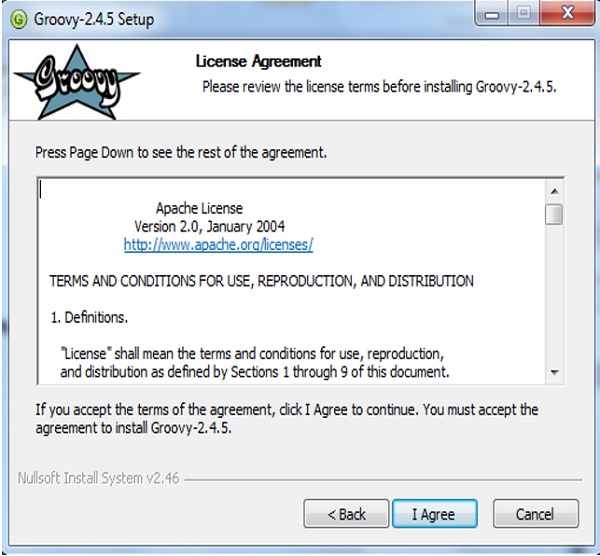
**Step 1** − Select the language installer.



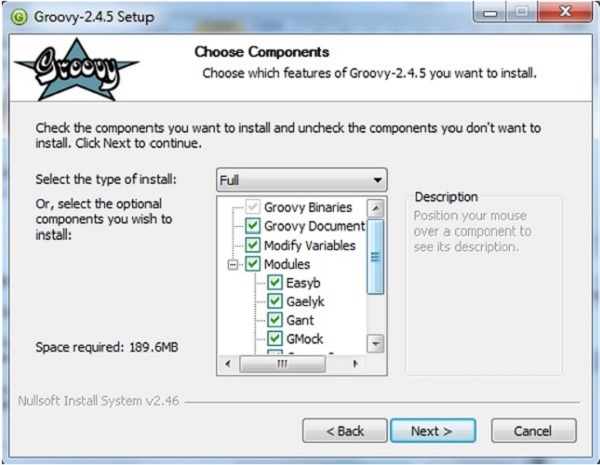
**Step 2** − Click the Next button in the next screen.



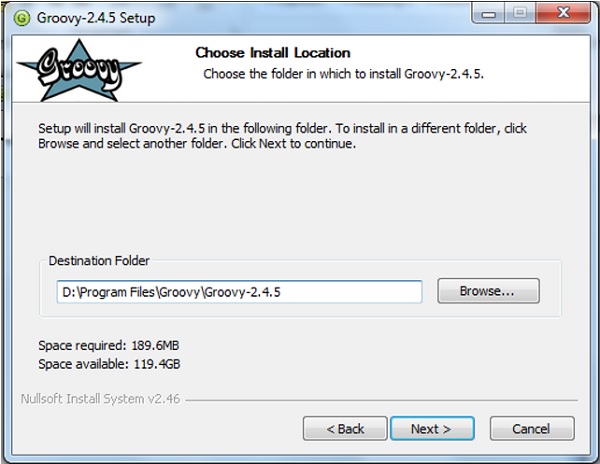
**Step 3** − Click the ‘I Agree’ button.



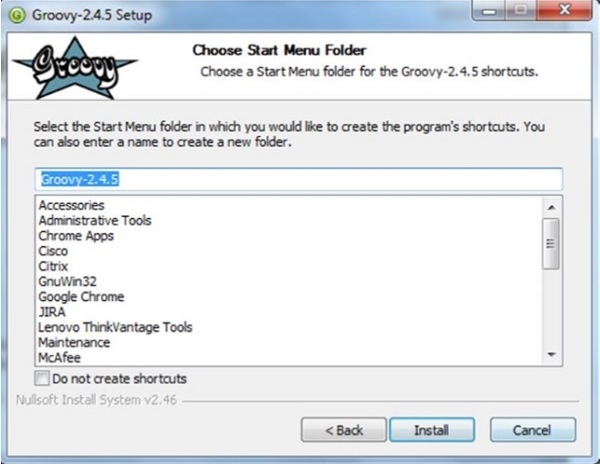
**Step 4** − Accept the default components and click the Next button.



