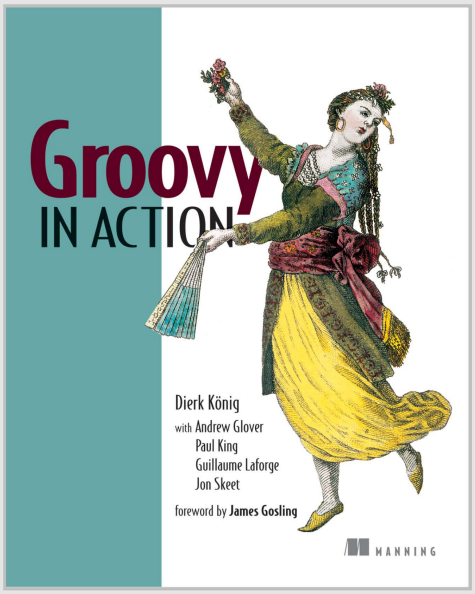

CHAPTER 9:

Working with the GDK

Excerpt From *Groovy in Action*
ISBN 1-932394-84-2

www.manning.com

9

Workingwith the GDK

Einstein argued that there must be simplified explanations of nature, because God is not capricious or arbitrary. No such faith comforts the software engineer.

—Fred Brooks

Learning a new programming language is a twofold task: learning the syntax and learning the standard library. Whereas learning the syntax is a matter of days and getting proficient with new language idioms may require a matter of a few weeks, working through a new library can easily take several months.

Luckily, no Java programmer needs to go through this time-consuming activity when learning Groovy. They already know most of the Groovy Standard Library, because that is the set of APIs that the Java Runtime provides. You can work with Groovy by solely using objects and methods as provided by the Java platform, although this approach doesn't fully leverage the power of Groovy.

Groovy extends (and in a few places modifies) the JRE to make it more convenient to work with, provide new dynamic features, and adapt the APIs to Groovy language idioms. The total of these extensions and modifications is called the GDK. Figure 9.1 gives the

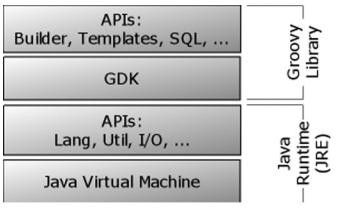

Figure 9.1 GDK's place in the Groovy architecture

architectural overview.

A big part of the GDK concerns the datatypes that Groovy supports at the language level, such as strings, numbers, lists, and maps. That part of the GDK was covered in part 1 of this book.

This chapter focuses on GDK capabilities that come from extending prominent JDK concepts such as `Object`, `File`, `Stream`, `Thread`, `Process`, and text processing with templates.

Let's start with `Object`, the most general and most important concept in the JDK, and see how Groovy further extends the concept with features for exploration and control.

9.1 Working with Objects

Java comes with a narrow API of 11 methods for its central abstraction `java.lang.Object`. These methods deal with the object lifecycle (`clone` , `finalize`), object equality (`equals`, `hashCode`), information (`toString`), self-reflection (`getClass`), and multithreading support (`notify`, `notifyAll`, three versions of `wait`).

Groovy adds much to the self-reflective and informational aspects of the API to better support live exploration of objects. It handles identity/equality differently and therefore needs to extend the respective API. It adds convenience methods to `Object` for the purpose of

making these methods available anywhere in the code. Finally, it adds collection-aware methods to `Object` that are useful when the object can be seen as some kind of collection even though it is not necessarily of static type `java.util.Collection` . This last category also includes the handling of object arrays.

We will go through these categories one by one, starting with self-reflective and informational methods.

9.1.1 Interactive objects

When working on a program, you often need to inspect your objects, whether for debugging, logging, or tracing purposes. In dynamic languages such as Groovy, this need is even greater, because you may work with your programming language in an interactive fashion, asking your objects about their state and capabilities to subsequently send them messages.

Object information in strings

Often, the first task is to ask an object for some general information about itself: `toString()` in Java parlance. Groovy adds two more methods of this kind:

- `dump` returns a description of the object's state, namely its fields and their values.
- `inspect` makes a best effort to return the object as it could appear in Groovy source code, with lists and maps in the format of their literal declaration. If it cannot do better, it falls back to `toString`.

Listing 9.1 shows these methods called on a string that contains a single newline character.

Listing 9.1 Usage of `dump` and `inspect`

```
def newline = "\n"

assert newline.toString() == "\n"

assert newline.dump() ==
'''<java.lang.String@a value=[
] offset=0 count=1 hash=10>'''

assert newline.inspect() == /"\n"/
```

Note how `inspect` returns a string that is equivalent to `newline`'s literal declaration: the characters backslash and `n` enclosed in double quotes (four characters total), whereas

`toString` returns only the newline character (one character). The `dump` of a string object may yield different results in other JVMs.

