Groovy def Keyword

**1. Overview**

In this quick tutorial, we'll explore the concept of the *def* keyword in Groovy. It provides an optional typing feature to this dynamic JVM language.

**2. Meaning of the *def* Keyword**

**The *def* keyword is used to define an untyped variable or a function in Groovy, as it is an optionally-typed language.**

When we're unsure of the type of a variable or field, we can leverage *def* to let Groovy decide types at runtime based on the assigned values:

**def** firstName = "Samwell"

**def** listOfCountries = ['USA', 'UK', 'FRANCE', 'INDIA']

Here, *firstName* will be a *String*, and *listOfCountries* will be an *ArrayList*.

We can also use the *def* keyword to define the return type of a method:

**def** multiply(x, y) {

**return** x\*y

}

Here, *multiply* can return any type of object, depending on the parameters we pass to it.

**3. *def* Variables**

Let's understand how *def* works for variables.

When we use *def* to declare a variable, Groovy declares it as a *NullObject*and assign a *null* value to it:

**def** list

**assert** list.getClass() == org.codehaus.groovy.runtime.NullObject

**assert** list.is(null)

The moment we assign a value to the *list*, Groovy defines its type based on the assigned value:

list = [1,2,4]

**assert** list **instanceof** ArrayList

Let's say that we want to have our variable type dynamic and change with an assignment:

**int** rate = 20

rate = [12] // GroovyCastException

rate = "nill" // GroovyCastException

We cannot assign *List* or *String* to an *int* typed variable, as **this will throw a runtime exception**.

So, to overcome this problem and invoke the dynamic nature of Groovy, we'll use the *def* keyword:

**def** rate

**assert** rate == null

**assert** rate.getClass() == org.codehaus.groovy.runtime.NullObject

rate = 12

**assert** rate **instanceof** Integer

rate = "Not Available"

**assert** rate **instanceof** String

rate = [1, 4]

**assert** rate **instanceof** List

**4. *def* Methods**

The *def* keyword is further used to define the dynamic return type of a method. This is handy when we can have different types of return values for a method:

**def** divide(**int** x, **int** y) {

**if** (y == 0) {

**return** "Should not divide by 0"

} **else** {

**return** x/y

}

}

**assert** divide(12, 3) **instanceof** BigDecimal

**assert** divide(1, 0) **instanceof** String

We can also use *def* to define a method with no explicit returns:

**def** greetMsg() {

println "Hello! I am Groovy"

}

**5. *def* vs. Type**

Let's discuss some of the best practices surrounding the use of *def*.

Although we may use both *def* and type together while declaring a variable:

**def** **int** count

**assert** count **instanceof** Integer

**The *def* keyword will be redundant there, so we should use either *def* or a type.**

Additionally, we should **avoid using *def* for untyped parameters** in a method.

Therefore, instead of:

**void** multiply(**def** x, **def** y)

We should prefer:

**void** multiply(x, y)

Furthermore, we should **avoid using *def* when defining constructors.**

**6. Groovy *def* vs. Java *Object***

As we've seen most of the features of the *def* keyword and its uses through examples, we might wonder if it's similar to declaring something using the *Object* class in Java. Yes, ***def* can be considered similar to *Object***:

**def** fullName = "Norman Lewis"

Similarly, we can use *Object* in Java:

**Object** fullName = "Norman Lewis";

**7. *def* vs. *@TypeChecked***

As many of us would be from the world of strictly-typed languages, we may wonder **how to force compile-time type checking in Groovy**. We can easily achieve this using the *@TypeChecked* annotation.

For example, we can use *@TypeChecked* over a class to enable type checking for all of its methods and properties:

@TypeChecked

**class** **DefUnitTest** **extends** **GroovyTestCase** {

**def** multiply(x, y) {

**return** x \* y

}

**int** divide(**int** x, **int** y) {

**return** x / y

}

}

Here, the *DefUnitTest* class will be type checked, and **compilation will fail due to the *multiply* method being untyped**. The Groovy compiler will display an error:

[Static type checking] - Cannot find matching method java.lang.Object#multiply(java.lang.Object).

Please check if the declared type is correct and if the method exists.

So, **to ignore a method, we can use *TypeCheckingMode.SKIP*:**

@TypeChecked(TypeCheckingMode.SKIP)

**def** multiply(x, y)