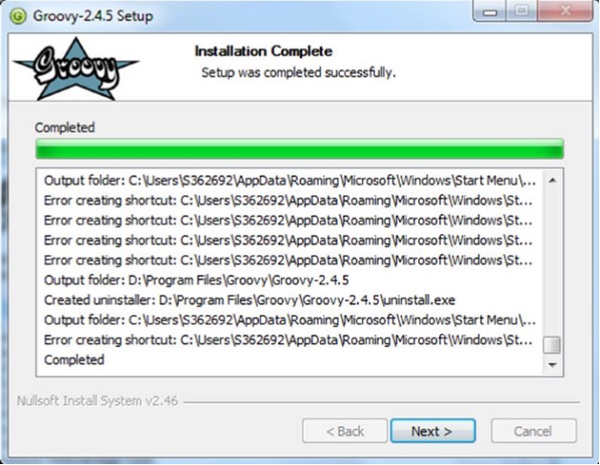
**Step 5** − Choose the appropriate destination folder and then click the Next button.



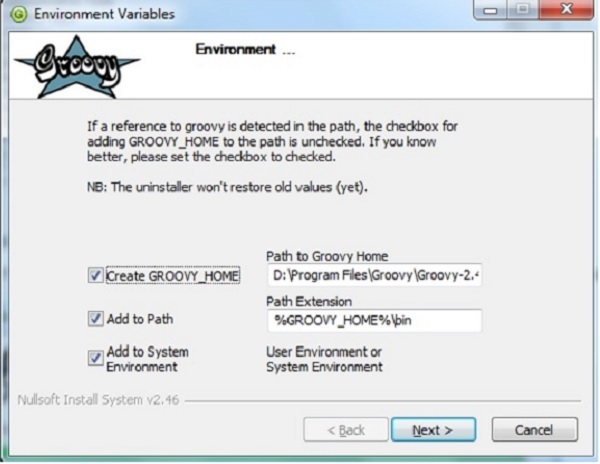
**Step 6** − Click the Install button to start the installation.



**Step 7** − Once the installation is complete, click the Next button to start the configuration.



**Step 8** − Choose the default options and click the Next button.



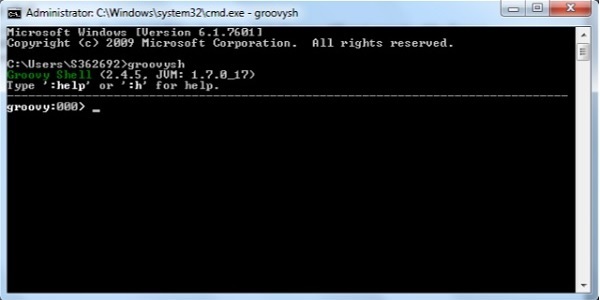
**Step 9** − Accept the default file associations and click the Next button.



**Step 10** − Click the Finish button to complete the installation.



Once the above steps are followed, you can then start the groovy shell which is part of the Groovy installation that helps in testing our different aspects of the Groovy language without the need of having a full-fledged integrated development environment for Groovy. This can be done by running the command groovysh from the command prompt.



In order to understand the basic syntax of Groovy, let’s first look at a simple Hello World program.

## Creating Your First Hello World Program

Creating your first hello world program is as simple as just entering the following code line −

class Example {

static void main(String[] args) {

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

println('Hello World');

}

}

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

Hello World

## 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”

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 −

[Live Demo](http://tpcg.io/iohPBC)

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

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 −

[Live Demo](http://tpcg.io/BQfpaa)

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.

[Live Demo](http://tpcg.io/C1ltlR)

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.

[Live Demo](http://tpcg.io/WK7YA7)

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

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 −

[Show Example](https://www.tutorialspoint.com/groovy/groovy_arithmetic_operators.htm)

|  |  |  |
| --- | --- | --- |
| **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 |

Relational operators

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

[Show Example](https://www.tutorialspoint.com/groovy/groovy_relational_operators.htm)

|  |  |  |
| --- | --- | --- |
| **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 |

Logical Operators

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

[Show Example](https://www.tutorialspoint.com/groovy/groovy_logical_operators.htm)

|  |  |  |
| --- | --- | --- |
| **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 |

Bitwise Operators

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

[Show Example](https://www.tutorialspoint.com/groovy/groovy_bitwise_operators.htm)

|  |  |
| --- | --- |
| **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 |

Assignment operators

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

[Show Example](https://www.tutorialspoint.com/groovy/groovy_assignment_operators.htm)

|  |  |  |
| --- | --- | --- |
| **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 |

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.

