

Dynamic programming with Groovy



This chapter covers

- How Groovy supports dynamic programming
- An explanation of the Meta Object Protocol (MOP)
- How to utilize the MOP for your own purposes

Until real software engineering is developed, the next best practice is to develop with a dynamic system that has extreme late binding in all aspects.

—Alan Kay

We’re going to start our journey with a few general considerations about dynamic programming, how it differs from conventional object-oriented approaches, and why you want to have it in your toolbox. We’ll show how the MOP serves as the central hub that provides you with dynamic programming capabilities. Groovy comes with dynamic features out-of-the-box but you can also add your own. There are various ways of achieving this and we’ll start with the simpler ones and slowly move on to the more advanced use cases. As you’ll see, there’s no reason to be scared about words like “dynamic” or “meta.” If by the end of this chapter you say, “Well, it isn’t so magical after all,” then we’ve achieved our goal.

If you seek perfection in completeness, designing and implementing an object-oriented system becomes hard. It may well be impossible.

Imagine you're responsible for `java.lang.Integer`. You're of course aware that this class will be used for counting, indexing, calculations, and so on, but you cannot possibly anticipate all use cases.

Before not too long, somebody will come along and would like to use it with a `times` method like in `3.times { println it }`, which you haven't foreseen, or calculate dates as in `2.weeks.from.today`, but you haven't provided a `getWeeks()` method on `Integer` that would be needed to make this possible.

On another occasion, another user of your class may prefer having an exception being thrown on `Integer.MAX_VALUE + 1` rather than returning a negative number.

A third user would like to optimize an algorithm and count the number of modulo operations on any integer that happen when the algorithm is executed. You're very unlikely to have anticipated such a requirement.

The good news is dynamic programming allows adding such *features* later—without even touching the original! And the original can even be a Java class as long as it's called from Groovy.

Changes to such a ubiquitously used class as `Integer` are better only applied to the scope where you need them or you risk interference with seemingly unrelated parts of your codebase. Therefore, dynamic programming allows using such a feature only temporarily: adding and removing it at runtime; limiting its use to a given piece of code, a class, or only single instances; or even confining it to the current thread of execution.

Dynamic programming has a wide range of applicability, including

- Designing DSLs (see chapter 18)
- Implementing builders (see chapter 11)
- Advanced logging, tracing, debugging, and profiling
- Automated testing even where testing seems impossible (see chapter 17)
- Putting lipstick on existing APIs—for example, by eliminating the “incomplete library class” smell¹—to make them more complete, coherent, and accessible
- Organizing the codebase so that features are kept in one place even if their behavior involves the collaboration of multiple classes; for example, you need abstractions for date, time, and duration working together to provide the date-calculation feature.

The last point is particularly interesting. You can observe it in Grails where the *persistence feature* is dynamically available in all domain classes. On a domain class like `Person` you can call `Person.findAllByFirstName('Dierk')` to find all people in the database that share the first name Dierk, even though this method doesn't exist!

¹ See chapter 3 of *Refactoring*, by Martin Fowler and Kent Beck (Addison-Wesley, 1999).

Note that such an approach has one quality that static code generation never achieves: because the code isn't materialized anywhere, you cannot introduce errors in it! Also, your code is kept as clean as possible and you never have to read through code that was generated!

In this chapter, we'll go through the various means of dynamic programming in Groovy and provide examples of use cases. Now, let's start with looking at what mechanics make programming *dynamic*.

8.1 *What is dynamic programming?*

In classic object-oriented systems, every class has a well-known set of states, captured in the fields of that class, and well-known behavior, defined by its methods. Neither the set of states nor the behavior ever changes after compilation, and it's identical for all instances of a class.

Dynamic programming breaks this limitation by allowing the introduction of a new state, or even more importantly, allowing the addition of a new behavior or modification of an existing one.

What is “meta”?

Meta means applying a concept onto itself—for example, metainformation is information about information. Likewise, because programming is “writing code,” metaprogramming means writing code that writes code. This includes source-code generation (for example, producing a long string that's then evaluated as a script), bytecode generation as explained in the next chapter, and pretending or synthesizing methods. The latter is part of dynamic programming and we'll encounter it later in this chapter.

