



Groovy Language
Productividad para la plataforma Java

Objetivos

- Conocer las opciones que nos brinda la JVM (plataforma Java)
- Historia de Groovy
- Cómo iniciar...
- Conceptos específicos del lenguaje

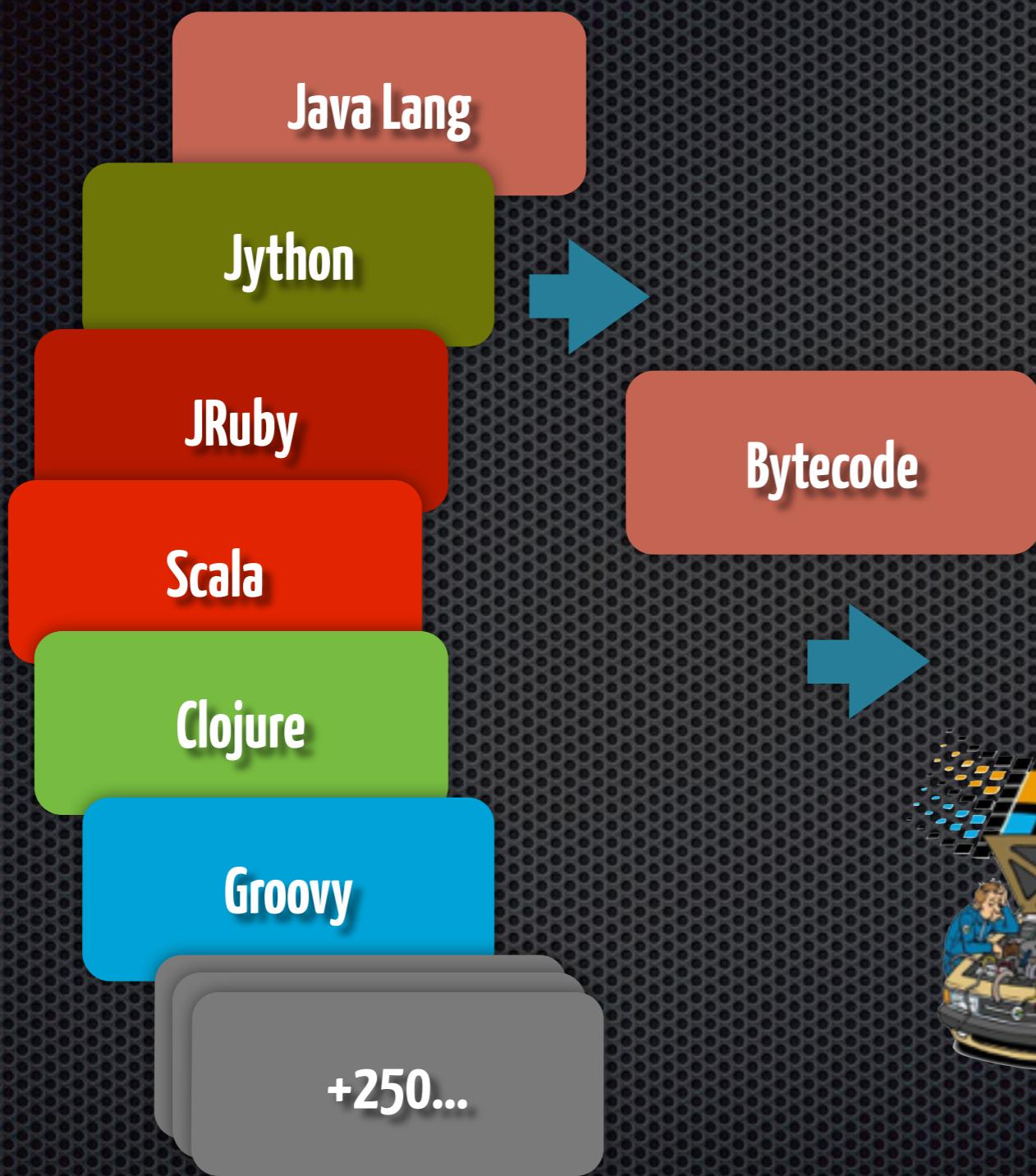


JVM

- El término “Java” puede referir a:
 - El lenguaje de programación Java
 - La API de Java
 - La plataforma de ejecución (máquina virtual) JVM



Lenguajes para la JVM





Java™



Clojure



JRuby



... +250



Historia de Groovy

- Creado por James Strachan en 2003, primer release 1.0 en 2007. Strachan se inspiró en Python, Ruby, Perl y Smalltalk para diseñar el lenguaje
- En 2007 ganó primer lugar en JAX Innovation Awards, en 2008 Grails ganó el segundo lugar
- G2One fue adquirida por SpringSource y ésta a su vez por VMWare



Características de Groovy

- Open Source (Apache License v2.0)
- Orientado a Objetos
- Se integra perfectamente con Java
- Curva de aprendizaje casi nula
- Tipado opcional (dinámico y estático)

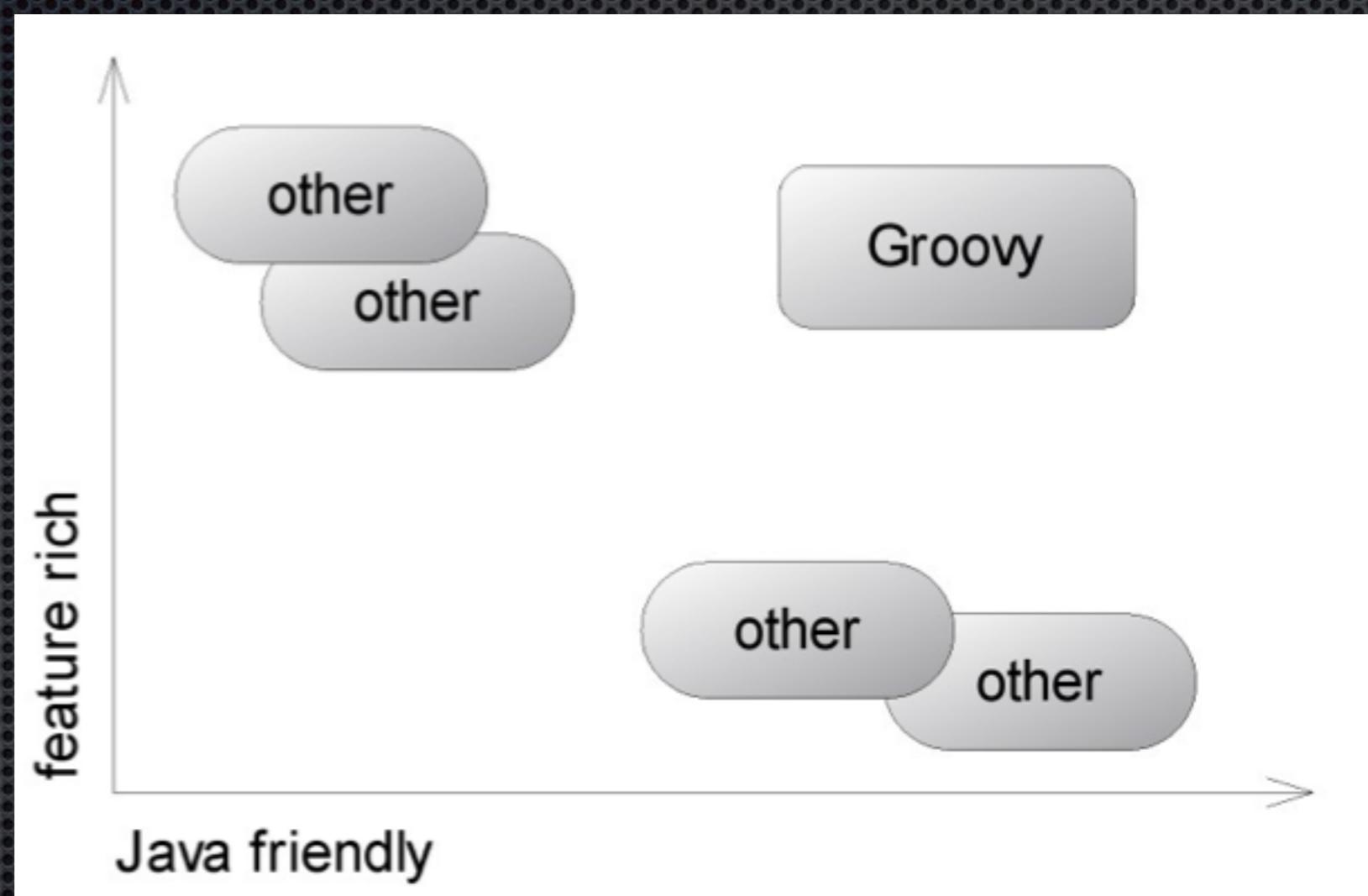


Características de Groovy

- Dinámico
- Ágil y compacto
- Con características “funcionales”
- Funciona también como lenguaje para “scripting”
- Integra de manera nativa soporte extra para colecciones



Características de Groovy



Orientación a Objetos

- Modularidad.
- Reusabilidad.
- Alta cohesión.
- Bajo acoplamiento.



Modularidad

- Trabajar por partes.
- Al hacerlo, subdividimos una aplicación en partes más pequeñas, independientes y manejables.
- Reducimos complejidad.
- Al crear módulos, solo pequeñas partes son afectadas cuando un cambio es realizado.



Reusabilidad

- Identificar partes de un sistema que varias entidades utilizan en común. Diseñarlas independientemente.
- Diseñarlas pensando en que puedan ser utilizadas desde diferentes puntos del sistema.
- No reinventar la rueda constantemente. *NIH*.



Alta cohesión

- Característica del software que determina qué tan estrechamente relacionados están los elementos dentro de un componente (objeto), con respecto a su definición.
- Qué tan coherentes son los atributos y métodos con el objeto mismo.



Bajo acoplamiento

- Característica del software que determina el grado de independencia entre los componentes con respecto a otros.
- Qué tanta afectación tiene un cambio en un componente con respecto a los demás.
- Qué tanta afectación tiene un cambio en los demás hacia el componente.
- Bajo acoplamiento aumenta la mantenibilidad.



Integración con Java



Integración con Java

- Groovy es “Java Friendly”, lo cual expresa que:
 - Se acopla perfectamente con bibliotecas y el entorno actual de Java
 - La sintaxis de Groovy está muy alineada con la de Java, es muy fácil leer código Groovy para un programador Java

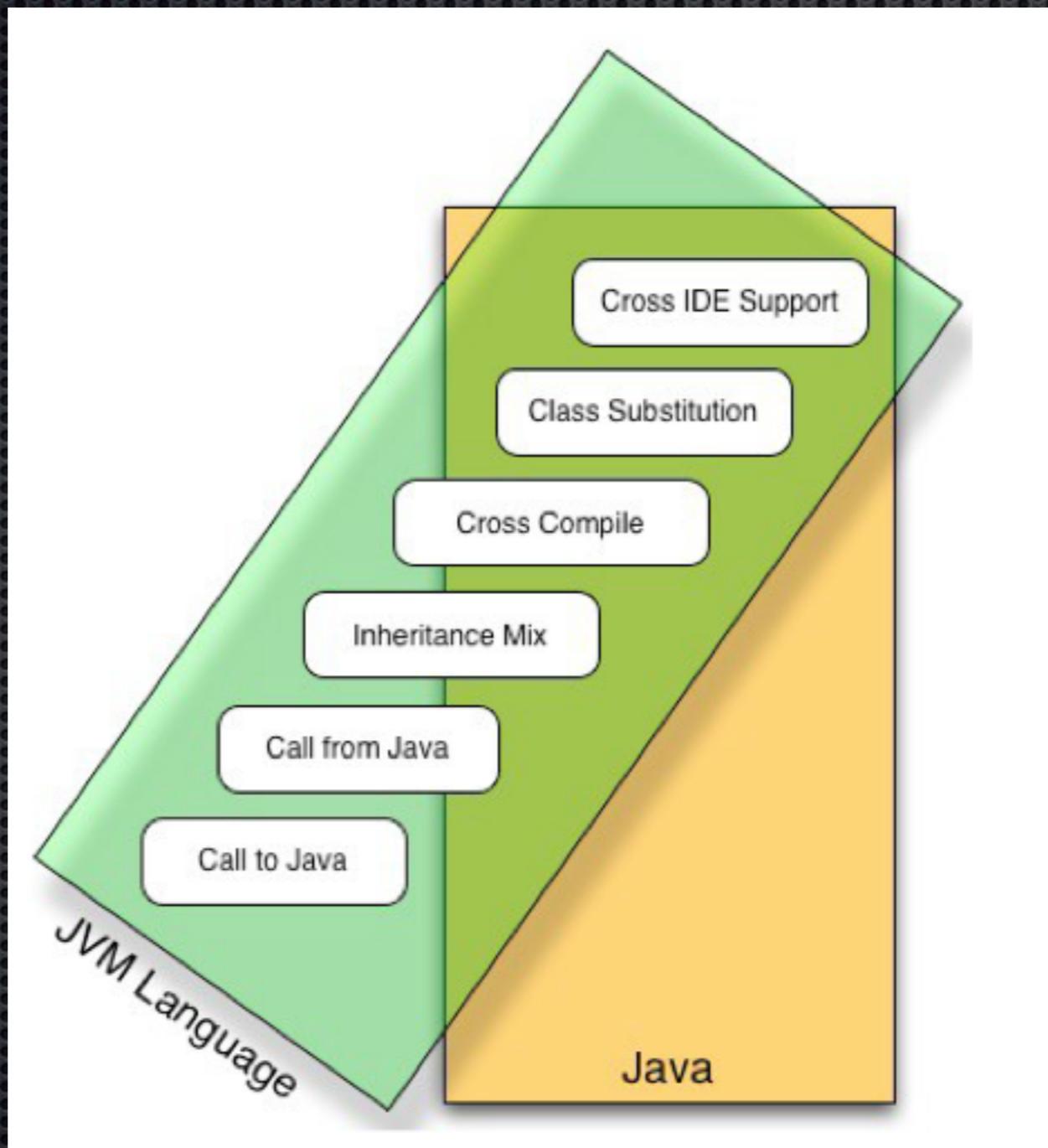


Integración con Java

- Se puede invocar código Java desde código Groovy
- Todos los tipos de dato en Groovy son subtipos de `java.lang.Object`
- También se pueden utilizar clases compiladas de Groovy en programas Java
 - `MiClaseGroovy.groovy > MiClaseGroovy.class`
 - Poner el `.class` en el `classpath` de algún programa Java