[Live Demo](http://tpcg.io/nHfO5r)

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 |

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](https://www.tutorialspoint.com/groovy/groovy_while_statement.htm)  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](https://www.tutorialspoint.com/groovy/groovy_for_statement.htm)  The for statement is used to iterate through a set of values. |
| 3 | [for-in Statement](https://www.tutorialspoint.com/groovy/groovy_forin_statement.htm)  The for-in statement is used to iterate through a set of values. |

## Loop Control Statements

|  |  |
| --- | --- |
| **S.No.** | **Statements & Description** |
| 1 | [Break Statement](https://www.tutorialspoint.com/groovy/groovy_break_statement.htm)  The break statement is used to alter the flow of control inside loops and switch statements. |
| 2 | [Continue Statement](https://www.tutorialspoint.com/groovy/groovy_continue_statement.htm)  The continue statement complements the break statement. Its use is restricted to while and for loops. |

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](https://www.tutorialspoint.com/groovy/groovy_if_statement.htm)  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](https://www.tutorialspoint.com/groovy/groovy_if_else_statement.htm)  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](https://www.tutorialspoint.com/groovy/groovy_nested_if_statement.htm)  Sometimes there is a requirement to have multiple if statement embedded inside of each other. |
| 4 | [Switch Statement](https://www.tutorialspoint.com/groovy/groovy_switch_statement.htm)  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](https://www.tutorialspoint.com/groovy/groovy_nested_switch_statement.htm)  It is also possible to have a nested set of switch statements. |

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.

[Live Demo](http://tpcg.io/iWjJ2h)

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 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 −

[Live Demo](http://tpcg.io/5BmZfv)

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.

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);

}

}

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 −

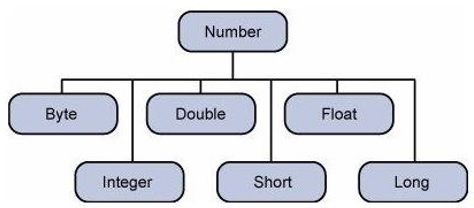
Integer x = 5;

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.

Number Methods

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

|  |  |
| --- | --- |
| **S.No.** | **Methods & Description** |
| 1 | [xxxValue()](https://www.tutorialspoint.com/groovy/groovy_xxxvalue.htm)  This method takes on the Number as the parameter and returns a primitive type based on the method which is invoked. |
| 2 | [compareTo()](https://www.tutorialspoint.com/groovy/groovy_compareto.htm)  The compareTo method is to use compare one number against another. This is useful if you want to compare the value of numbers. |
| 3 | [equals()](https://www.tutorialspoint.com/groovy/groovy_equals.htm)  The method determines whether the Number object that invokes the method is equal to the object that is passed as argument. |
| 4 | [valueOf()](https://www.tutorialspoint.com/groovy/groovy_valueof.htm)  The valueOf method returns the relevant Number Object holding the value of the argument passed. |
| 5 | [toString()](https://www.tutorialspoint.com/groovy/groovy_tostring.htm)  The method is used to get a String object representing the value of the Number Object. |
| 6 | [parseInt()](https://www.tutorialspoint.com/groovy/groovy_parseint.htm)  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. |
| 7 | [abs()](https://www.tutorialspoint.com/groovy/groovy_abs.htm)  The method gives the absolute value of the argument. The argument can be int, float, long, double, short, byte. |
| 8 | [ceil()](https://www.tutorialspoint.com/groovy/groovy_ceil.htm)  The method ceil gives the smallest integer that is greater than or equal to the argument. |
| 9 | [floor()](https://www.tutorialspoint.com/groovy/groovy_floor.htm)  The method floor gives the largest integer that is less than or equal to the argument. |
| 10 | [rint()](https://www.tutorialspoint.com/groovy/groovy_rint.htm)  The method rint returns the integer that is closest in value to the argument. |
| 11 | [round()](https://www.tutorialspoint.com/groovy/groovy_round.htm)  The method round returns the closest long or int, as given by the methods return type. |
| 12 | [min()](https://www.tutorialspoint.com/groovy/groovy_min.htm)  The method gives the smaller of the two arguments. The argument can be int, float, long, double. |
| 13 | [max()](https://www.tutorialspoint.com/groovy/groovy_max.htm)  The method gives the maximum of the two arguments. The argument can be int, float, long, double. |
| 14 | [exp()](https://www.tutorialspoint.com/groovy/groovy_exp.htm)  The method returns the base of the natural logarithms, e, to the power of the argument. |
| 15 | [log()](https://www.tutorialspoint.com/groovy/groovy_log.htm)  The method returns the natural logarithm of the argument. |
| 16 | [pow()](https://www.tutorialspoint.com/groovy/groovy_pow.htm)  The method returns the value of the first argument raised to the power of the second argument. |
| 17 | [sqrt()](https://www.tutorialspoint.com/groovy/groovy_sqrt.htm)  The method returns the square root of the argument. |
| 18 | [sin()](https://www.tutorialspoint.com/groovy/groovy_sin.htm)  The method returns the sine of the specified double value. |
| 19 | [cos()](https://www.tutorialspoint.com/groovy/groovy_cos.htm)  The method returns the cosine of the specified double value. |
| 20 | [tan()](https://www.tutorialspoint.com/groovy/groovy_tan.htm)  The method returns the tangent of the specified double value. |
| 21 | [asin()](https://www.tutorialspoint.com/groovy/groovy_asin.htm)  The method returns the arcsine of the specified double value. |
| 22 | [acos()](https://www.tutorialspoint.com/groovy/groovy_acos.htm)  The method returns the arccosine of the specified double value. |
| 23 | [atan()](https://www.tutorialspoint.com/groovy/groovy_atan.htm)  The method returns the arctangent of the specified double value. |
| 24 | [atan2()](https://www.tutorialspoint.com/groovy/groovy_atan2.htm)  The method Converts rectangular coordinates (x, y) to polar coordinate (r, theta) and returns theta. |
| 25 | [toDegrees()](https://www.tutorialspoint.com/groovy/groovy_numbers_todegrees.htm)  The method converts the argument value to degrees. |
| 26 | [radian()](https://www.tutorialspoint.com/groovy/groovy_radian.htm)  The method converts the argument value to radians. |
| 27 | [random()](https://www.tutorialspoint.com/groovy/groovy_random.htm)  The method is used to generate a random number between 0.0 and 1.0. The range is: 0.0 =< Math.random < 1.0. Different ranges can be achieved by using arithmetic. |