The use of “meta” as a qualifier in the Groovy runtime system is in many places debatable. Anyway, it isn't only there for historical reasons; it also suggests that we're working on an elevated abstraction level whenever this word is used.

How can you possibly add a new state and behavior at runtime, when you're working on the JVM and the Java object model provides no such means? As the saying goes, “Every problem in computer science can be solved with a layer of indirection (besides the problem of too many layers of indirection).”² Enter the Meta Object Protocol.

8.2 *Meta Object Protocol*

The approach is actually rather straightforward. Whenever Groovy calls a method, it doesn't call it directly but asks an intermediate layer to do so on its behalf. The intermediate layer provides hooks that allow you to influence its inner workings.

² For a brief biography of David Wheeler, said to have invented the subroutine, see http://en.wikipedia.org/wiki/David_Wheeler_%28British_computer_scientist%29.

A protocol is a collection of rules and formats. The MOP is the collection of rules of how a request for a method call is handled by the Groovy runtime system and how to control the intermediate layer. The format of the protocol is defined by the respective APIs, which we'll walk through in the course of this chapter.

An important part of understanding the mechanics is knowing what it means when we say that Groovy calls a method. When writing Groovy source code, the Groovy compiler generates bytecode that calls into the MOP.

As an illustration, assume that your Groovy source code contains the statement

```
println 'Hello' // Groovy
```

Then the resulting bytecode that Groovy produces is roughly equivalent to the following Java code:

```
InvokerHelper.invokeMethod(this, "println", {"Hello"}); // Java
```

When executed, the `InvokerHelper` as part of the MOP looks for the method named "println" with a `String` argument, finds that the Groovy runtime has registered such a method for `java.lang.Object`, and calls that implementation. This is a slightly simplified description of what actually happens, but one that explains the principle and one that we can start with.

NOTE Every innocent method call that you write in Groovy is really a call into the MOP, regardless of whether the call target has been compiled with Groovy or Java. This applies equally to static and instance method calls, constructor calls, and property access, even if the target is the same object as the caller.

Figure 8.1 shows how the MOP works like a filter for all method calls that originate from code that was compiled by Groovy. The MOP is like a pair of rainbow-colored glasses that makes all objects appear rich and powerful.

Figure 8.1 shows what happens by default. Of course, you can also call into the MOP from Java but this requires calling `InvokerHelper.invokeMethod()` explicitly. By default, Java classes only see what's in the bytecode of a class and not what the MOP adds to it, even if the target class was compiled by Groovy.

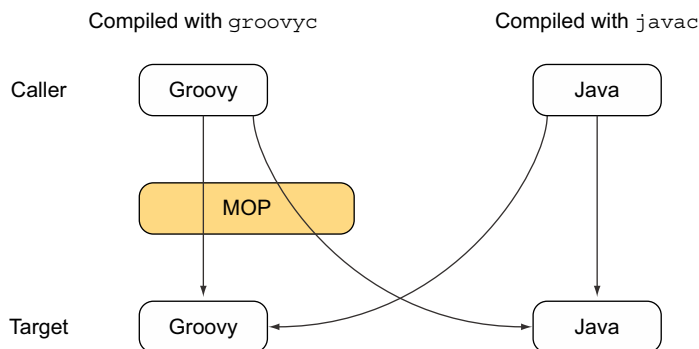


Figure 8.1 Every method call from a Groovy class or object into either Groovy or Java automatically goes through the MOP. Method calls from Java to both Groovy and Java targets don't use the MOP per default.

There's no spoon

Relating to the *Matrix* motion picture,³ there's no such thing as a Groovy class. You may have noticed that we avoid the wording of Java class versus Groovy class. That's because classes are classes, regardless of who compiled them. They have the same format and constraints. Of course, they differ in content, but so do all classes anyway.

The MOP needs a lot of information to find the right call target for each method call that it serves. This information is stored in so-called metaclasses. These metaclasses aren't fixed. One part of dynamic programming is changing the content of metaclasses and replacing one metaclass with another. We'll explore this in section 8.4.

But even with the default metaclass being in place, which does nothing fancy besides providing the GDK methods and doing some very advanced performance optimizations, the MOP knows about some special methods that allow the first degree of dynamic behavior. They're called *hook methods*.