Niveles de integración



Setup

- Groovy requiere la instalación del JDK (Java Development Kit)
- Agregar **JAVA_HOME** a variables de entorno y **%JAVA_HOME%/bin** a la variable **PATH**
- Descargar y descomprimir **Groovy**, agregar **GROOVY_HOME** a las variables de entorno y **%GROOVY_HOME%/bin** a la variable **PATH**



Instalación de Groovy

- Groovy contiene 3 programas:
 - **groovysh** - Shell
 - **groovyConsole** - GUI (interfaz gráfica)
 - **groovy script.groovy** - Intérprete



Hola Mundo en Java

```
 1 ▼ public class HelloWorld{  
 2     private String name;  
 3  
 4 ▼     public static void main(String[] args) {  
 5         HelloWorld helloWorld = new HelloWorld();  
 6         helloWorld.setName("Javier");  
 7         System.out.println(helloWorld.sayHello());  
 8     }  
 9     public String sayHello(){  
10         return "Hello " + name;  
11     }  
12     public void setName(String name){  
13         this.name = name;  
14     }  
15     public String getName(){  
16         return name;  
17     }  
18 }
```

Line: 18:2 | Java | Tab Size: 4 | getName()



Hola Mundo en Groovy

```
1 ▼ public class HelloWorld{
2     private String name;
3
4     public static void main(String[] args) {
5         HelloWorld helloWorld = new HelloWorld();
6         helloWorld.setName("Javier");
7         System.out.println(helloWorld.sayHello());
8     }
9     public String sayHello(){
10        return "Hello " + name;
11    }
12    public void setName(String name){
13        this.name = name;
14    }
15    public String getName(){
16        return name;
17    }
18 }
```

Line: 17:3 | Groovy | Tab Size: 4 | getName()



Diferencias

El punto y coma es
opcional

```
public class HelloWorld{  
    private String name;  
  
    public static void main(String[] args) {  
        HelloWorld helloworld = new HelloWorld();  
        helloworld.setName("Javier");  
        System.out.println(helloworld.sayHello());  
    }  
    public String sayHello(){  
        return "Hello " + name;  
    }  
    public void setName(String name){  
        this.name = name;  
    }  
    public String getName(){  
        return name;  
    }  
}
```



Diferencias

- Se puede evitar la concatenación con la **interpolación de variables**
 - “**Hola \$nombre**” en lugar de “**Hola “ + nombre**
- No es necesario usar “**return**”, si la última sentencia de un método es una expresión Groovy la retornará
- Es posible evitar usar paréntesis al invocar métodos
 - `println “Hola mundo!”`



Diferencias

- Java es un lenguaje de tipado estático
- Groovy es un lenguaje de tipado dinámico
 - Podemos declarar variables con def, sin necesidad de definir su tipo
 - Duck Typing (definición semántica de variables en base a los valores que almacenan)



Hola mundo estilo Groovy

```
1 ▼ class HelloWorld{
2     def name
3     static main(args) {
4         def helloworld = new HelloWorld()
5         helloworld.name = "Javier"
6         println helloworld.sayHello()
7     }
8     def sayHello(){
9         "Hello $name"
10    }
11 }
```

Line: 3:2 | Groovy | Tab Size: 4 | Symbols



Scripting

GroovyConsole

```
1 //No need for a class
2 new File("/path/to/base")
3     if (it.isDirectory() && !it.list().length) new File("$it.absolutePath/PLACEHOLDER") << ''
4 }
5 println "Finished creating placeholders"
```

bnk@benek-mac ~ % groovysh
Groovy Shell (2.1.4, JVM: 1.7.0_21)
Type 'help' or '\h' for help.

```
groovy> println "Finished creating placeholders"
Finished creating placeholders
groovy:000> name = "Javier"
====> Javier
groovy:000> println "Hello $name"
Hello Javier
====> null
groovy:000>
```

Execution complete. Result was null. 5:41



Comentarios

```
1 //Comentario hasta el final de la linea  
2 /* Comentario entre bloques,  
3 este tipo de comentario puede abarcar  
4 multiples lineas */
```



Strings

```
1 'String con comilla simple'  
2  
3 "String con comilla doble"  
4  
5 '''String  
6 con triple  
7 comilla simple'''  
8  
9 """String con  
10 triple comilla  
11 doble"""
```



Strings

- Las cadenas con comilla simple son realmente Strings de Java
- Las cadenas con comilla doble son llamadas GStrings, y contienen adiciones hechas por Groovy como la interpolación de variables o métodos adicionales



Groovy vs Java

- Groovy evita que escribamos código ceremonioso que requiere Java, pero con el mismo estilo de programación y una sintaxis muy parecida
- Groovy es un lenguaje de programación orientado a conseguir mayor productividad



Abriendo un archivo en Java

```
1 import java.io.BufferedReader;
2 import java.io.FileNotFoundException;
3 import java.io.FileReader;
4 import java.io.IOException;
5
6 public class LeerArchivo {
7     BufferedReader br = null;
8     try{
9         br = new BufferedReader(new FileReader("../simpleFile.txt"));
10        String linea = null;
11        while((linea = br.readLine()) != null){
12            System.out.println(linea);
13        }
14    } catch(FileNotFoundException e) {
15        e.printStackTrace();
16    } catch (IOException e) {
17        e.printStackTrace();
18    } finally {
19        if (br != null) {
20            try {
21                br.close();
22            } catch(IOException e) {
23                e.printStackTrace();
24            }
25        }
26    }
27}
```



Lo mismo en Groovy

```
1 new File("../simpleFile.txt").eachLine { linea ->
2     println linea
3 }
```



Default Imports

- Java incluye por default imports a **java.util.***
- Groovy añade los siguientes:
 - **java.io.***
 - **java.lang.***
 - **java.math.BigDecimal**
 - **java.math.BigInteger**
 - **java.net.***
 - **java.util.***
 - **groovy.lang.***
 - **groovy.util.***



Unchecked Exceptions

- Java maneja el concepto de **Checked** y **Unchecked Exceptions**
 - Checked son excepciones que el compilador no permitirá que pasen desapercibidas
 - Unchecked son las que el compilador puede ignorar aunque no estén siendo tratadas por nosotros
- En Groovy es muy diferente, **todas las excepciones son tratadas como Unchecked Exceptions**



Safe navigation

- NullPointerException es una de las excepciones más comunes en Java
- Se tiene que evitar revisando si las variables a ocupar son nulas antes de utilizarlas



Control de flujo

- If-Else y Else-If es similar al de Java

```
1 def a = 3
2 if (a > 2) {
3   println "was true"
4 } else {
5   println "was false"
6 }
```



Control de flujo

- Groovy soporta el operador “ternario”, una versión corta del if-else

```
1 def y = 5
2 def x = (y > 1) ? "si" : "no"
3 assert x == "si"
```



Switch

- El switch con respecto al de Java es mucho más completo (power switch)
- Puede manejar más tipos de comparaciones

```
1 def x = 1.23
2 def resultado = ""
3
4 switch ( x ) {
5     case "foo":
6         resultado = "palabra foo"
7         // sin break
8     case "bar":
9         resultado += "bar"
10    case [4, 5, 6, 'inList']:
11        resultado = "lista"
12        break
13    case 12..30:
14        resultado = "rango"
15        break
16    case Integer:
17        resultado = "integer"
18        break
19    case Number:
20        resultado = "number"
21        break
22    default:
23        resultado = "default"
24 }
25 assert resultado == "number"
```

Loops

- Groovy soporta **while** y **for** tal como en Java

```
1 def x = 0
2 def y = 5
3
4 while ( y-- > 0 ) {
5     x++
6 }
7
8 assert x == 5
```



Loops

```
1 for (int i = 0; i < 5; i++) {  
2 }  
3  
4 // iterar sobre un rango  
5 def x = 0  
6 for ( i in 0..9 ) {  
7     x += i  
8 }  
9 assert x == 45  
10  
11 // iterar sobre una lista  
12 x = 0  
13 for ( i in [0, 1, 2, 3, 4] ) {  
14     x += i  
15 }  
16 assert x == 10  
17
```



Loops

```
1 // iterar sobre un arreglo
2 array = (0..4).toArray()
3 x = 0
4 for ( i in array ) {
5     x += i
6 }
7 assert x == 10
8
9 // iterar sobre un mapa
10 def map = ['abc':1, 'def':2, 'xyz':3]
11 x = 0
12 for ( e in map ) {
13     x += e.value
14 }
15 assert x == 6
16
```



Loops

```
1 // iterar los valores de un mapa
2 x = 0
3 for ( v in map.values() ) {
4     x += v
5 }
6 assert x == 6
7
8 // iterar sobre los caracteres de una cadena
9 def text = "abc"
10 def list = []
11 for (c in text) {
12     list.add(c)
13 }
14 assert list == ["a", "b", "c"]
```



Loops con each

- Groovy añade closures para poder iterar:
 - each
 - eachWithIndex

```
1 def stringList = [ "java", "perl", "python", "ruby", "c#", "cobol",
2                         "groovy", "jython", "smalltalk", "prolog", "m", "yacc" ];
3
4 stringList.each() { print " ${it}" }; println "";
5
6 stringList.eachWithIndex() { obj, i -> println " ${i}: ${obj}" };
```



Collections

- Groovy añade soporte nativo para colecciones
 - Listas
 - Mapas
 - Arreglos



Collections

- Las listas en Groovy son en realidad ArrayList

```
1 def listaVacia = []
2
3 def asistentes = ["Estela", "Angel", "Vero"]
4 asistentes.add("Javier")
5
6 println listaVacia.size()
7 println asistentes.size()
8
9 println asistentes[2]
10
11 println asistentes.class
```



Collections

- Groovy añade un tipo de colección para **Rangos**

```
1 def rango = 5..18 //rango inclusivo
2 println rango
3 println rango.size()
4
5 rango = 5..<30 //half-open
6 println rango
7 println rango.size()
8
9 rango = 'a'..'z'
10 println rango|
11 println rango.size()
12
13 println rango.from
14 println rango.to
```



Collections

- Rangos en loops

```
1 for (i in 1..10) {  
2     println "Hola ${i}"  
3 }  
4  
5 (1..10).each { i ->  
6     println "Hola ${i}"  
7 }  
8  
9 switch (years) {  
10     case 1..10: tasaInteres = 0.076; break;  
11     case 11..25: tasaInteres = 0.052; break;  
12     default: tasaInteres = 0.037;  
13 }  
14  
15
```



Collections

- El manejo de mapas también es nativo

```
1 def mapaVacio = [:]  
2  
3 def asistentes = ["Javier":"javier@pgj.gob.mx", "Angel":"angel@pgj.gob.mx"]  
4  
5 println mapaVacio.size()  
6 println asistentes.size()  
7  
8 println asistentes["Javier"]
```



Operador star-dot *.