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 −

[Live Demo](http://tpcg.io/LOkkzU)

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 −

[Live Demo](http://tpcg.io/Y0Okru)

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

First let’s learn the basic string operations in groovy. They are given below.

|  |  |
| --- | --- |
| **S.No.** | **String Operation & Description** |
| 1 | [Concatenation of two strings](https://www.tutorialspoint.com/groovy/groovy_concatenation_of_two_strings.htm)  The concatenation of strings can be done by the simple ‘+’ operator. |
| 2 | [String Repetition](https://www.tutorialspoint.com/groovy/groovy_string_repetition.htm)  The repetition of strings can be done by the simple ‘\*’ operator. |
| 3 | [String Length](https://www.tutorialspoint.com/groovy/groovy_string_length.htm)  The length of the string determined by the length() method of the string. |

## String Methods

Here is the list of methods supported by String class.

|  |  |
| --- | --- |
| **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. |

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 −

[Live Demo](http://tpcg.io/LOkkzU)

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 −

[Live Demo](http://tpcg.io/Y0Okru)

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

First let’s learn the basic string operations in groovy. They are given below.

|  |  |
| --- | --- |
| **S.No.** | **String Operation & Description** |
| 1 | [Concatenation of two strings](https://www.tutorialspoint.com/groovy/groovy_concatenation_of_two_strings.htm)  The concatenation of strings can be done by the simple ‘+’ operator. |
| 2 | [String Repetition](https://www.tutorialspoint.com/groovy/groovy_string_repetition.htm)  The repetition of strings can be done by the simple ‘\*’ operator. |
| 3 | [String Length](https://www.tutorialspoint.com/groovy/groovy_string_length.htm)  The length of the string determined by the length() method of the string. |

## String Methods

Here is the list of methods supported by String class.

|  |  |
| --- | --- |
| **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. |

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.

