Groovy Project

Authors: Jack Contreras, Ann Chang

Email: jacobc1@email.arizona.edu, achang7@email.arizona.edu

Abstract

Introduction

History

Groovy was first appeared in 2003 and the first version was released in 2007. James Strachan is the designer of Groovy and collaborated with Guillaume Laforge, Jochen Theodorou, Paul King, and Cedric Champeau to develop the language. The purpose of the language was to assist the Java community to have a compatible language for the Java platform and still in use today.

Control Structures

0.1 Order of Evaluation

```
if ( x > 5 && y < 0)
```

In Groovy, it would evaluate an expression from left to right. When examining the example above, it would check if x is greater than 5 first. Since Groovy does use short circuit evaluation, if the first statement is FALSE, it would not evaluate the second statement. Therefore, when evaluating the example above, if x is less than 5, it would not check if y is less than 0 because the ADD expression would only evaluate to TRUE when both statement is TRUE. This also applies to OR expression. It would evaluate the statement to TRUE if the first statement is TRUE.

0.2 Statements – Selection

```
switch(a) {
        case 1:
           println("The value of a is an Integer");
           break;
        case [21, 'test', 9.12]:
           println("The value is a List");
           break;
        case { a instanceof Float }:
           println("The value is a Float");
           break;
        default:
           println("The value is unknown");
           break;
}
```

In Groovy we can use different classifiers for a switch statement instead of only an integer. There is a function called isCase(), which users can implement different classifiers. Groovy already allows users to compare Classes (isInstance), Object (equals), Collection (contains), and expressions (match). The switch function also includes a default, which allows the user to set a response if the parameter doesn't fit any cases.

0.3 Statements – Iteration

```
for(int i in 1..5) {
    println(i);
}
```

The example above is another way to write a for loops in Groovy. It works similar to a for loop; however, it sets its range differently. The output of the example above will print numbers from 1 to 5. Any number that is declared before the IN key word would be the start of i. It would increment by 1 till it gets to the number that is declared after the dot.

```
0.step 5, 2, {
   println(it)
}
```

Groovy supports a function called step, which can iterate through numbers like a for loop. The 0 in the example above is the starting number in the loop, the 5 is stopping point (inclusive), and the 2 is how much it is increments. If the user wants to use the numbers, the variable would be "it". Therefore, the output above would be 0, 2, 4 on separate lines.

0.4 Error Handling

```
try {
   //Protected code
} catch(ArrayIndexOutOfBoundsException ex) {
   println("Catching the Array out of Bounds exception");
} finally {
   //The finally block always executes.
}
```

Groovy handles run-time errors through the try catch block. It would run the code in try block and when it encounters an error, it would go to one of the catch clauses. The user can have cases handling a specific error or have an catch clause catch any error and have general error message. The final-clause is executed regardless of an error is thrown or not.

Data Types

0.5 Type System

Current versions of Groovy support both dynamic and static typing. Groovy allows for either Java like syntax, with typing variables and return types, or allows you to not include typing, or some combination of the two. When types are not statically typed, they are inferred and checked at run-time.

0.6 Primitive Types

Groovy supports the same primitive types as Java. This includes byte, short, int, long, float, double, char, and Boolean.

0.6.1 Integral Numbers

Byte, Short, Int, and Long are all classified as integral numbers (whole numbers). Each of these types behave similarly, with the exception that they have different maximum and minimum values. Byte is an 8 bit number, short is 16 bit, int is 32 bit, and long is 64 bit. The Groovy compiler is able to recognize these numbers written in decimal, binary (with prefix 0b), octal (with prefix 0), and hexadecimal (with prefix 0x) form.

```
byte b = 9
short s = 0b1001
int i = 011
long l = 0x9
assert b == s && s == i && i == l
```

These number types all support the operations: addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (**).

0.6.2 Decimal Numbers

Groovy uses both float (single precision) and double (double precision) for decimal values. Just like in Java, Groovy also supports using scientific notation for decimal types using e or E. Floats and Decimals support the same operations as the integral numeric types. These operations are also supported between the two number types, often resulting in a float or double result.