- Al manejar colecciones, Groovy provee un poderoso operador llamado star-dot, para invocar métodos en todos los elementos de una colección

```
1 def nombres = ["Armando", "Alejandro", "Javier"]
2 println nombres*.size()
3
4 def numeros = [2, 4, 6, 8]
5 println numeros*.plus(10)
```



Assertions

- assert = afirmar, hacer valer
- Palabra reservada en Java y Groovy para poner a prueba expresiones
- Nos sirven para validar que el lenguaje funciona como lo esperamos, de lo contrario el assert lanzará un error



Assertions

- Si no se genera un error, los asserts pasaron



A screenshot of a Groovy code editor window titled "Assertions.groovy". The code contains several assert statements:

```
1 assert(true)
2
3 assert 1 == 1
4
5 def x = 1
6 assert x == 1
7
8 def y = 1 ;
9 assert y == 1
```

The code editor interface includes a toolbar at the top, a status bar at the bottom displaying "Line: 9:14 | Groovy", and various tool buttons.



Assertions

The screenshot shows two windows. On the left is a terminal window titled "AssertionsFail.groovy" with the following code:

```
1 def
2 def
3 asse
```

On the right is a "Groovy Console" window titled "Running 'AssertionsFail.groovy'...". It displays the following error output:

```
Caught: Assertion failed:
assert b == a + a
| | | |
9 | 5 | 5
|
false

Assertion failed:
assert b == a + a
| | | |
9 | 5 | 5
|
false

at AssertionsFail.run(AssertionsFail.groovy:3)
```

Below the error output, the message "Program exited with code #1 after 1.50 seconds." is displayed. To the right of the message is a "copy output" link.



Tipos de dato

■ Sistema de tipos en Java

| Primitive Type | Wrapper Type | Description |
|----------------|---------------------|--|
| byte | java.lang.Byte | 8-bit signed integer |
| short | java.lang.Short | 16-bit signed integer |
| int | java.lang.Integer | 32-bit signed integer |
| long | java.lang.Long | 64-bit signed integer |
| float | java.lang.Float | Single-precision (32-bit) floating-point value |
| double | java.lang.Double | Double-precision (64-bit) floating-point value |
| char | java.lang.Character | 16-bit Unicode character |
| boolean | java.lang.Boolean | Boolean value (true or false) |



Tipos de dato

- En Groovy, todo es un objeto
- Al contrario de Java, no distingue entre tipos primitivos y tipos wrapper
- Groovy utiliza solamente los tipos wrapper de Java



Tipos numéricos

- `byte`
- `char`
- `short`
- `int`
- `long`
- `java.lang.BigInteger`

```
// primitive types
byte b = 1
char c = 2
short s = 3
int i = 4
long l = 5

// infinite precision
BigInteger bi = 6
```

- Si se utiliza `def` para declarar una variable con un valor numérico, el tipo que use Groovy dependerá del tamaño



Tipos numéricos

```
 1 def a = 1
 2 assert a instanceof Integer
 3
 4 // Integer.MAX_VALUE
 5 def b = 2147483647
 6 assert b instanceof Integer
 7
 8 // Integer.MAX_VALUE + 1
 9 def c = 2147483648
10 assert c instanceof Long
11
12 // Long.MAX_VALUE
13 def d = 9223372036854775807
14 assert d instanceof Long
15
16 // Long.MAX_VALUE + 1
17 def e = 9223372036854775808
18 assert e instanceof BigInteger
```



Literales numéricas

- Para forzar una literal a ser de un tipo definido utilizamos sufijos

| Type | Suffix |
|------------|--------|
| BigInteger | G or g |
| Long | L or l |
| Integer | I or i |
| BigDecimal | G or g |
| Double | D or d |
| Float | F or f |



Numéricos non-base 10

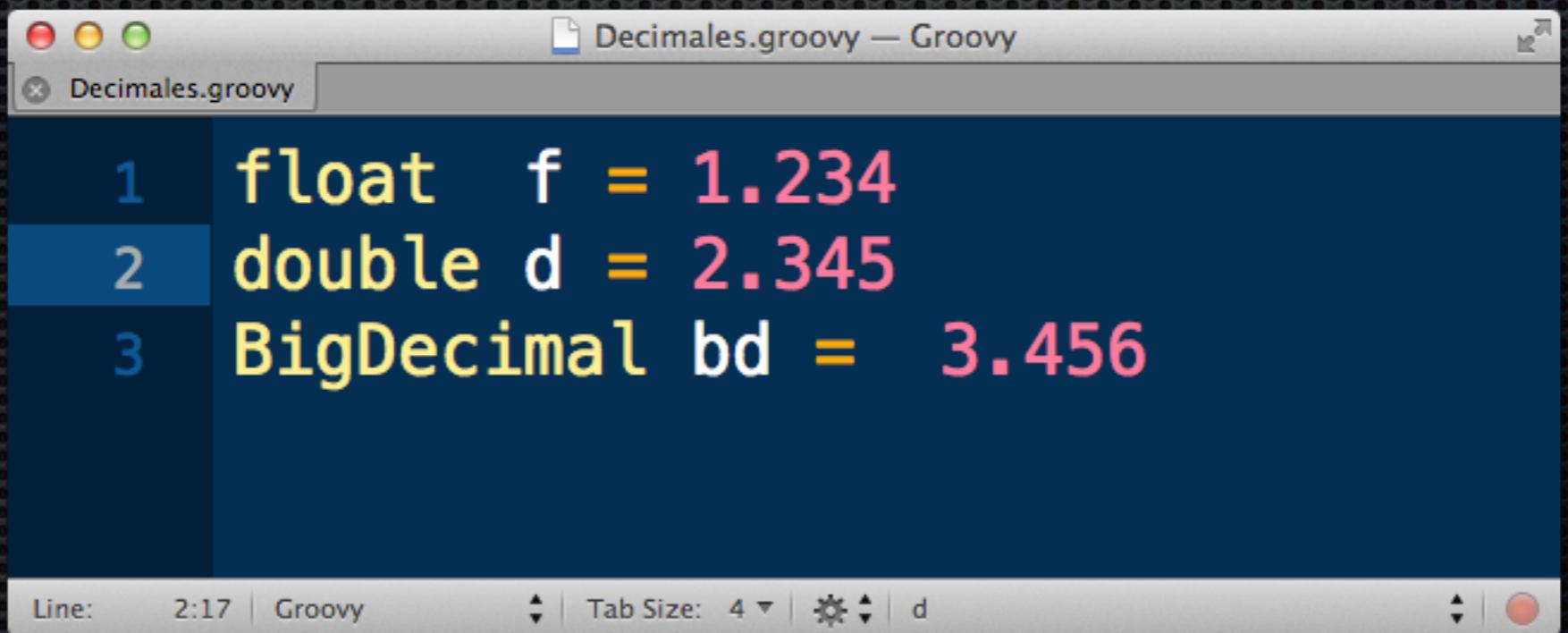
```
BinarioOctalHexa.groovy — Groovy  
BinarioOctalHexa.groovy  
1 //Binario -> 0b  
2 int xInt = 0b10101111  
3 assert xInt == 175  
4  
5 //Octal -> 0  
6 int xInt = 077  
7 assert xInt == 63  
8  
9 //Hexadecimal  
10 int xInt = 0x77  
11 assert xInt == 119
```

Line: 11:19 | Groovy | Tab Size: 4 | xlnt



Numéricos con punto

- **float**
- **double**
- **java.lang.BigDecimal**



```
Decimales.groovy — Groovy
Decimales.groovy

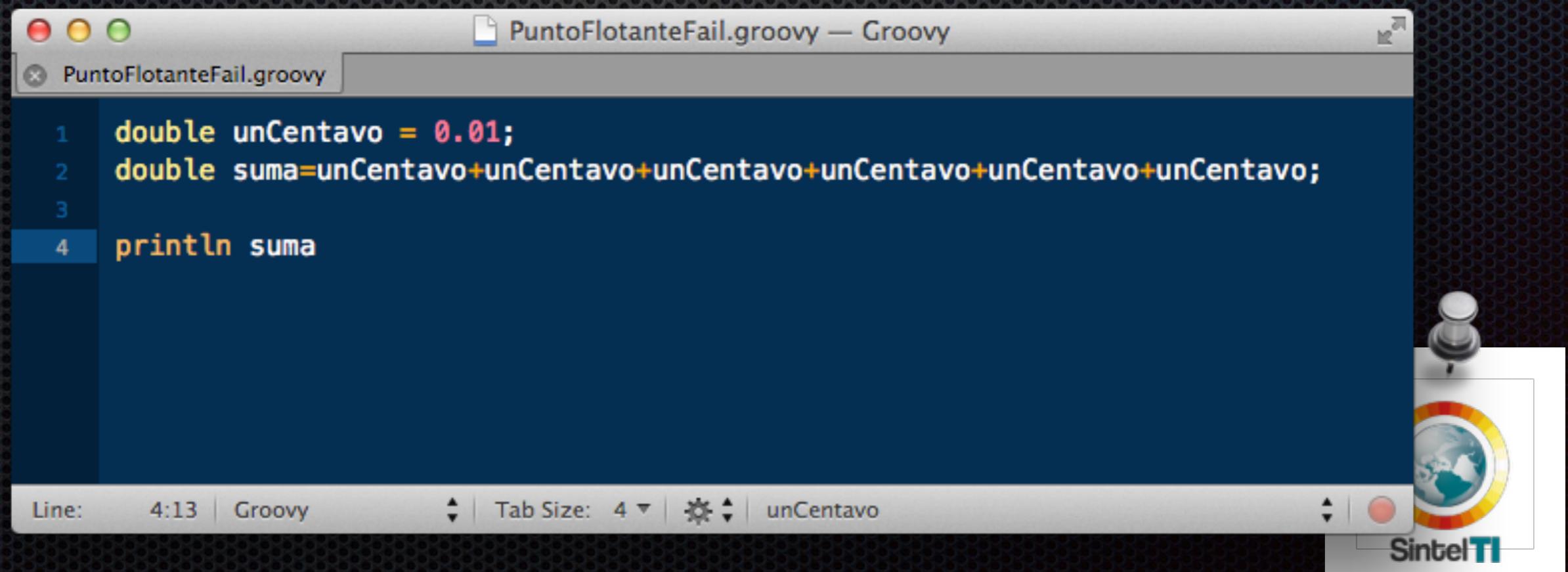
1 float f = 1.234
2 double d = 2.345
3 BigDecimal bd = 3.456
```

The screenshot shows a Groovy script named "Decimales.groovy" in a code editor. The code defines three variables: a float (f), a double (d), and a BigDecimal (bd) with their respective values. The code editor interface includes a title bar, tabs, status bar, and various toolbars.



Numéricos con punto

- Al contrario de Java, Groovy eligió **BigDecimal** como default para manejar decimales, debido a la imprecisión de los numéricos de punto flotante **float** y **double**



