



```
String name;

void sayHello() {
    println("Hello "+getName()+"!");
}

static void main(String[] args) {

    Hello hello = new Hello();
    hello.setName("world");
    hello.sayHello();
}
}
```

#### Listing 4. Further Shortening the Program

There's also a difference in how Groovy deals with String objects—using double quotation marks with strings allows for variable substitution. There are also strings with single quotation marks, as shown in Listing 5, which do not:

```
class Hello {

    String name;

    void sayHello() {
        println("Hello $name!");
    }

    static void main(String[] args) {

        Hello hello = new Hello();
        hello.setName('world');
        hello.sayHello();
    }
}
```

#### Listing 5. Strings with Double Quotation Marks and Single Quotation Marks

Groovy also allows dot notation for getting and setting fields of classes, just like Java, as shown in Listing 6. Unlike Java, this will actually go through the getter/setter methods (which, you'll recall, are automatically generated in our current example):

```
class Hello {

    String name;

    void sayHello() {
        println("Hello $name!");
    }

    static void main(String[] args) {

        Hello hello = new Hello();
        hello.name = 'world';
        hello.sayHello();
    }
}
```

#### Listing 6. Dot Notation

Typing information is also optional; instead of specifying a type, you can just use the keyword `def`, as shown in Listing 7. Use of `def` is mandatory for methods, but it is optional for parameters on those methods. You should also use `def` for class and method variables. While we're at it, let's take out those semicolons; they're optional in Groovy.

```
class Hello {

    def sayHello(name) {
        println("Hello $name!")
    }

    static def main(args) {
        Hello hello = new Hello()
        def name = 'world'
        hello.sayHello(name)
    }
}
```

#### Listing 7. The `def` Keyword

OK, that's nice, but we can make this even shorter. Because Groovy is a scripting language, there's automatically a wrapping class (called `Script`, which will become very important to us later). This wrapping class means that we can get rid of our own wrapping class, as well as the `main` method, like so:

```
def sayHello(name) {
    println("Hello $name!")
}
def name = 'world'
sayHello(name)
```

#### Using an Array

So I started with a simple Java program (which also worked in Groovy) and slowly stripped out the cruft. Now, let's add a little change to use an array. Since Groovy is roughly a superset of Java, you might be tempted to do something like this:

```
def sayHello(name) {
    println("Hello $name!")
}

String[] names = {"SintSi", "Kaitlyn", "Keira"} for (String name : names) {    sayHello(name)
}
```

But this won't work. This is one place where Java syntax differs from Groovy's. To create a static array, you'd instead do this:

```
def sayHello(name) {
    println("Hello $name!")
}

String[] names = ["SintSi", "Kaitlyn", "Keira"]

for (String name : names) {
    sayHello(name)
}
```

```
}
```

As before, we can eliminate the type declarations:

```
def sayHello(name) {
    println("Hello $name!")
}

def names = ["SintSi", "Kaitlyn", "Keira"]

for (def name : names) {
    sayHello(name)
}
```

But a more Groovy way of doing the same thing would be to use the `in` keyword:

```
def sayHello(name) {
    println("Hello $name!")
}

def names = ["SintSi", "Kaitlyn", "Keira"]

for (name in names) {
    sayHello(name)
}
```

Note that under the hood, this code is no longer creating an array; rather, Groovy is (invisibly) creating an `ArrayList`. This gives us a number of new options, for instance, sorting, as shown in Listing 8:

```
def sayHello(name) {
    println("Hello $name!")
}

def names = ["SintSi", "Kaitlyn", "Keira"]

names.sort()

for (name in names) {
    sayHello(name)
}
```

#### Listing 8. Sorting

You can also add entries to and remove entries from the list, as shown in Listing 9:

```
def sayHello(name) {
    println("Hello $name!")
}

def names = ["SintSi", "Kaitlyn", "Keira"]

names += 'Jim'
names -= 'SintSi'

names.sort()

for (name in names) {
    sayHello(name)
}
```

#### Listing 9. Adding or Removing Entries

But this still isn't the way that many Groovy coders would do it. They'd probably use the built-in method `each`, which takes a closure as an argument. If you aren't already familiar with closures (I knew them already from JavaScript), they're similar in many ways to method pointers. And you might as well start learning about them since they're [coming to Java](#).

To use a closure, you define it using enclosing curly braces, and you can call it with the `call` method, like so:

```
def sayHello(name) {
    println("Hello $name!")
}

def clos = {name -> sayHello(name)}
clos.call('world')
```

This will print "Hello world!" Note that if you're trying this in `groovyConsole`, you'll see an additional value at the end, because Groovy scripts always return the last value as the return value, and `groovyConsole` is showing you the value of the `names ArrayList`. In this example, the `name ->` preamble defines a single parameter that the closure takes.

Now let's use the closure with the `each` method, as shown in Listing 10:

```
def sayHello(name) {
    println("Hello $name!")
}

def names = ["SintSi", "Kaitlyn", "Keira"]

names += 'Jim'
names -= 'SintSi'
names.sort()

def clos = {name -> sayHello(name)}
names.each(clos)
```

#### Listing 10. Using the Closure with the `each` Method

Under the covers, `names.each` is iterating through the collection and passing each value to the closure as the first parameter, just as in our previous example. We can simplify this by creating the closure in-place. And since in Groovy, method parentheses are optional when the method takes one or more parameters, we can use the format shown in Listing 11, which is pretty darn readable:

```
def sayHello(name) {
    println("Hello $name!")
}

def names = ["SintSi", "Kaitlyn", "Keira"]

names += 'Jim'
names -= 'SintSi'
names.sort()
names.each {name -> sayHello(name)}
```

#### Listing 11. Parentheses Are Optional for Some Methods

Chat

One more thing: By default, the first parameter of a closure is named `it`. So, you could instead use `that`, as shown in Listing 12:

```
def sayHello(name) {
    println("Hello $name!")
}

def names = ["SintSi", "Kaitlyn", "Keira"]

names += 'Jim'
names -= 'SintSi'
names.sort()

names.each {sayHello(it)}
```

**Listing 12. Final Groovy Program**

**Conclusion**

That's it for now. If you've followed along so far, you've gotten more than enough Groovy under your belt to be able to say, "Sure, more or less," when someone asks, "Do you know any Groovy?"

**See Also**


[Jim Driscoll's Blog](#)  
[Groovy](#)  
[Project Lambda](#)

**About the Author**

Jim Driscoll is a Senior Engineer at Oracle, working on the Groovy integration with Oracle Application Development Framework's model layer, ADFm. He's especially interested in DSLs and the creation of languages with simple techniques. He has over 30 years experience in computer software, but then, he started when he was still 12. He's programmed computers for the U.S. Air Force, a now-defunct systems integrator, and a medium-sized hardware vendor before joining Sun's JavaSoft division in 1996 and then the Oracle ADF team in 2010.

**Join the Conversation**

Join the Java community conversation on [Facebook](#), [Twitter](#), and the [Oracle Java Blog](#)!

 E-mail this page  Printer View

**Contact Us**

US Sales: +1.800.633.0738  
Global Contacts  
Support Directory  
Subscribe to Emails

**About Oracle**

Careers  
Company Information  
Social Responsibility  
Communities

**Downloads and Trials**

Java Runtime Download  
Java for Developers  
Software Downloads  
Try Oracle Cloud Free

**News and Events**

Acquisitions  
Blogs  
Events  
Newsroom