If these methods are not sufficient when working with Groovy interactively, remember that you can fire up the graphical `ObjectBrowser` via

```
groovy.inspect.swingui.ObjectBrowser.inspect(obj)
```

You have seen the `dump` method reveal the object's fields and their values. The same and more can be done with the object's properties.

Accessing properties

Remember that any Groovy object can be seen as a JavaBean, as you saw in section 7.4. You have already seen that its properties can be inspected with the `getProperties` method or the `properties` property. The method returns a read-only map of property names and their current values. During inspection, printing the whole map of properties is as easy as

```
println properties
```

or

```
println someObj.properties
```

When doing so, you may see more properties than you expected, because Groovy's class-generation mechanism introduces accessors for that object's *class* and `MetaClass` properties behind the scenes.[1]

Listing 9.2 shows property reflection in use. The example uses a class with a `first` property and a `second` read-only property that returns a derived value and is not backed by a field. A `third` property is only a field without accessor methods. The listing shows how to list all keys of that object's properties.

Of course, you can ask the map of properties for the value of a property either with the subscript operator or with the dot-propertyname syntax. This last option looks exactly the same as directly asking the object for the value of a property if its name is known at coding time. This raises the question of whether you can ask an object directly for a property value if its name is only known at runtime and resides in a variable. Listing 9.2 shows that you can do so by using the subscript operator directly on the object without the need for redirection over the `properties` map.

Because we know that the subscript operator is implemented via the `getAt` method, it would be surprising if the `putAt` method for subscript-assignment weren't implemented in

[1] It is planned to remove the appearance of `MetaClass` at this point, which may have happened by the time you read this. Listing 9.2 is also affected by this removal.

the same manner. Again, listing 9.2 shows that this works and allows us to assign a value to a property whose name is derived dynamically.

Listing 9.2 Reflecting on properties

```
class MyClass {
    def first = 1                    // read-write property
    def getSecond() { first * 2 }    // read-only  property
    public third = 3                 // public field property
}

obj = new MyClass()

keys = ['first','second','third',
        'class','metaClass']
assert obj.properties.keySet() == new HashSet(keys)

                                     | Properties
                                     | map
assert 1 == obj.properties['first']
assert 1 == obj.properties.first

assert 1 == obj.first

                                                    b Direct
assert 1 == obj['first']    // getAt('first')       access

one = 'first'
two = 'second'

                                                 c Dynamic
obj[one] = obj[two]         // putAt(one)        assignment

                                              d Field introspection
assert obj.dump() =~ 'first=2'
```

At **b** and **c**, you see that objects implement the `getAt` and `putAt` methods by default, such that the code appears to be accessing a map of properties as far as the subscript operator is concerned.

d shows a simple way of introspecting an object via the `dump` method. Because the `first` property is backed by a field of the same name, this field and its current value appear in the dump. Note that this field is private and wouldn't be visible otherwise. This trick is useful, especially in test cases.

NOTE

When working with Groovy code, you may also come across `Object`'s method `getMetaPropertyValues` . It is used internally with an object's meta information

and returns a list of `PropertyValue` objects that encapsulate the *name*, *type*, and *value* of a property.

Working with properties means working on a higher level of abstraction than working with methods or even fields directly. We will now take one step down and look at dynamic method invocation.

Invoking methods dynamically

In the Java world, methods (and fields) belong to `Class` rather than to `Object`. This is appropriate for most applications of reflection, and Groovy generally follows this approach. When you need information about an object's methods and fields, you can use the following GPath expressions:

```
obj.class.methods.name
obj.class.fields.name
```

This covers methods and fields as they appear in the bytecode. For dynamically added methods like those of the GDK, Groovy's `MetaClass` provides the information:

```
obj.metaClass.metaMethods.name
```

NOTE

You can add a `.unique` or `.sort` to the preceding GPaths to narrow down the list.

Groovy follows a slightly different approach than Java when it comes to invoking these methods dynamically. You saw the `invokeMethod` functionality for `GroovyObjects` in section 7.1.2. The GDK makes this functionality ubiquitously available on any (Java) object. In other words, a Groovy programmer can call

```
object.invokeMethod(name, params)
```

on any arbitrary object.

NOTE

This simple call is much easier than JDK reflection, where you need to go through the `Class` object to fetch a `Method` object from a list, invoke it by passing it your object, and take care of handling numerous exceptions.

Dynamic method invocation is useful when the names of the method or its parameters are not known before runtime.