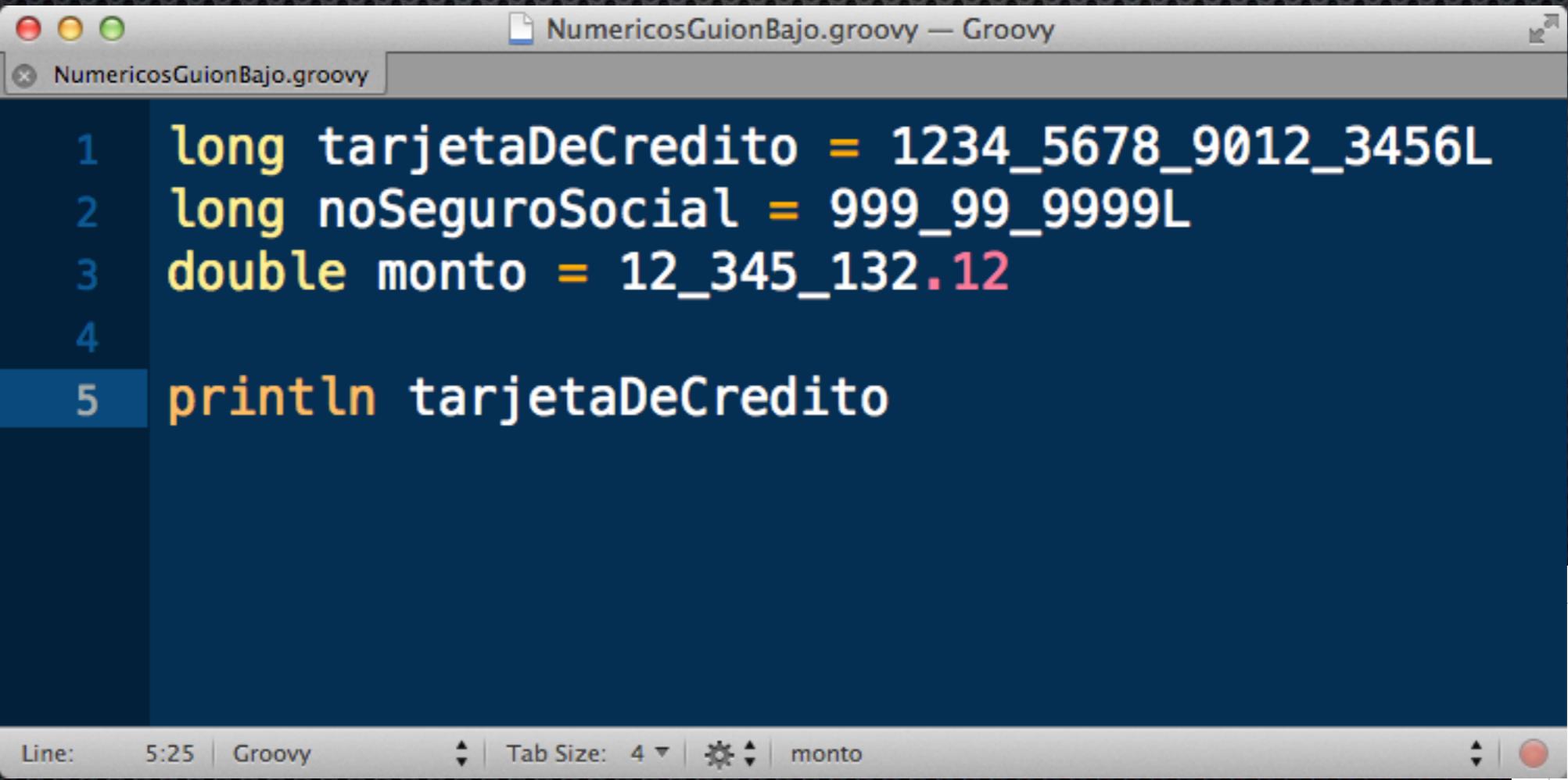
```
PuntoFlotanteFail.groovy — Groovy
PuntoFlotanteFail.groovy

1 double unCentavo = 0.01;
2 double suma=unCentavo+unCentavo+unCentavo+unCentavo+unCentavo+unCentavo;
3
4 println suma
```

Line: 4:13 | Groovy | Tab Size: 4 | unCentavo | Sintel TI

Numéricos

- Como apoyo visual en Groovy también es posible separar números con guiones bajos



The screenshot shows a Groovy code editor window titled "NumericosGuionBajo.groovy". The code contains the following lines:

```
1 long tarjetaDeCredito = 1234_5678_9012_3456L
2 long noSeguroSocial = 999_99_9999L
3 double monto = 12_345_132.12
4
5 println tarjetaDeCredito
```

The code editor has tabs for "NumericosGuionBajo.groovy" and "NumericosGuionBajo.groovy". The status bar at the bottom shows "Line: 5:25 | Groovy" and "Tab Size: 4". A logo for "Sintel TI" is visible in the bottom right corner.

Boolean

- Como en otros lenguajes, representa un valor verdadero o falso
- Sólo puede almacenar **true** o **false**



Operadores

- Groovy soporta los operadores existentes en Java
- Aunque añade algunos operadores más que resultan muy útiles
- Groovy también provee un mecanismo para **sobrecarga de operadores**



Operadores aritméticos

| Operator | Purpose |
|----------|----------------|
| + | addition |
| - | subtraction |
| * | multiplication |
| / | division |
| % | modulo |
| ** | power |



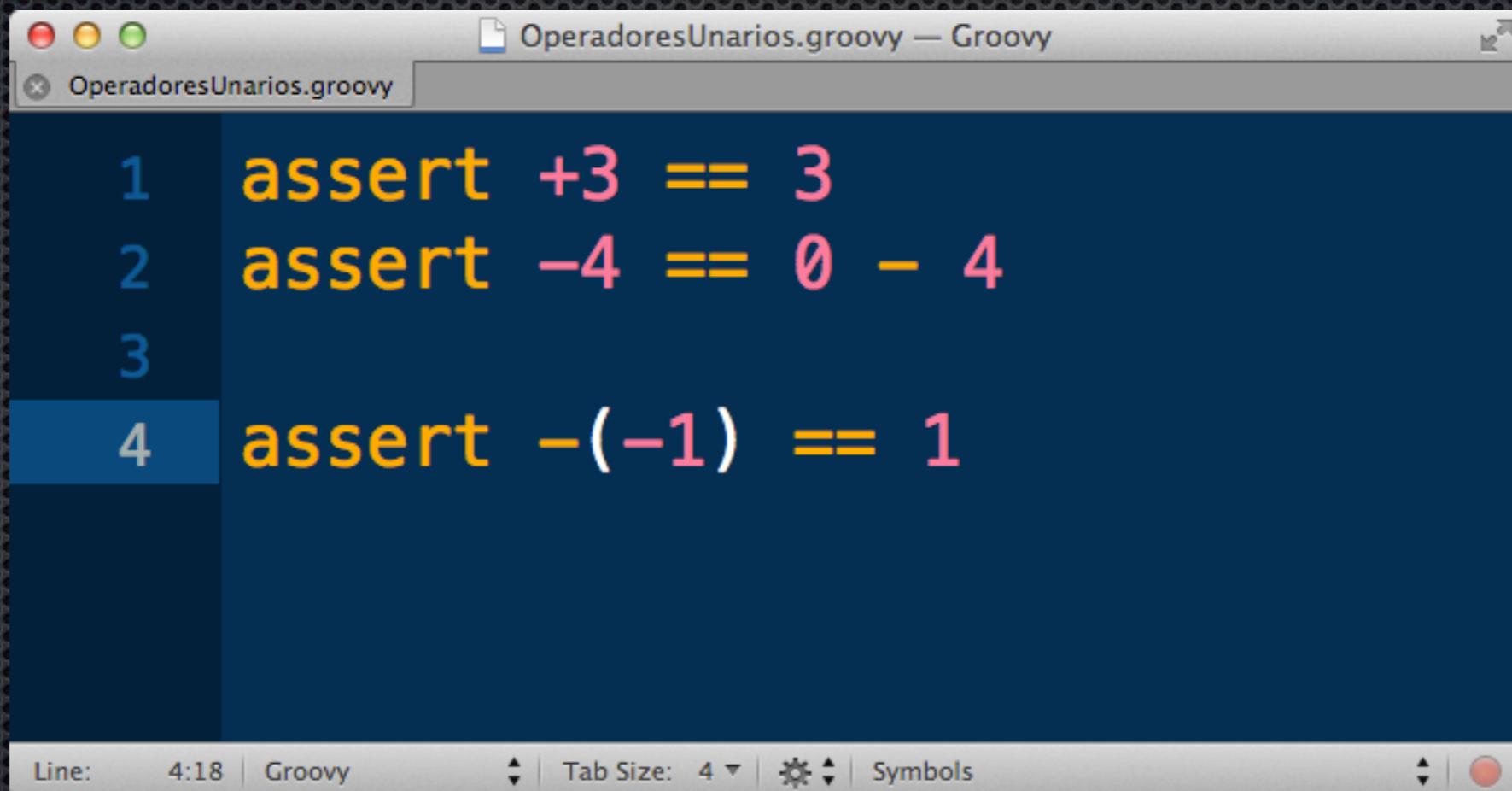
Operadores aritméticos

```
OperadoresAritmeticos.groovy — Groovy  
OperadoresAritmeticos.groovy  
1 assert 1 + 2 == 3 //suma  
2 assert 4 - 3 == 1 //resta  
3 assert 3 * 5 == 15 //multiplicacion  
4 assert 3 / 2 == 1.5 //division  
5 assert 10 % 3 == 1 //modulo  
6 assert 2 ** 3 == 8 //potencia
```



Operadores Unarios

- Reflejan el signo de un valor numérico



The screenshot shows a Groovy code editor window titled "OperadoresUnarios.groovy". The code contains four lines of Groovy script:

```
1 assert +3 == 3
2 assert -4 == 0 - 4
3
4 assert -(-1) == 1
```

The editor interface includes a toolbar at the top with file, edit, and search icons. The status bar at the bottom displays "Line: 4:18 | Groovy" and "Tab Size: 4". A pinned logo for "Sintel TI" is visible in the bottom right corner.



Operadores de incremento y decrecimiento

The screenshot shows a Groovy script named "OperadoresIncremento.groovy" in an IDE. The code demonstrates the use of increment operators (++ and --) in Groovy. It defines variables 'a' and 'b' with initial values of 2 and 3 respectively. It then increments 'a' by 1 using 'a++' and multiplies it by 3 to get 6, which is assigned to 'b'. An assert statement checks if 'a' equals 3 and 'b' equals 6. The next part of the code defines 'c' as 3, decrements it by 1 using 'c--' to get 2, and multiplies it by 2 to get 4, which is assigned to 'd'. Another assert statement checks if 'c' equals 2 and 'd' equals 4.

```
def a = 2
def b = a++ * 3
assert a == 3 && b == 6

def c = 3
def d = c-- * 2
assert c == 2 && d == 4
```



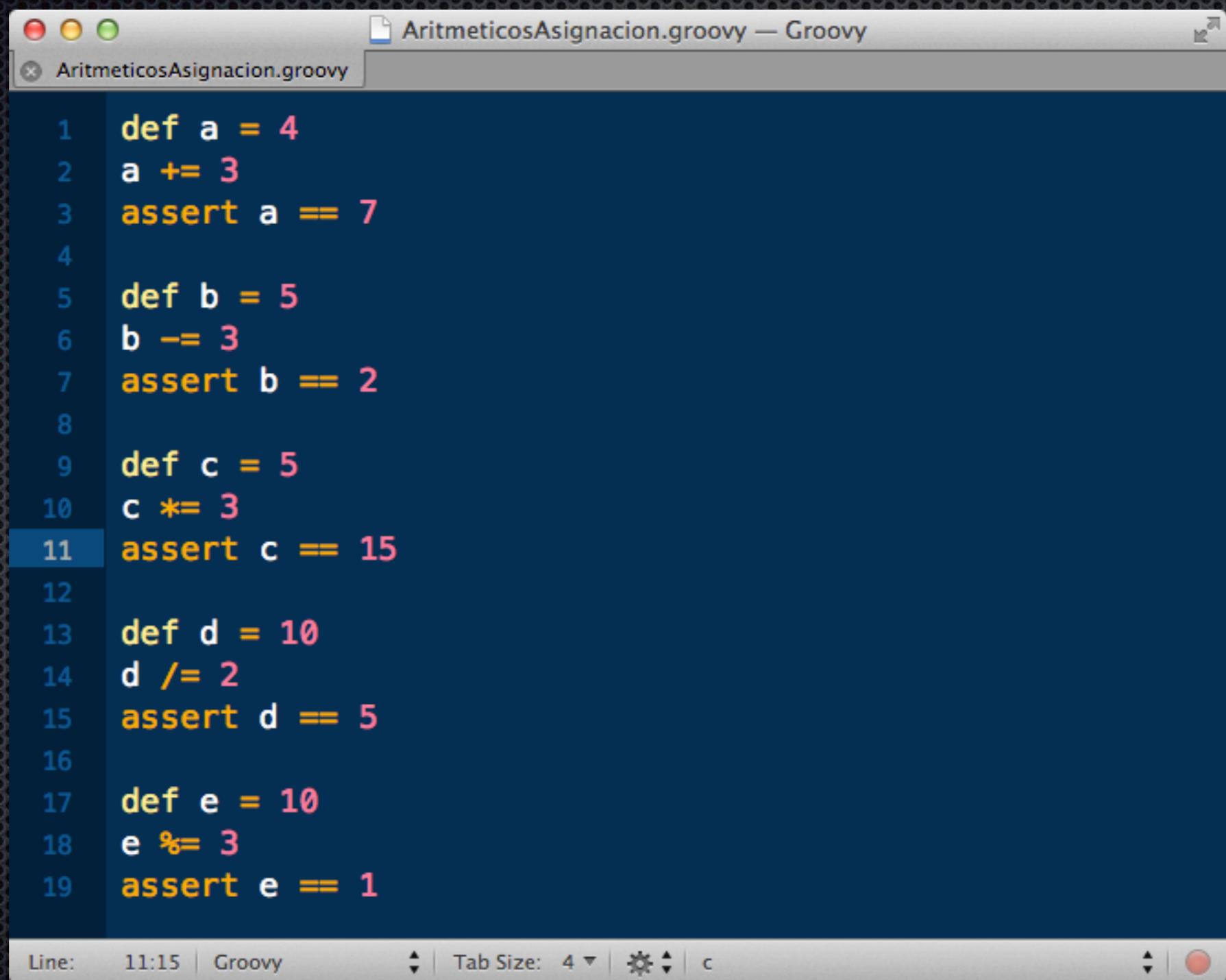
Aritméticos de asignación

- 
- 
- 
- 
- 

- Son útiles para realizar operaciones aritméticas y al mismo tiempo asignar el resultado



Aritméticos de asignación



The screenshot shows a Groovy script titled "AritmeticosAsignacion.groovy" in a code editor. The script demonstrates various arithmetic assignment operators:

```
1 def a = 4
2 a += 3
3 assert a == 7
4
5 def b = 5
6 b -= 3
7 assert b == 2
8
9 def c = 5
10 c *= 3
11 assert c == 15
12
13 def d = 10
14 d /= 2
15 assert d == 5
16
17 def e = 10
18 e %= 3
19 assert e == 1
```

The code uses the following assignment operators:

- Line 2: `a += 3` (Addition assignment)
- Line 7: `b -= 3` (Subtraction assignment)
- Line 10: `c *= 3` (Multiplication assignment)
- Line 14: `d /= 2` (Division assignment)
- Line 18: `e %= 3` (Modulo assignment)

The editor interface includes tabs for "AritmeticosAsignacion.groovy" and "Groovy", status bar showing "Line: 11:15 | Groovy", and a toolbar at the bottom.