0.6.3 Characters

Groovy has the primitive type char, although it does not have an explicit char literal statement. Since String literals can be defined by single quotes, instantiating a char requires casting to char. Groovy provides three ways to do this:

```
String s = 'A'
char c1 = s
char c2 = s as char
char c3 = (char) s
```

0.7 Data Collection

Groovy's data collection types all come from their Java equivalents. Groovy supports all data collection types that come with Java.util (List, Map, Set, etc.). However, Groovy allows for instantiation of Lists and Maps without creating Java objects. Groovy also uses Java's Arrays and has a collection called Ranges.

0.7.1 Lists

Groovy's list type is based upon Java's ArrayList object type. This allows for a dynamic like sizing of the list. Unlike Java, though, Groovy does not require list elements to be of the same type.

```
def same_type = [1, 2, 3, 4, 5]
def diff_type = [1, 'a', true, 3.14]
```

The above code will compile even though the second array contains different types.

0.7.2 Maps

Groovy's map type is based upon Java's LinkedHashMap. In Groovy a new map can be created using the "[:]", representing an empty map. Groovy also provides two ways to both add and access data to/from a map.

```
def capitals = [:]
capitals['Arizona'] = 'Phoenix'
capitals.Colorado = 'Denver'

assert capitals['Colorado'].equals('Denver')
assert capitals.Arizona.equals('Phoenix')
```

As presented in the above code snippit, maps can be manipulated using square bracket syntax "dictionary[key] = value" or a dot syntax "dictionary.key = value".

Subprograms

Groovy contains many of the same subprogram features as Java. This includes packages, classes, interfaces, and methods. In addition to these subprograms, Groovy also allows for closure statements, allowing for smaller subroutine definition.

0.8 Methods

Groovy provides multiple ways to define methods since it is optionally typed (Either statically or dynamically). With each method, the programmer is only required to give the method a name and a list of parameters with the method. Groovy does not require a return type and also does not require parameters to have a type reference, leaving it up to the programmer to either provide type referencing or completely disregard it. Groovy also makes it optional to provide visibility modifiers (public, private, protected) to method declarations, defaulting to public visibility if no modifier is given.

```
static int fact(int n){
...
}

static def listSum(list){
...
}
```

In the above examples, we can see how a static method can be defined using type referencing and without type referencing. In the end, Groovy's compiler is able to compile both methods without error. Instance methods follow the same pattern but are written without the static tag.

0.9 Classes

Groovy's class syntax is similar to Java's, but just like with method declaration, Groovy does not require visibility and defaults visibility to public. In Groovy, defining a constructor is not required. They can be defined similarly to Java, or if no constructor is given then a default constructor is assigned at compile time. When calling the constructor, Groovy allows for the use of named parameters as well as positional parameters. So, in the case that a constructor is not defined, all fields can still be set using the named parameter syntax. Suppose the following class declaration for a cylinder is given:

```
class Cylinder{

    float radius
    float height

    Cylinder(radius, height){
    }

    def surface_Area(){
    }

    def volume(){
    }
}
```

We can then construct a new Cylinder object by either using positional or named parameters.

```
Cylinder c1 = new Cylinder(4,7)
Cylinder c2 = new Cylinder(height : 5, radius : 3)
```

Also, unlike Java, Groovy does not require the file name of your program match the class name in the file. The main benefit of this, is that Groovy allows for multiple class definitions to be made in one file. The Groovy compiler will thus compile each of these classes into their own .class executable.

0.10 Closures

Groovy has a subprogram type called a closure, which is a block of code that acts like an object, and can be assigned to a variable and passed to other methods. Closures can take parameters, and can also manipulate data that is within its lexical scope (i.e if a closure is defined in a class, it can access data that is visible within the class). Closures are surrounded by curly brackets "" and start with comma separated parameters followed by an arrow "-¿". After the arrow is the body of the closure. The following is a simple example of a closure definition followed by a call to the closure, which is similar to calling a method.

```
def add = {x, y -> x + y}
add(5,6)
```

Closures in Groovy also support currying. They can be curried from either the left, using ".curry", or the right, using ".rcurry".

```
def copy = {n, string = string * n}

def double = copy.curry(2)
```

```
def lessThan = {x , y -> x < y}

def lessThan100 = lessThan.rcurry(100)
```

The above examples show how to turn a copy n times function into the specialized double function and how to turn the less than function into the specialized less than 100.

Summary

Reference

References

1.