Consider this scenario: You implement a persistence layer with *Data Access Objects* (DAOs), or objects that care for accessing persistent data. A `Person` DAO may have methods like `findAll`, `findByLastName`, `findByMaximum`, and so on. This DAO may be used in a web application setting as depicted in figure 9.2. It may respond to HTTP requests with request parameters for the type of *find* action and additional parameters. This calls for a way to dispatch from the request parameters to the method

Figure 9.2
Dispatching from HTTP request values to method calls in a Data Access Object

call.

Such a dispatch can be achieved with `if-else` or `switch` constructions. Listing 9.3 shows how dynamic method invocation[2] makes this dispatching logic a one-liner. Because this is only an illustrative example, we return the SQL statement strings from the DAO methods, not the `Person` objects as we would probably do in real DAOs.

Listing 9.3 Dynamic method invocation in DAOs

```
class PersonDAO {
    String findAll() {
        'SELECT * FROM Person'
    }
    String findByLastname(name) {
        findAll() + " WHERE p.lastname = '$name'"
    }
    String findByMaximum(attribute) {
```

[2]In real-life applications, you need to consider certain security constraints. This kind of dispatch should be used only when the user is allowed to safely use all available methods.

```
        findAll() + " WHERE $attribute = " +
        "SELECT maximum($attribute) FROM Person"
    }
}
dao = new PersonDAO()

action = 'findAll'  // some external input
params = [] as Object[]
assert dao.invokeMethod(action, params) == 'SELECT * FROM Person'
```

The `action` and `params` variables refer to external input, such as from an HTTP request. Note that this example is characteristic for a variety of applications. Almost every reasonably sophisticated client-server application has to deal with this kind of dispatching and can thus benefit from dynamic method invocation.

As a second example, an external configuration in plain-text files, tables, or XML may specify what action to take under certain circumstances. Think about *domain specific languages* (DSLs), the specification of a finite state machine, a workflow description, a rule engine, or a Struts configuration. Dynamic method invocation can be used to trigger such actions.

These scenarios are classically addressed with the Command pattern. In this pattern, dynamic invocation can fully replace simple commands that only encapsulate actions (that is, they don't encapsulate state or support additional functionality like *undo*).

While we are on this topic, dynamic invocation can be applied not only to methods but also to closures. In order to select a closure *by name*, you can store such closures in properties. An idiomatic variant of listing 9.3 could thus be

```
class PersonDAO {
    public findAll = {
        'SELECT * FROM Person'
    }
    // more finder methods as Closure fields ...
}
dao = new PersonDAO()

action = 'findAll'  // some external input
params = []
assert dao[action](*params) == 'SELECT * FROM Person'
```

Note that `findAll` is now a public field with a closure assigned to it, dynamically accessed via `dao[action]`. This dynamically accessed closure can be called in various ways. We choose the shortest variant of putting parentheses after the reference, including any arguments. The `*` spread operator distributes the arguments over the closure parameters (if—unlike `findAll`—the closure has any parameters).

These variants differ slightly in size where the closure variant is a bit shorter but may be less readable for the casual Groovy user. The closure variant additionally offers the possibility of changing the closure at runtime by assigning a new closure to the respective field. This can be handy in combination with the State or Strategy pattern.[3]

To further make programming Groovy a satisfying experience, the GDK adds numerous convenience methods to `Object`.

9.1.2 Convenient Object methods

How often have you typed `System.out.println` when programming Java? In Groovy, you can achieve the same result with `println`, which is an abbreviation for `this.println`; and because the GDK makes `println` available on `Object`, you can use this anywhere in the code. This is what we call a convenience method.

This section walks through the available convenience methods and their usage, as listed in table 9.1.

Table 9.1 `Object` convenience methods

Introduced `Object` method	Meaning
`is(other)`	Compare `Object` identities (references)
`isCase(caseValue, switchValue)`	Default implementation: equality
`obj.identity {closure}`	Call closure with object identity (delegate)
`print(), print(value), println(), println(value)`	`System.out.print…`
`printf(formatStr, value)` `printf(formatStr, value[])`	Java 5 `printf…`
`sleep(millis)` `sleep(millis) {onInterrupt}`	`static Thread.currentThread(). sleep(millis)`

[3] Erich Gamma et al, *Design Patterns: Elements of Reusable Object-Oriented Software*, (Addison-Wesley, 1995).

`use(categoryClass) {closure}` `use(categoryClassList) {closure}`	Use meta-methods as defined in `categoryClass` for the scope of the closure

Let's go through the methods.

Because Groovy uses the `==` operator for equality instead of identity checking, you need a replacement for the Java meaning of `==`. That is what the `is` method provides.