Operadores relacionales

| Operator | Purpose |
|--------------------|-----------------------|
| <code>==</code> | equal |
| <code>!=</code> | different |
| <code><</code> | less than |
| <code><=</code> | less than or equal |
| <code>></code> | greater than |
| <code>>=</code> | greater than or equal |

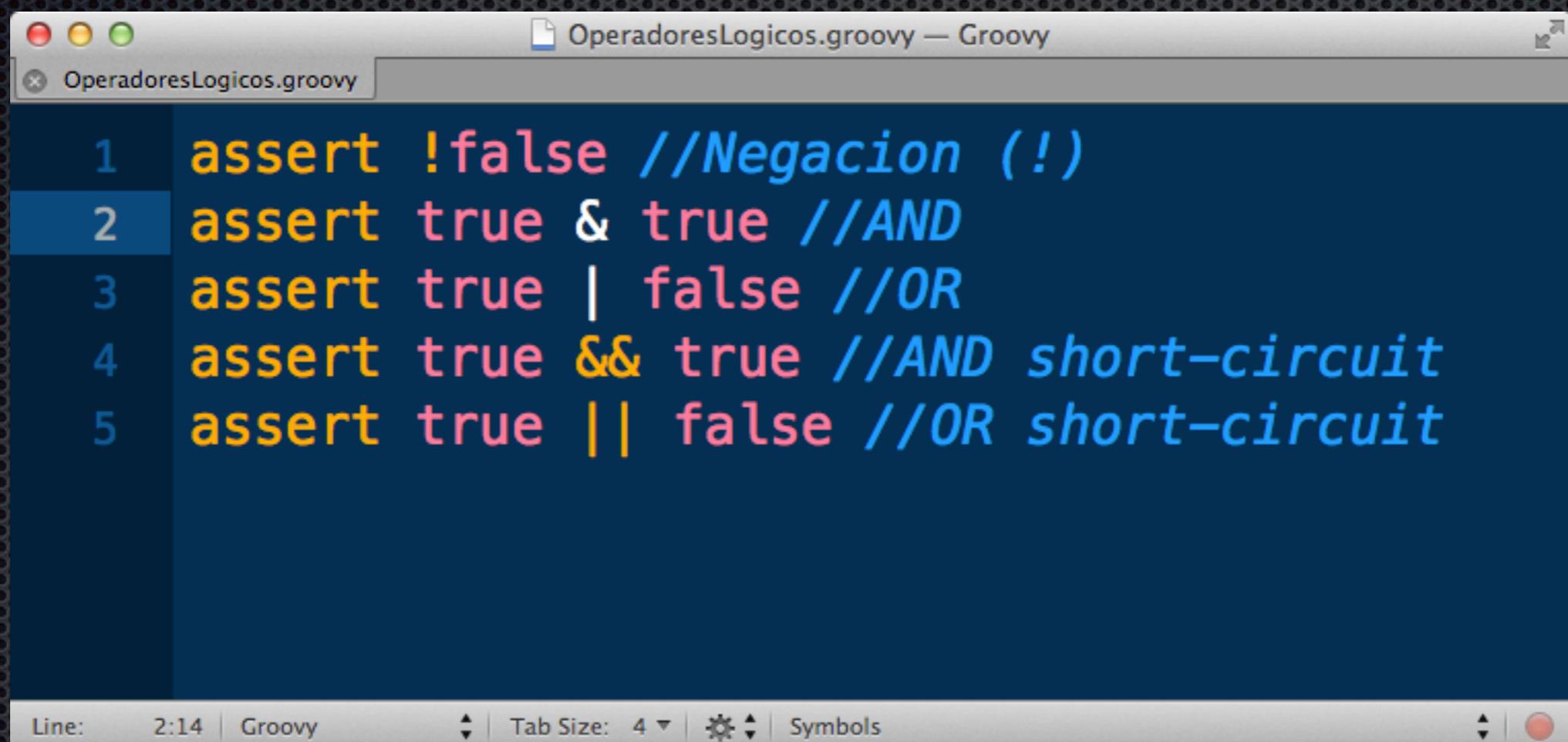


Operadores relacionales

```
OperadoresRelacionales.groovy — Groovy  
OperadoresRelacionales.groovy  
1 assert 1 + 2 == 3  
2 assert 3 != 4  
3  
4 assert -2 < 3  
5 assert 2 <= 2  
6 assert 3 <= 4  
7  
8 assert 5 > 1  
9 assert 5 >= -2  
Line: 6:14 | Groovy | Tab Size: 4 | Symbols
```



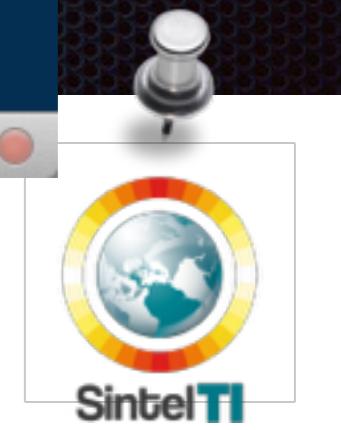
Operadores lógicos



A screenshot of a Groovy code editor window titled "OperadoresLogicos.groovy". The code editor displays the following Groovy script:

```
1 assert !false //Negacion (!)
2 assert true & true //AND
3 assert true | false //OR
4 assert true && true //AND short-circuit
5 assert true || false //OR short-circuit
```

The code editor interface includes a toolbar at the top with standard file operations, a status bar at the bottom showing "Line: 2:14 | Groovy", and a tab bar with multiple tabs. The status bar also includes settings for "Tab Size: 4" and "Symbols".



Operadores lógicos de corto circuito

```
OperadoresShortCircuit.groovy — Groovy
OperadoresShortCircuit.groovy

1▼ def a() {
2    println "en a()"
3    true
4▲ }
5▼ def b() {
6    println "en b()"
7    false
8▲ }
9 println a() | b()
10 println a() || b()

Line: 10:19 | Groovy | Tab Size: 4 | ☰ | b0
```



Operador safe-navigation

- Nos sirve para invocar de manera segura métodos sobre objetos que podrían ser nulos
- Una manera mucho más breve, sólo anteponiendo un signo de interrogación antes de la invocación de un método **?**.
- Incluso pueden anidarse, y si alguno de los objetos anidados es nulo la invocación no se efectuará



Operador safe-navigation

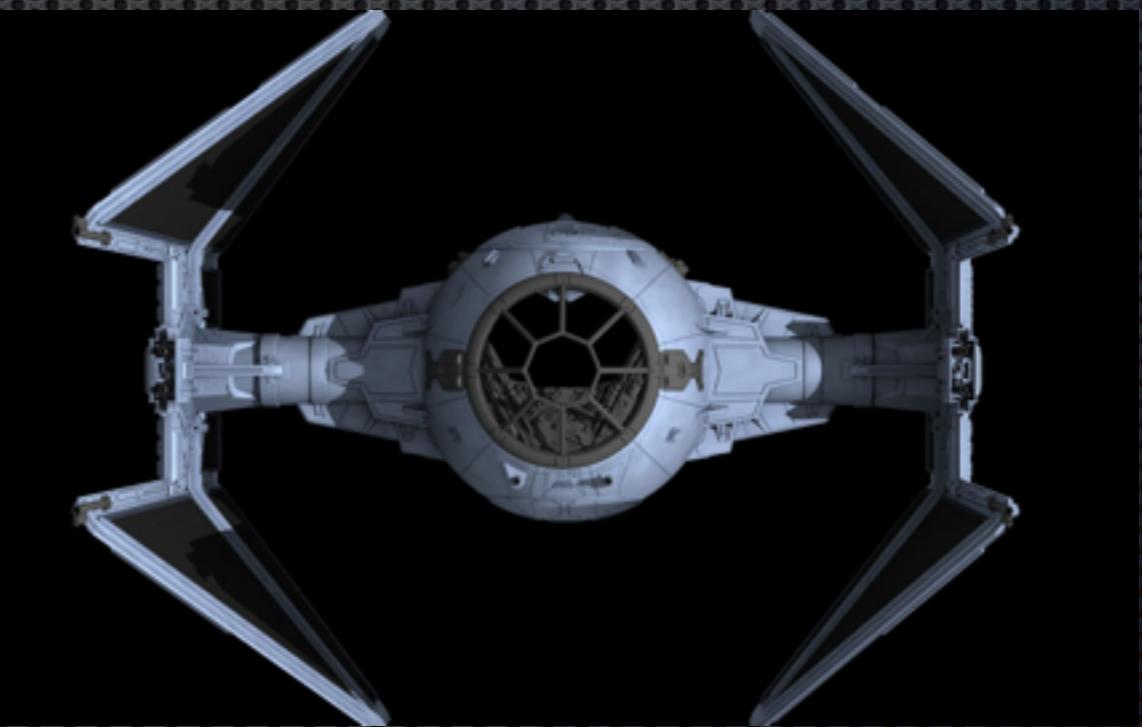
The screenshot shows a Groovy code editor window titled "SafeNavigation.groovy — Groovy". The code defines three classes: Empresa, Direccion, and Calle. The Empresa class has a Direccion attribute. The Direccion class has three attributes: calle, codigoPostal, and ciudad. The Calle class has two attributes: nombre and numero. A script at the bottom demonstrates the safe-navigation operator (?.) to access the calle attribute of the direccion attribute of an Empresa object.

```
1  class Empresa {  
2      def nombre  
3      Direccion direccion  
4  }  
5  class Direccion {  
6      Calle calle  
7      String codigoPostal  
8      String ciudad  
9  }  
10 class Calle {  
11     String nombre  
12     String numero  
13 }  
14  
15 def empresa = new Empresa()  
16  
17 //En Java  
18 if (empresa != null && empresa.direccion() != null && empresa.direccion().getCalle() != null){  
19     System.out.println empresa.direccion().getCalle().nombre  
20 }  
21 //Con safe-navigation de Groovy  
22 println empresa?.direccion?.calle?.nombre
```

Line: 14 | Groovy | Tab Size: 4 | numero

Operador spaceship

- $\langle \rangle$
- Comparación de dos variables o valores
- Devuelve -1 si el operando de la izquierda es menor, 0 si es igual o 1 si es mayor



Operador Spaceship

- Funciona sobre objetos que puedan ser comparables

```
OperadorSpaceship.groovy — Groovy
OperadorSpaceship.groovy

1 println 1 <=> 1
2 println 1 <=> 10
3 println 5 <=> 1
4
5 println 'Mexico' <=> 'Brasil'
6 println 'Mexico' <=> 'Alemania'
7 println 'Mexico' <=> 'Holanda'
8 println 'Mexico' <=> 'USA'

Line: 8:27 | Groovy | Tab Size: 4 | Symbols
```



The Groovy Truth

- Booleanos - true o false
- Colecciones - si están vacías devuelven false
- Iteradores - si no quedan elementos devuelven false
- Mapas - si están vacíos devuelven false
- Strings - si están vacías devuelven false
- Números - un valor 0 devuelve false
- Objetos - referencias nulas devuelven false