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | [contains()](https://www.tutorialspoint.com/groovy/groovy_contains.htm)  Checks if a range contains a specific value |
| 2 | [get()](https://www.tutorialspoint.com/groovy/groovy_get.htm)  Returns the element at the specified position in this Range. |
| 3 | [getFrom()](https://www.tutorialspoint.com/groovy/groovy_getfrom.htm)  Get the lower value of this Range. |
| 4 | [getTo()](https://www.tutorialspoint.com/groovy/groovy_getto.htm)  Get the upper value of this Range. |
| 5 | [isReverse()](https://www.tutorialspoint.com/groovy/groovy_isreverse.htm)  Is this a reversed Range, iterating backwards |
| 6 | [size()](https://www.tutorialspoint.com/groovy/groovy_size.htm)  Returns the number of elements in this Range. |
| 7 | [subList()](https://www.tutorialspoint.com/groovy/groovy_sublist.htm)  Returns a view of the portion of this Range between the specified fromIndex, inclusive, and toIndex, exclusive |

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

In this chapter, we will discuss the list methods available in Groovy.

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | [add()](https://www.tutorialspoint.com/groovy/groovy_add.htm)  Append the new value to the end of this List. |
| 2 | [contains()](https://www.tutorialspoint.com/groovy/groovy_lists_contains.htm)  Returns true if this List contains the specified value. |
| 3 | [get()](https://www.tutorialspoint.com/groovy/groovy_lists_get.htm)  Returns the element at the specified position in this List. |
| 4 | [isEmpty()](https://www.tutorialspoint.com/groovy/groovy_isempty.htm)  Returns true if this List contains no elements |
| 5 | [minus()](https://www.tutorialspoint.com/groovy/groovy_lists_minus.htm)  Creates a new List composed of the elements of the original without those specified in the collection. |
| 6 | [plus()](https://www.tutorialspoint.com/groovy/groovy_lists_plus.htm)  Creates a new List composed of the elements of the original together with those specified in the collection. |
| 7 | [pop()](https://www.tutorialspoint.com/groovy/groovy_pop.htm)  Removes the last item from this List |
| 8 | [remove()](https://www.tutorialspoint.com/groovy/groovy_remove.htm)  Removes the element at the specified position in this List. |
| 9 | [reverse()](https://www.tutorialspoint.com/groovy/groovy_reverse.htm)  Create a new List that is the reverse the elements of the original List |
| 10 | [size()](https://www.tutorialspoint.com/groovy/groovy_lists_size.htm)  Obtains the number of elements in this List. |
| 11 | [sort()](https://www.tutorialspoint.com/groovy/groovy_sort.htm)  Returns a sorted copy of the original List. |

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

In this chapter, we will discuss the map methods available in Groovy.

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | [containsKey()](https://www.tutorialspoint.com/groovy/groovy_containskey.htm)  Does this Map contain this key? |
| 2 | [get()](https://www.tutorialspoint.com/groovy/groovy_maps_get.htm)  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. |
| 3 | [keySet()](https://www.tutorialspoint.com/groovy/groovy_keyset.htm)  Obtain a Set of the keys in this Map. |
| 4 | [put()](https://www.tutorialspoint.com/groovy/groovy_put.htm)  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. |
| 5 | [size()](https://www.tutorialspoint.com/groovy/groovy_maps_size.htm)  Returns the number of key-value mappings in this Map. |
| 6 | [values()](https://www.tutorialspoint.com/groovy/groovy_values.htm)  Returns a collection view of the values contained in this Map. |

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.

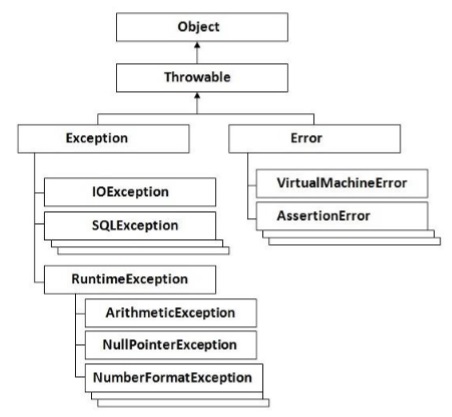
Caught: java.lang.ArrayIndexOutOfBoundsException: 5

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.

[Live Demo](http://tpcg.io/9tQnCw)

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.

[Live Demo](http://tpcg.io/NFEbAz)

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.