In Java:

```
if ( a == b ) { /* more code here */}
```

In Groovy:

```
if ( a.is(b)) { /* more code here */}
```

The `is` method saves you the work of comparing the `System.identityHashCode(obj)` of `a` and `b` manually.

The `isCase` method occurred often in the Groovy language description in part 1. For `Object`, the GDK provides a default implementation that checks for object equality. Note that this means you can use any (Java) object in a Groovy `grep` or `switch`:

```
switch(new Date(0)){
    case new Date(0) : println 'dates are equal'
}
```

The `identity` method calls the attached closure with the receiver object as the closure's delegate. This has an effect similar to that of the `WITH` keyword in Visual Basic. Use it when a piece of code deals primarily with only one object, like the following:

```
new Date().identity {
    println "$date.$month.$year"
}
```

The properties `date`, `month`, and `year` will now be resolved against the current date. Such a piece of code has by definition the smell of *inappropriate intimacy*.[4] This calls for making this closure a method on the receiver object (`Date`), which you can do in Groovy with the `use` method as covered in section 7.5.3.

The versions of `print` and `println` print to `System.out` by default, whereas `println` emits an additional line feed. Of course, you can still call these methods on any kind of `PrintStream` or `PrintWriter` to send your output in other directions.

The same is true for the `printf` method. It is based on Java's formatted print support, which has been available since Java 5 and (currently) works only if you run Groovy under

[4]Fowler et al, *Refactoring: Improving the Design of Existing Code* (Addison-Wesley, 1999).

Java 5 or higher. A `RuntimeException` is thrown otherwise. In terms of supported formatting features, we cannot present the full list here. Have a look at the Javadoc for class `java.util.Formatter`. The full description covers about 1,800 lines.

In Groovy, `printf` isn't as crucial as in other languages, because GStrings already provide excellent support at the language level and the `string` datatype provides the most common features of left and right padding and centering text. However, there are times when formatted output is more convenient to achieve with a *format* string, especially when the user should be able to configure the output format to their preferences. The following line

```
printf('PI=%2.5f and E=%2.5f', Math.PI, Math.E) // with Java 5 only !
```

prints

```
PI=3.14159 and E=2.71828
```

Note that we have used `printf` with three arguments, but because a format string may contain an arbitrary number of placeholders, `printf` supports an argument list of arbitrary length. It goes without saying that the number of additional arguments must match the number of placeholders in the format string, unless you explicitly specify the argument number to use in the format string. You can also provide a single argument of type `list`—for example, `[Math.PI,Math.E]`.

When working through the Formatter API documentation, you will notice some advanced topics around `printf`:

- *Conversions* apply when a placeholder and the corresponding argument are of different types.
- Placeholders can be prefixed with *n*`$` to map the placeholder to the *n*th argument in the list. This may get you in conflict with the GString meaning of `$`. Therefore, it's wise to use only single-quoted string literals as `printf` format strings.

The last convenience method in our list is `sleep`, which suspends the current thread for a given number of milliseconds. It enhances the JDK method `Thread.sleep` by automatically handling interruptions such that sleep is re-called until the given time has elapsed (as closely as the machine timer can tell). This makes the effective sleep time more predictable.

If you want to handle interruptions differently, you can attach a closure that is called when `sleep` encounters an `InterruptedException` .

With the `sleep` method, you can have some fun, as with the following example. Run it from the Groovy shell or console after predicting its output. Did you guess correctly what it does?

```
text = """
This text appears
slowly on the screen
```

```
as if someone would
tpye \b\b \b\b \b\b \bype it.
"""
for (c in text) {
    sleep 100
    print c
}
```

These are all methods that the GDK adds to every object for convenience. However, objects frequently come in a crowd. For such cases, the GDK provides methods to select them one-by-one, as shown in the next section.

9.1.3 *Iterative Object methods*

In the Java world, any *collection* (in the general meaning of the word) of objects can support inspection of its contained items by providing an `Iterator`, a separate object that knows how to walk through that collection. Oh—wait, sometimes an `Enumeration` is used instead. As a further inconsistency, `Iterator`s are not directly available on arrays and a lot of other common types.

Even if you are lucky and found how to get an `Iterator` object in the API documentation, you cannot do much more with it than use it in a procedural way like this:[5]

```
// Java !
for (Iterator collection.iterator(); iterator.hasNext(); ){
    MyClass obj = (MyClass) iterator.next();
    // do something with obj
}
```

Groovy instead provides a simple and consistent way of doing this:

```
collection.each { /* do something with it */}
```

Besides the simple `each` method, you can use any of the methods that are listed in table 9.2.