The Groovy Truth



```
GroovyTruth.groovy — Groovy
GroovyTruth.groovy

1 def a = true
2 if (a){ println 'Valor boolean devuelve true o false como en cualquier lenguaje' }

3
4 def numeros = [5, 6, 7, 8, 9]
5 if (numeros){ println 'Una colección con elementos devuelve true' }

6
7 def letras = []
8 if (letras){ println 'Una colección vacía devuelve false' }

9
10 def cadena = 'Hola mundo'
11 if (cadena){ println 'Una cadena con caracteres devuelve true' }

12
13 def cadenaVacia = ''
14 if (cadenaVacia){ println 'Una cadena vacía devuelve false' }

15
16 if (1){ println 'Cualquier numérico diferente de cero devuelve true' }
17
18 if (0){ println 'El valor numérico 0 devuelve false' }
19
20 if (new Object()){ println 'Un objeto instanciado devuelve true' }

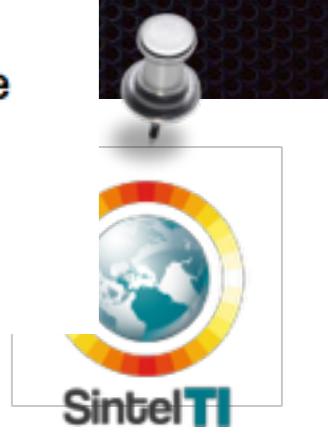
21
22 def obj = null
23 if (obj){ println 'Una referencia null devuelve false' }

Line: 14:62 | Groovy | Tab Size: 4 | cadenaVacia
```

Tipado opcional

- Mencionamos que el tipado en Groovy es opcional

| Statement | Type of Value | Comment |
|------------------|----------------------|---|
| def a = 1 | java.lang.Integer | Implicit typing |
| def b = 1.0f | java.lang.Float | |
| int c = 1 | java.lang.Integer | Explicit typing using the Java primitive type names |
| float d = 1 | java.lang.Float | |
| Integer e = 1 | java.lang.Integer | Explicit typing using reference type names |
| String f = '1' | java.lang.String | |



Tipado dinámico

- El tipado dinámico también puede confundirse con otra característica de los lenguajes: **type inference (inferencia de tipos)**
- El dinamismo de Groovy no solamente evita que escribamos el tipo de una variable, sino también se encarga de realizar las conversiones necesarias en caso de cambiarla de tipo



Tipado dinámico

```
TiposDeDatos.groovy — Groovy  
TiposDeDatos.groovy  
1 (60 * 60 * 24 * 365).toString(); // invalido en Java  
2  
3 int segundosPorAnio = 60 * 60 * 24 * 365;  
4 segundosPorAnio.toString(); // no valido en Java  
5  
6 new Integer(segundosPorAnio).toString();  
7 assert "abc" - "a" == "bc" // no valido en Java  
  
Line: 7:48 | Groovy | Tab Size: 4 | segundosPorAnio
```



Groovy GDK

- Groovy añade extensiones a lo que ya existe en Java
- A esta serie de adiciones se le llama GDK
- Algunas adiciones son directamente a la clase Object
- Otras son directamente a las clases que ya existen en Java para manejo de archivos, cadenas o fechas



Adiciones GDK a Object

- any { } - true si alguno de los items coincide
- collect { } - regresa una lista de items
- each { } - se ejecuta para cada elemento
- eachWithIndex { } - igual que each pero con índice
- every { } - true si todos los items coinciden
- find { } - regresa el primer item coincidente
- findAll { } - regresa todos los items coincidentes



Adiciones GDK a Object



```
GDKObject.groovy — Groovy
```

```
1 def numeros = [5, 7, 9, 12, 15]
2
3 println numeros.any { it % 2 == 0 } //true si ALGUN elemento
. es par
4
5 println numeros.every { it > 4 } //true si TODOS son mayores a
. 4
6
7 println numeros.findAll { it in 6..10 } //devuelve una lista
. de los elementos que cumplen con la condicion
8
9 println numeros.collect { 'Hola ' + ++it } //ejecuta la
. expresion sobre cada uno de los elementos y devuelve una lista
. con los nuevos valores
10
11 numeros.eachWithIndex{ num, id -> println "$id: $num" }
. //imprime cada numero y su indice
```

Line: 9 | Groovy | Tab Size: 4 | numeros

Más adiciones GDK

- **File.deleteDir()** - A diferencia de File.delete() es posible eliminar directorios no vacíos
- **File.getText()** - Obtiene el contenido de un archivo de texto
- **File.readLines()** - Similar a getText() pero devuelve cada línea en una lista de Strings
- **File.setText()** - Escribe contenido en el archivo, eliminando lo que tiene actualmente



Más adiciones GDK

- **File.write()** - Sinónimo de File.setText()
- **File.append()** - Añade contenido a un archivo
- **File.size()** - Devuelve el tamaño total de caracteres del archivo
- **Iteraciones con archivos o directorios** - eachDir(), eachDirRecurse(), eachDirMatch(), eachFile(), eachFileMatch(), eachFileRecurse(), eachLine()



Más adiciones GDK

- **String.count(String)** - Cuenta el número de ocurrencias de una cadena dentro de otra
- **String.center(int)** - Centra una cadena entre el número de caracteres indicado
- **Test de conversión de String hacia otro tipo** -
isBigDecimal(), isBigInteger(), isDouble(), isLong(),
isFloat(), isNumber()
- **String.readLines()**



Más adiciones al GDK

- **String.reverse()**
- **String.toList()**
- **String.leftShift()** - Permite concatenar mediante <<
- **String.minus()**
- **String.execute()** - Ejecuta la cadena como un comando en el sistema operativo actual



Más adiciones al GDK

- **Number.downto()**
- **Number.upto()**
- **Number.times()**
- **List.first(), List.last(), List.head(), List.tail()**
- **List.reverse()**
- **List.reverseEach()**



Más adiciones al GDK

- **List.execute()**
- **Date.clearTime(), Calendar.clearTime()**
- **Date.format(), Calendar.format()**



Tópicos Avanzados

El verdadero poder de Groovy

Named Parameters

- Groovy provee un mecanismo para poder definir las propiedades de una clase o método nombrando directamente sus propiedades y asignando sus valores
- Similar a como lo haría un constructor con parámetros, pero con la posibilidad de elegir el orden



Named Parameters

```
NamedParameters.groovy — Groovy  
NamedParameters.groovy  
1  class Auto {  
2      String marca  
3      int modelo  
4      String submarca  
5      String version  
6      String color  
7      String transmision  
8      int hp  
9      int noPuertas  
10 }  
11  
12 def jetta = new Auto(submarca: 'Jetta',  
13                         marca: 'Volkswagen',  
14                         modelo: 2010,  
15                         version: 'Clasico',  
16                         color: 'Fucsia',  
17                         transmision: 'Tiptronic',  
18                         noPuertas: 5)  
19  
20 println jetta.marca  
21 println jetta.submarca  
22 println jetta.modelo  
Line: 22:21 | Groovy | Tab Size: 4 | Symbols
```



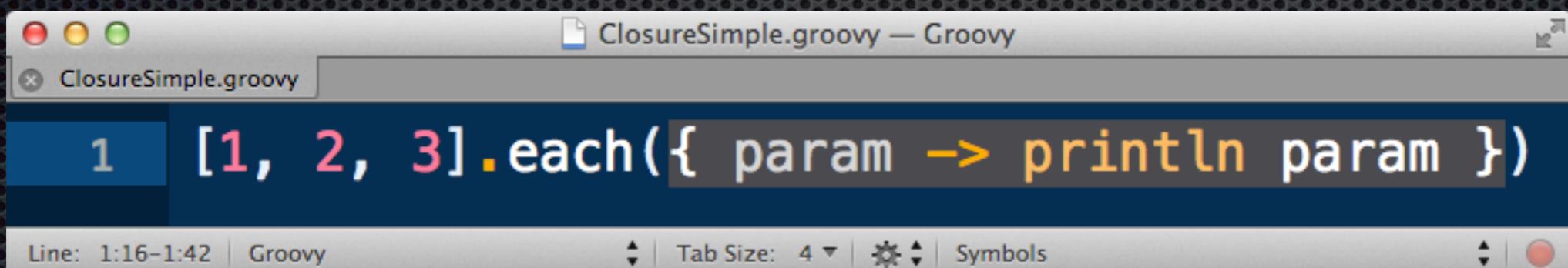
Closures

- Una de las características ‘funcionales’ de Groovy son los **Closures**
- Por definición un **closure** es una función anónima, es decir un bloque de código que puede ser pasado como parámetro a un método o asignado a una variable



Closures

- Aceptan uno o más parámetros

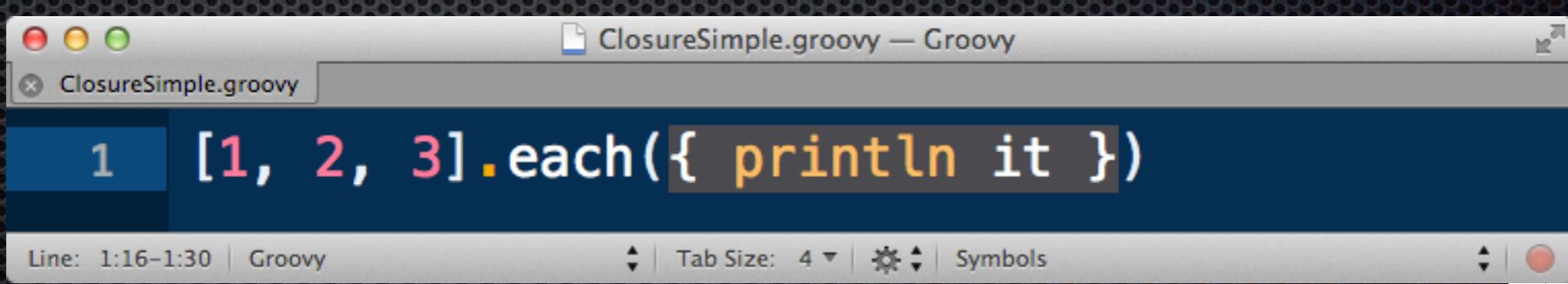


A screenshot of a Groovy code editor window titled "ClosureSimple.groovy". The code in the editor is:

```
1 [1, 2, 3].each({ param -> println param })
```

The code editor interface includes standard window controls (red, yellow, green buttons), a tab bar with "ClosureSimple.groovy", status bar with "Line: 1:16-1:42 | Groovy", and a toolbar with "Tab Size: 4" and "Symbols" buttons.

- En Groovy, si no ponemos explícitamente el parámetro
Groovy lo nombrará como 'it'



A screenshot of a Groovy code editor window titled "ClosureSimple.groovy". The code in the editor is:

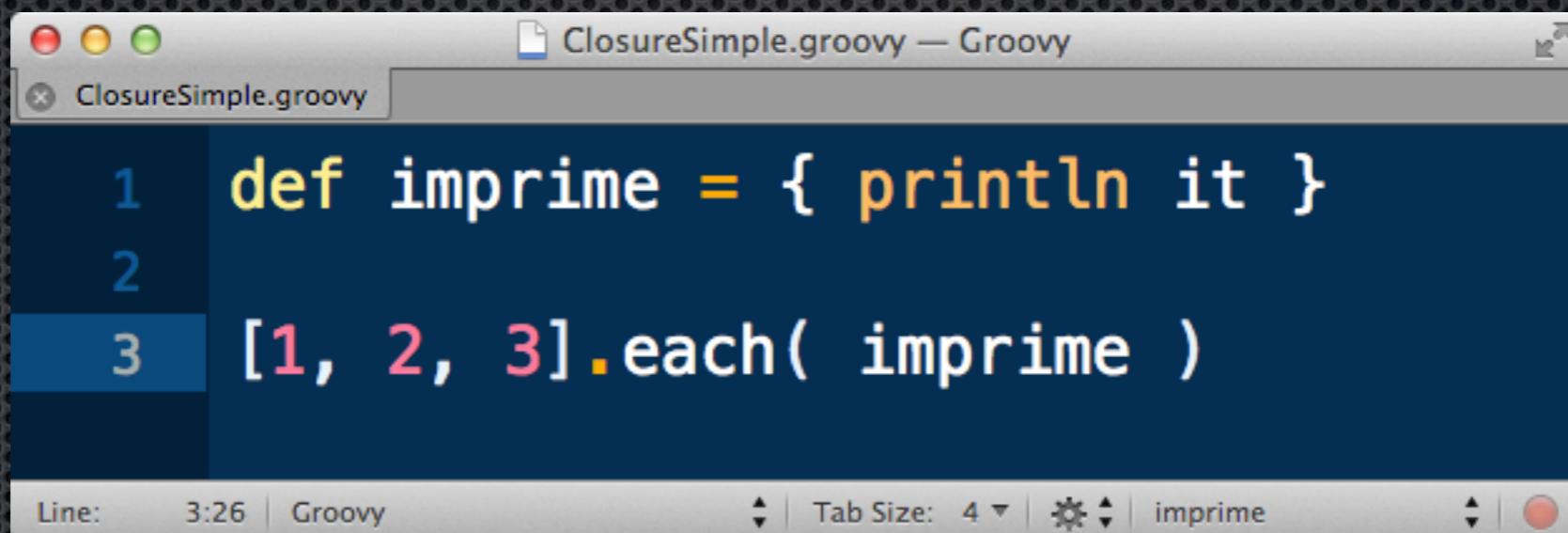
```
1 [1, 2, 3].each({ println it })
```

The code editor interface is identical to the previous screenshot, showing the same window controls, tab bar, status bar, and toolbar.