### **Example**

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

[Live Demo](http://tpcg.io/NAzQHf)

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");

}

}

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

java.lang.ArrayIndexOutOfBoundsException: 5

5

[org.codehaus.groovy.runtime.dgmimpl.arrays.IntegerArrayPutAtMetaMethod$MyPojoMetaMet

hodSite.call(IntegerArrayPutAtMetaMethod.java:75),

org.codehaus.groovy.runtime.callsite.CallSiteArray.defaultCall(CallSiteArray.java:48) ,

org.codehaus.groovy.runtime.callsite.AbstractCallSite.call(AbstractCallSite.java:113) ,

org.codehaus.groovy.runtime.callsite.AbstractCallSite.call(AbstractCallSite.java:133) ,

Example.main(Sample:8), sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method),

sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57),

sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43) ,

java.lang.reflect.Method.invoke(Method.java:606),

org.codehaus.groovy.reflection.CachedMethod.invoke(CachedMethod.java:93),

groovy.lang.MetaMethod.doMethodInvoke(MetaMethod.java:325),

groovy.lang.MetaClassImpl.invokeStaticMethod(MetaClassImpl.java:1443),

org.codehaus.groovy.runtime.InvokerHelper.invokeMethod(InvokerHelper.java:893),

groovy.lang.GroovyShell.runScriptOrMainOrTestOrRunnable(GroovyShell.java:287),

groovy.lang.GroovyShell.run(GroovyShell.java:524),

groovy.lang.GroovyShell.run(GroovyShell.java:513),

groovy.ui.GroovyMain.processOnce(GroovyMain.java:652),

groovy.ui.GroovyMain.run(GroovyMain.java:384),

groovy.ui.GroovyMain.process(GroovyMain.java:370),

groovy.ui.GroovyMain.processArgs(GroovyMain.java:129),

groovy.ui.GroovyMain.main(GroovyMain.java:109),

sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method),

sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57),

sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43) ,

java.lang.reflect.Method.invoke(Method.java:606),

org.codehaus.groovy.tools.GroovyStarter.rootLoader(GroovyStarter.java:109),

org.codehaus.groovy.tools.GroovyStarter.main(GroovyStarter.java:131),

sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method),

sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57),

sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43) ,

java.lang.reflect.Method.invoke(Method.java:606),

com.intellij.rt.execution.application.AppMain.main(AppMain.java:144)]

The final block

Let's move on after the exception

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.

[Live Demo](http://tpcg.io/ewYXBx)

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.

[Live Demo](http://tpcg.io/rin2jc)

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.

[Live Demo](http://tpcg.io/xBccLF)

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, a 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 an 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 are able to access the name instance member defined in the Outer class.

[Live Demo](http://tpcg.io/u0tKZh)

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.

[Live Demo](http://tpcg.io/VFroxe)

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.

[Live Demo](http://tpcg.io/KFhxDg)

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

Traits are a structural construct of the language which allow −

* Composition of behaviors.
* Runtime implementation of interfaces.
* Compatibility with static type checking/compilation

They can be seen as interfaces carrying both default implementations and state. A trait is defined using the trait keyword.

An example of a trait is given below −

trait Marks {

void DisplayMarks() {

println("Display Marks");

}

}

One can then use the implement keyword to implement the trait in the similar way as interfaces.

class Example {

static void main(String[] args) {

Student st = new Student();

st.StudentID = 1;

st.Marks1 = 10;

println(st.DisplayMarks());

}

}

trait Marks {

void DisplayMarks() {

println("Display Marks");

}

}

class Student implements Marks {

int StudentID

int Marks1;

}

Implementing Interfaces

Traits may implement interfaces, in which case the interfaces are declared using the implements keyword.

An example of a trait implementing an interface is given below. In the following example the following key points can be noted.

* An interface Total is defined with the method DisplayTotal.
* The trait Marks implements the Total interface and hence needs to provide an implementation for the DisplayTotal method.

[Live Demo](http://tpcg.io/217WUe)

class Example {

static void main(String[] args) {

Student st = new Student();

st.StudentID = 1;

st.Marks1 = 10;

println(st.DisplayMarks());

println(st.DisplayTotal());

}

}

interface Total {

void DisplayTotal()

}

trait Marks implements Total {

void DisplayMarks() {

println("Display Marks");

}

void DisplayTotal() {

println("Display Total");

}

}

class Student implements Marks {

int StudentID

int Marks1;

}

The output of the above program would be −

Display Marks

Display Total

Properties

A trait may define properties. An example of a trait with a property is given below.