Table 9.2 Iterative `Object` methods

Return value	Method
`Boolean`	`any {closure}`
`List`	`collect {closure}`

[5]Java 5 allows a simpler syntax, but it has chosen to use a symbol instead of introducing a keyword. We end up with `for (MyClass obj : collection)`. Better, but not ideal.

Collection	collect(Collection collection) {closure}
(void)	each {closure}

continued on next page

(void)	eachWithIndex {closure}
Boolean	every {closure}
Object	find {closure}
List	findAll {closure}
Integer	findIndexOf {closure}
List	grep(Object filter)

What's so useful about the methods in table 9.2 is that you can use them on *any* object you fancy. The GDK makes these methods available on `Object` and yields the respective items. As we described in section 6.3.2, this iteration strategy is also used in Groovy's `for` loop.

Getting the items is done with a best-effort strategy for the candidate types listed in table 9.3, where the first matching possibility is chosen.

Table 9.3 Priority of `Object`'s iteration strategy

No.	Candidate	Use with
1	`java.util.Iterator`	Itself
2	`org.w3c.dom.NodeList`	Iterator over `Nodes`
3	`java.util.Enumeration`	Convert to iterator
4	`java.util.regex.Matcher`	Iterator over matches

5	Responds to `iterator` method	Call it
6	Collectable	`Collection.iterator()`
7	`java.util.Map`	Iterator over `Map.Entry` objects
8	Array	Iterator over array items
9	`MethodClosure`	Iterator over calls
10	`java.lang.String`	Iterator over characters
11	`java.io.File`	Iterator over lines
12	`null`	Empty iterator
13	Otherwise	Iterator that only contains the candidate

This allows for flexible usages of Groovy's iteration-aware methods. There is no more need to care whether you work with an iterator, an enumeration, a collection, or whatever, for example within a GPath expression.

The possible candidates in table 9.3 are fairly straightforward, but some background information certainly helps:

Candidate 2: A `NodeList` is used with a *Document Object Model (DOM)*. Such a DOM can be constructed from, for example, XML or HTML documents. We will revisit this topic in chapter 12.

Candidate 5: A candidate object may provide its `Iterator` with the `iterator` method. Instead of a single static interface, the availability of the `iterator` method is used in the sense of duck-typing. An example of such an object is `groovy.util.Node`.

Candidate 6: A candidate object is *collectable* if it can be coerced into an object of type `java.util.Collection`.

Candidate 9: This is an unconventional way of providing an `Iterator`, but it's interesting because it puts our Groovy knowledge to the test.

Suppose you have a method that takes a closure as a parameter and calls the closure back with a single argument, multiple times, using a different argument each time. This could be seen as successively passing arguments to a closure. *Successively passing arguments* is exactly what an `Iterator` does. To make this method work as an iterator, refer to it as a `MethodClosure`, as described in section 5.3.3.

As an example, imagine calculating *sin(x)* for sample domain values of *x* between zero and 2p. A `domain` method can feed an arbitrary `yield` closure with these *x* samples:

```
samples = 4

def domain(yield) {
    step = Math.PI * 2 / samples
    (0..samples).each { yield it*step }
}
```

Printing the *x* values would be as simple as invoking

```
domain { println it}
```

As the `domain` method successively passes objects to the given closure, it can be used with the object-iteration methods—for example, with `collect` to get a list of sine values for all samples from the domain. Use a reference to the `domain` method: `this.&domain`, which makes it a `MethodClosure`.

```
this.&domain.collect { Math.sin(it) }
```

Using a `MethodClosure` as an `Iterator` doesn't seem to provide much advantage other than reusing a method that possibly already exists. Our `domain` method could have returned a list of *x* values. Things would have been easier to understand that way. There also isn't a performance or memory consumption gain, because this list is constructed behind the scenes anyway when converting the closure.

However, it may be handy when the method does more than our simple example. It could produce side-effects—for example, for statistical purposes. It could get data from a live datafeed or some expensive resource with an elaborate caching strategy. Because references to `MethodClosure`s can be held in variables, you could change this strategy at runtime (Strategy pattern).[6]

Those were the GDK methods for `Object`. There are more methods in the GDK for arrays of objects. They make arrays usable as lists such that Groovy programmers can use them interchangeably. These methods were described in section 4.2.

Not surprisingly, GDK's object methods are about all-purpose functionality such as revealing information about an object's state and dynamically accessing properties and invoking methods. Iterating over objects can be done regardless of each object's behavior.

[6]Gamma et al.

The next sections will cover GDK methods for more specialized but frequently used JDK classes used for I/O, such as `File`.