Closures

- Mencionamos que los closures son bloques de código que pueden ser asignados a una variable



A screenshot of a Groovy code editor window titled "ClosureSimple.groovy". The code in the editor is:

```
1 def imprime = { println it }
2
3 [1, 2, 3].each( imprime )
```

The code defines a closure named "imprime" that prints its argument. It then uses the ".each()" method on a list containing 1, 2, and 3, passing the closure "imprime" as an argument.



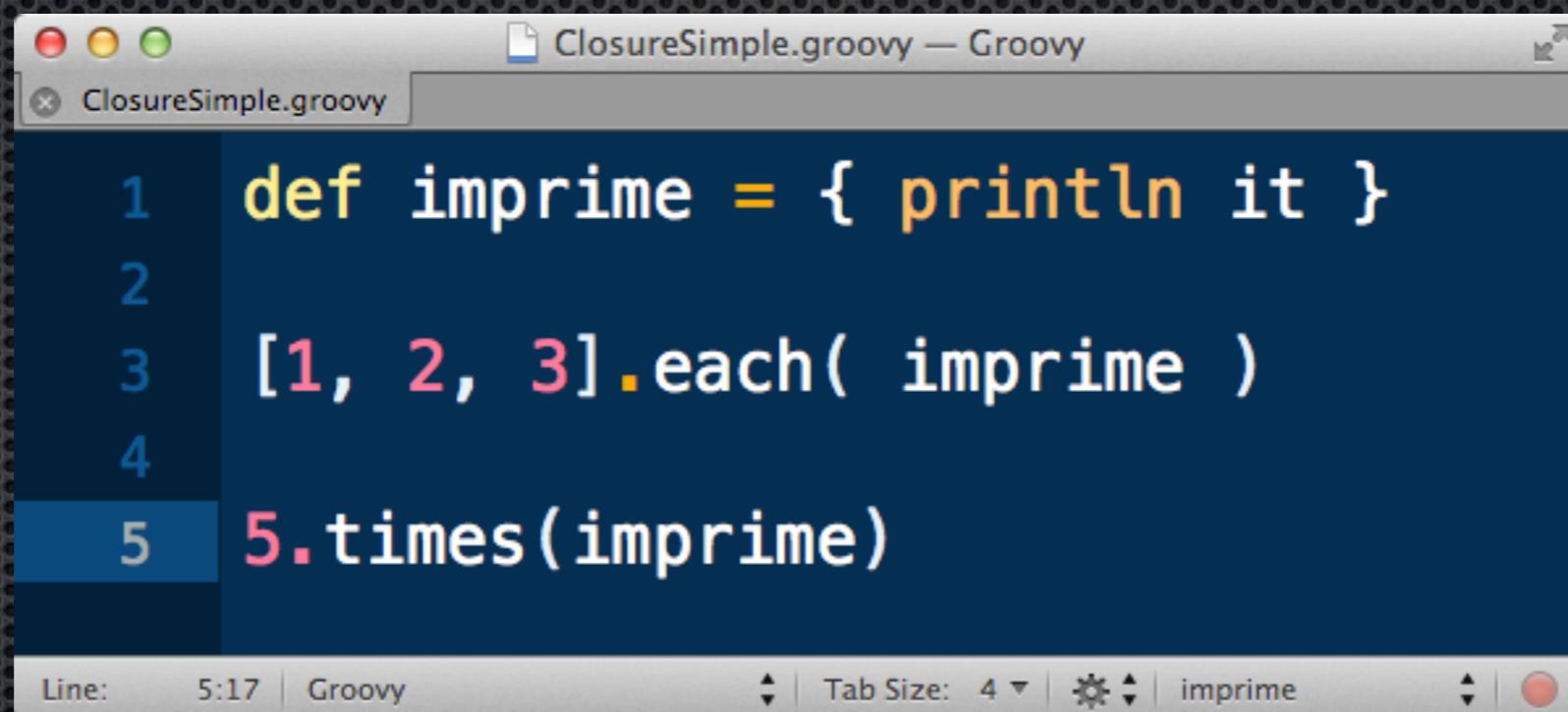
Closures

- Una ventaja de los closures es poder pasar un bloque de código a otra función externa por medio de sus parámetros
- Por ejemplo el método times() acepta un closure, esto hace que podamos pasarle un closure a ejecutar como si ese bloque de código estuviera dentro de times()



Closures

- El closure es simplemente un bloque que ejecuta con los valores dependiendo de su entorno



The screenshot shows a Groovy code editor window titled "ClosureSimple.groovy — Groovy". The code is as follows:

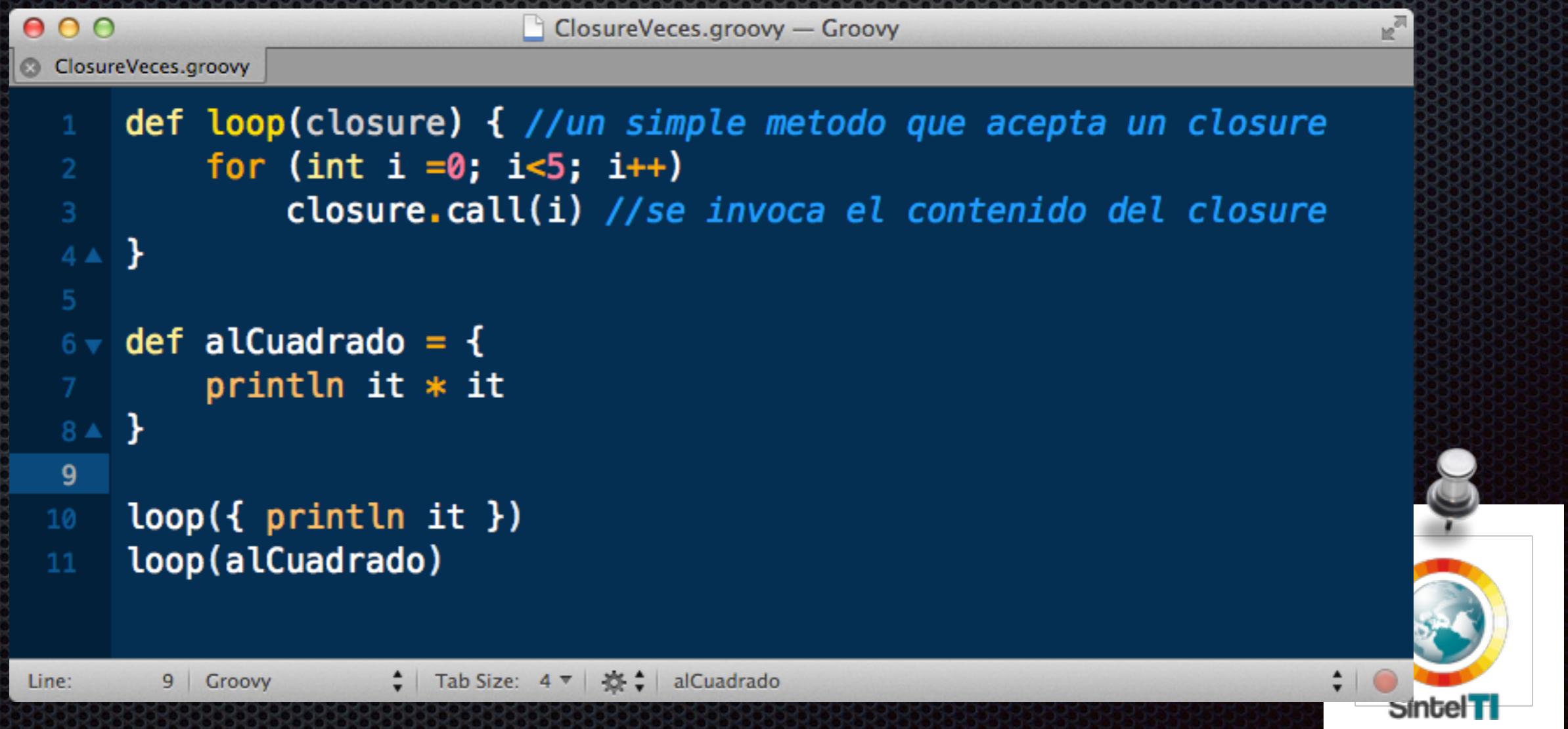
```
1 def imprime = { println it }
2
3 [1, 2, 3].each( imprime )
4
5 5.times(imprime)
```

The code defines a closure named "imprime" that prints its argument. It then uses the ".each()" method on a list [1, 2, 3] to call "imprime" on each element. Finally, it calls "5.times(imprime)" to execute the closure five times.



Closures

- Para mejor comprensión escribiremos un método parecido a times() que acepte un closure



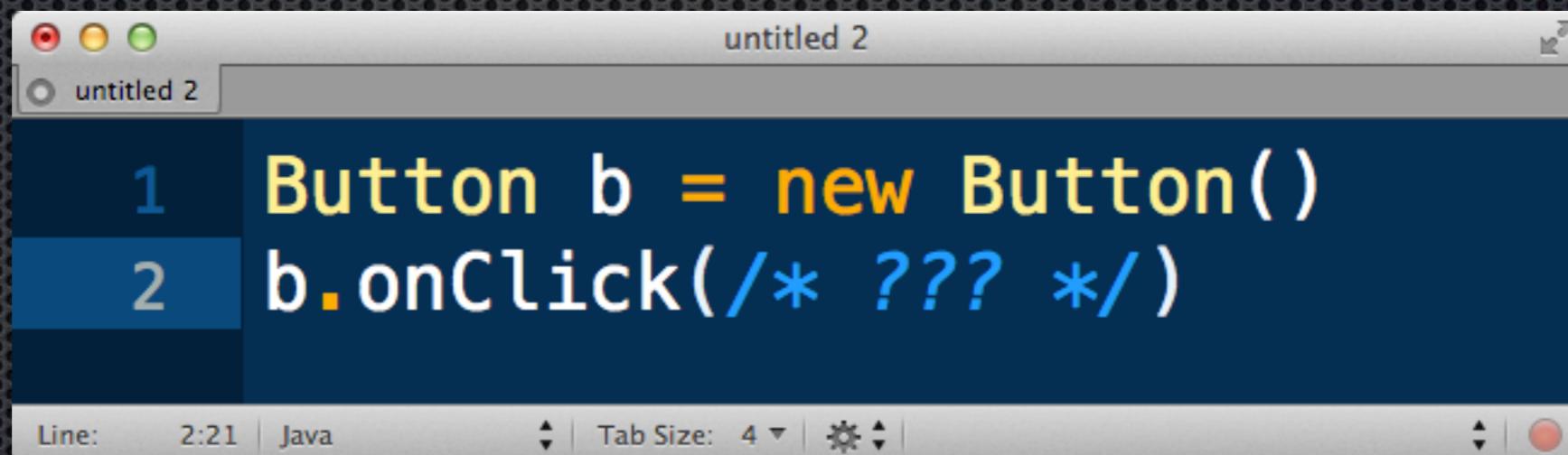
The screenshot shows a Groovy code editor window titled "ClosureVeces.groovy". The code is as follows:

```
1 def loop(closure) { //un simple metodo que acepta un closure
2     for (int i =0; i<5; i++)
3         closure.call(i) //se invoca el contenido del closure
4 }
5
6 def alCuadrado = {
7     println it * it
8 }
9
10 loop({ println it })
11 loop(alCuadrado)
```

The code defines a method `loop` that takes a closure as an argument. It then iterates from 0 to 4, calling the closure with each value of `i`. Below this, a closure `alCuadrado` is defined that prints the square of its input. Finally, the `loop` method is called twice: first with a closure that prints each number, and second with the `alCuadrado` closure.