In the following example, the Marks1 of type integer is a property.

class Example {

static void main(String[] args) {

Student st = new Student();

st.StudentID = 1;

println(st.DisplayMarks());

println(st.DisplayTotal());

}

interface Total {

void DisplayTotal()

}

trait Marks implements Total {

int Marks1;

void DisplayMarks() {

this.Marks1 = 10;

println(this.Marks1);

}

void DisplayTotal() {

println("Display Total");

}

}

class Student implements Marks {

int StudentID

}

}

The output of the above program would be −

10

Display Total

Composition of Behaviors

Traits can be used to implement multiple inheritance in a controlled way, avoiding the diamond issue. In the following code example, we have defined two traits – **Marks** and **Total**. Our Student class implements both traits. Since the student class extends both traits, it is able to access the both of the methods – **DisplayMarks** and **DisplayTotal**.

[Live Demo](http://tpcg.io/2sTaFN)

class Example {

static void main(String[] args) {

Student st = new Student();

st.StudentID = 1;

println(st.DisplayMarks());

println(st.DisplayTotal());

}

}

trait Marks {

void DisplayMarks() {

println("Marks1");

}

}

trait Total {

void DisplayTotal() {

println("Total");

}

}

class Student implements Marks,Total {

int StudentID

}

The output of the above program would be −

Total

Marks1

Extending Traits

Traits may extend another trait, in which case you must use the **extends** keyword. In the following code example, we are extending the Total trait with the Marks trait.

[Live Demo](http://tpcg.io/93Toux)

class Example {

static void main(String[] args) {

Student st = new Student();

st.StudentID = 1;

println(st.DisplayMarks());

}

}

trait Marks {

void DisplayMarks() {

println("Marks1");

}

}

trait Total extends Marks {

void DisplayMarks() {

println("Total");

}

}

class Student implements Total {

int StudentID

}

The output of the above program would be −

Total

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.

[Live Demo](http://tpcg.io/DhJykZ)

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.

[Live Demo](http://tpcg.io/F0Ud5Y)

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.

[Live Demo](http://tpcg.io/cNTYnU)

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.

[Live Demo](http://tpcg.io/OdRLoI)

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.

[Live Demo](http://tpcg.io/PQqb6J)

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.

[Live Demo](http://tpcg.io/Vbnr3i)

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.

[Live Demo](http://tpcg.io/mmicyq)

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. |
| 2 | [findAll()](https://www.tutorialspoint.com/groovy/groovy_findall.htm)  It finds all values in the receiving object matching the closure condition. |
| 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. |
| 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. |

Groovy allows one to omit parentheses around the arguments of a method call for top-level statements. This is known as the "command chain" feature. This extension works by allowing one to chain such parentheses-free method calls, requiring neither parentheses around arguments, nor dots between the chained calls.

If a call is executed as **a b c d**, this will actually be equivalent to **a(b).c(d)**.

DSL or Domain specific language is meant to simplify the code written in Groovy in such a way that it becomes easily understandable for the common user. The following example shows what exactly is meant by having a domain specific language.

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

print lst

The above code shows a list of numbers being printed to the console using the println statement. In a domain specific language the commands would be as −

Given the numbers 1,2,3,4

Display all the numbers

So the above example shows the transformation of the programming language to meet the needs of a domain specific language.

Let’s look at a simple example of how we can implement DSLs in Groovy −

class EmailDsl {

String toText

String fromText

String body

/\*\*

\* This method accepts a closure which is essentially the DSL. Delegate the

\* closure methods to

\* the DSL class so the calls can be processed

\*/

def static make(closure) {

EmailDsl emailDsl = new EmailDsl()

// any method called in closure will be delegated to the EmailDsl class

closure.delegate = emailDsl

closure()

}

/\*\*

\* Store the parameter as a variable and use it later to output a memo

\*/

def to(String toText) {

this.toText = toText

}

def from(String fromText) {

this.fromText = fromText

}

def body(String bodyText) {

this.body = bodyText

}

}

EmailDsl.make {

to "Nirav Assar"

from "Barack Obama"

body "How are things? We are doing well. Take care"

}

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

How are things? We are doing well. Take care

The following needs to be noted about the above code implementation −

* A static method is used that accepts a closure. This is mostly a hassle free way to implement a DSL.
* In the email example, the class EmailDsl has a make method. It creates an instance and delegates all calls in the closure to the instance. This is the mechanism where the "to", and "from" sections end up executing methods inside the EmailDsl class.
* Once the to() method is called, we store the text in the instance for formatting later on.
* We can now call the EmailDSL method with an easy language that is easy to understand for end users