Closures

- Otro uso que ejemplifica las ventajas de los closures son los métodos que simbolizan acciones, pero dependen de que alguien más las defina



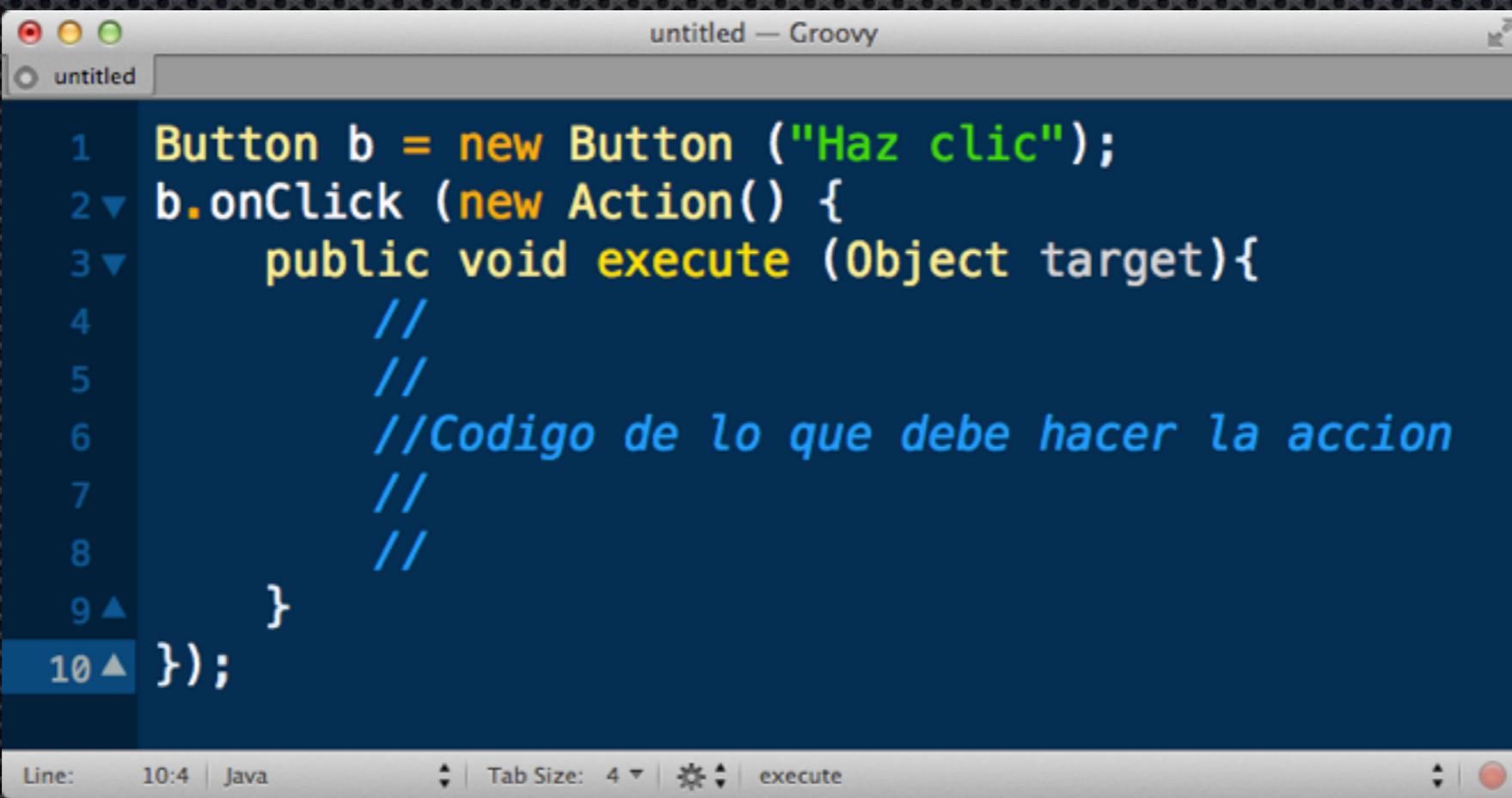
```
untitled 2
1 Button b = new Button()
2 b.onClick(/* ??? */)
```

The screenshot shows a Java code editor window titled "untitled 2". It contains two lines of Java code. The first line defines a button named "b" using the "new" keyword. The second line is a call to the "onClick" method of the button "b", followed by a comment block starting with "/* ??? */". The code editor has a dark blue theme. The status bar at the bottom shows "Line: 2:21 | Java" and "Tab Size: 4".



Closures

- En este escenario, Java se apoya de clases anónimas para cumplir este propósito, aunque no se ve natural



The screenshot shows a Groovy code editor window titled "untitled — Groovy". The code is as follows:

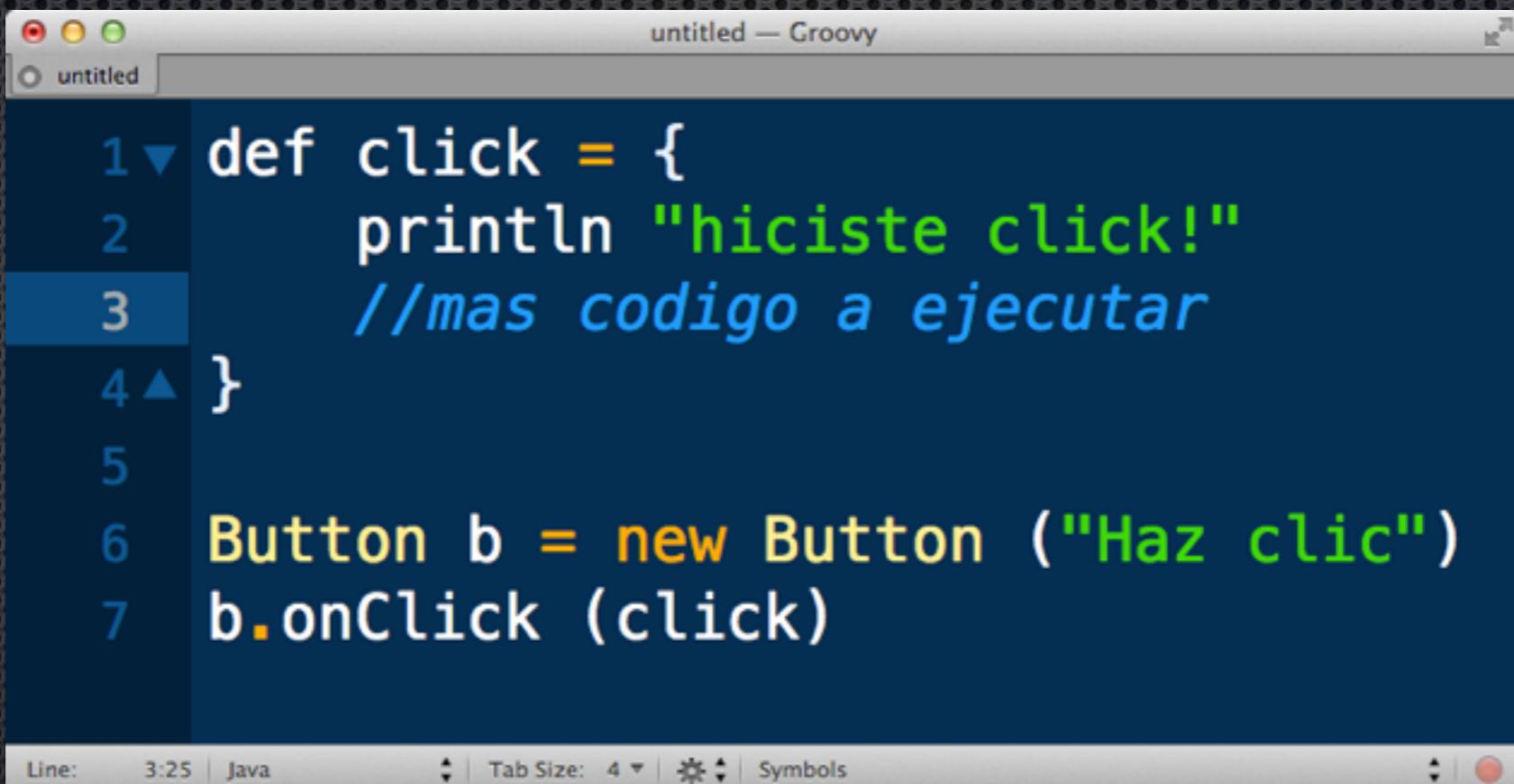
```
1 Button b = new Button ("Haz clic");
2 b.onClick (new Action() {
3     public void execute (Object target){
4         //
5         //
6         //Codigo de lo que debe hacer la accion
7         //
8         //
9     }
10});
```

The code creates a button and sets its click action to a new anonymous class that prints a message. The code is annotated with comments in Spanish: //Codigo de lo que debe hacer la accion.



Closures

- Con closures es fácil pasar como parámetro un bloque de código a otro método para que sea ejecutado



The screenshot shows a Groovy code editor window titled "untitled — Groovy". The code is as follows:

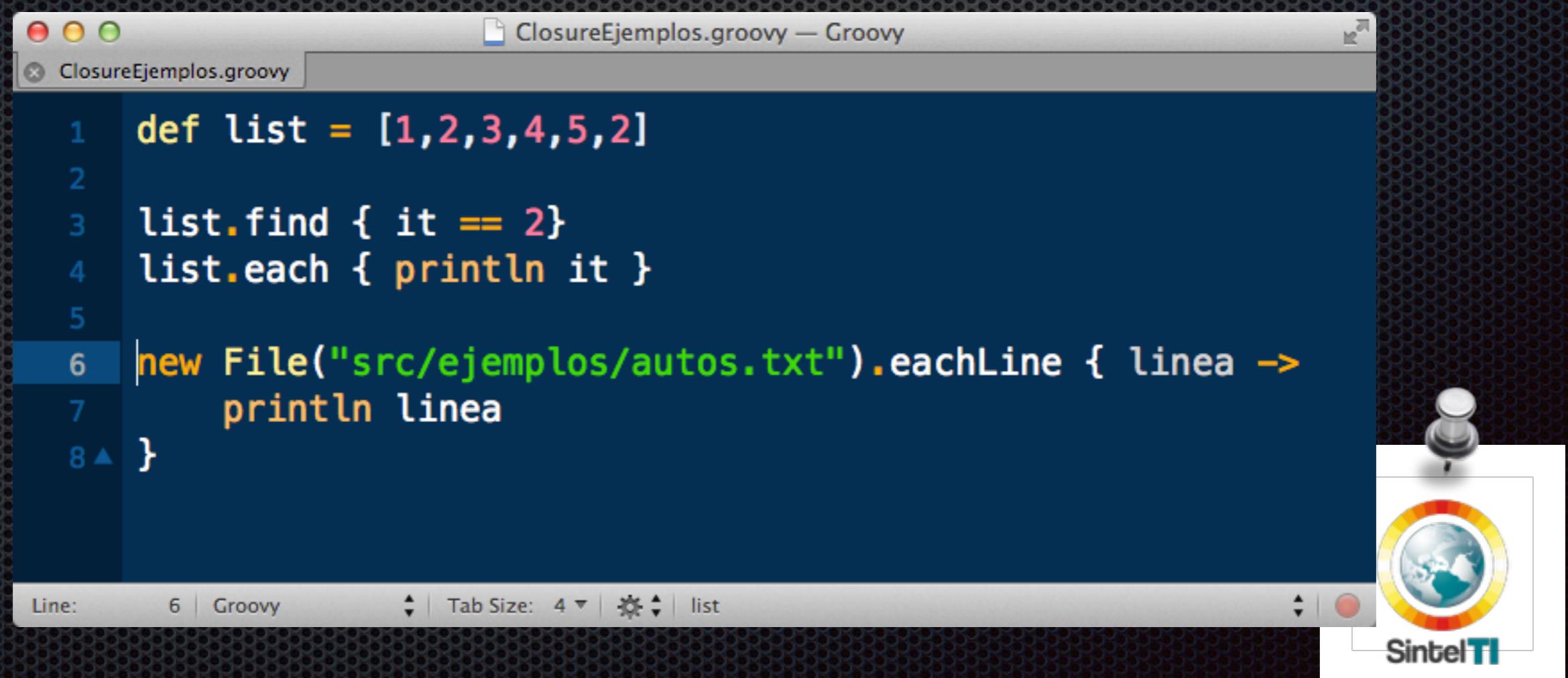
```
1 def click = {  
2     println "hiciste click!"  
3     //mas codigo a ejecutar  
4 }  
5  
6 Button b = new Button ("Haz clic")  
7 b.onClick (click)
```

The code defines a closure named "click" that prints "hiciste click!" and contains a comment "//mas codigo a ejecutar". It then creates a button "b" and sets its "onClick" event to trigger the "click" closure.



Closures

- Hemos estado utilizando closures en algunos de los ejemplos que se han expuesto



The screenshot shows a Groovy code editor window titled "ClosureEjemplos.groovy". The code is as follows:

```
1 def list = [1,2,3,4,5,2]
2
3 list.find { it == 2}
4 list.each { println it }
5
6 new File("src/ejemplos/autos.txt").eachLine { linea ->
7     println linea
8 }
```

The code uses several Groovy features, including closures. Line 6 demonstrates a closure passed as an argument to the `eachLine` method of a `File` object. The closure takes a single parameter `linea` and prints its value.