8.3 Customizing the MOP with hook methods

The core of the MOP responsibilities is finding and selecting the right target method and handling the case when the requested method cannot be found. The first hook method that we'll look at allows customizing the "missing method" case. A second one covers the case that a property access fails to find a property of the requested name. Then we'll explore the effects of combining hook methods with closure properties to allow instance-specific hooks that can even change at runtime. We finish up with custom logic for methods that objects need to provide if they implement the Groovy-Object interface.

8.3.1 Customizing methodMissing

Whenever a method cannot be found in the target object, the MOP looks for the hook method

```
Object methodMissing(String name, Object arguments)
```

and invokes it with the requested method name and arguments.

The next listing uses this hook in a Pretender class to merely return a string that shows what had been requested. The Pretender only *pretends*⁴ to have the method `hello(String)`, but in the bytecode of the class there's no such method.

³ The main character sees a child playing with a dynamic spoon object and realizes that it isn't a truly physical object. Dynamic programming is kind of like that.

⁴ Some call this a synthesized method, but we feel this suggests that it somehow materializes, which it doesn't.

Listing 8.1 Bouncing when a missing method is called

```
class Pretender {
  def methodMissing(String name, Object args) {
    "called $name with $args"
  }
}

def bounce = new Pretender()
assert bounce.hello('world') == 'called hello with [world]'
```

The target is absolutely free in what it does inside `methodMissing`. It may provide a more sophisticated error handling than merely throwing a `MissingMethodException` (which is the default), delegate all calls to a collaborating object, or inspect the method name and arguments to derive what needs to be done.

The use case in listing 8.1 goes into the direction that Grails provides with its dynamic finder methods in the Groovy object-relational mapping (GORM). The following listing outlines the approach. A method name like `findByXXX` is analyzed to select the search criterion.

Listing 8.2 Using `methodMissing` to simulate a miniature GORM

```
class MiniGorm {
  def db = []
  def methodMissing(String name, Object args) {
    db.find { it[name.toLowerCase() - 'findBy'] == args[0] }
  }
}

def people = new MiniGorm()
def dierk = [first: 'Dierk', last: 'Koenig']
def paul = [first: 'Paul', last: 'King']
people.db << dierk << paul

assert people.findByFirst('Dierk') == dierk
assert people.findByLast('King') == paul
```

Extracts criterion from method name

Sets up test data

Calls with pretended methods

Of course, a full implementation of GORM dynamic finder methods is more complex, but the principle and the mechanics are the same.

FOR THE GEEKS You can share the implementation of `methodMissing` by various means. One is to put it in a base superclass. In section 8.4 we'll see how methods can be injected into a class without even pretending to have access to the code of that class! You can even pretend to have hook methods.

The `methodMissing` hook is quite simple to understand and use. Yet, it's very versatile and covers the vast majority of use cases for dynamic programming with DSLs. It's a very good entry point into dynamic programming and a good default choice when deciding upon which means of dynamic programming to apply. It comes with a counterpart that does to property access what `methodMissing` does to method calls.

8.3.2 Customizing `propertyMissing`

What `methodMissing` is for method calls, `propertyMissing` is for property access. You implement

```
Object propertyMissing(String name)
```

to catch all access to nonexisting properties. All the rest is exactly analogous to `methodMissing` such that the following listing should be rather self-explanatory. We try to access the `hello` property, which isn't in the bytecode.

Listing 8.3 Bouncing when a missing property is accessed

```
class PropPretender {
    def propertyMissing(String name) {
        "accessed $name"
    }
}

def bounce = new PropPretender()
assert bounce.hello == 'accessed hello'
```

This hook is a specialization of `methodMissing`: if you pretend the respective getter method, you achieve the same effect. Anyway, having this more specialized hook is sometimes convenient. In listing 8.4 we use this hook as a method of the `Script` class to implement an easy way to calculate with binary numbers. This actually feels like a DSL.

The idea is quite simple. We'd like to use symbols like `II0I` to specify a positive integer of value 9 in its binary form. Now, simply using this symbol would throw us a `MissingPropertyException`. By providing a `propertyMissing` hook we can do the translation from a string into an integer.

Listing 8.4 Using `propertyMissing` to calculate with binary numbers in DSL style

```
def propertyMissing(String name) {
    int result = 0
    name.each {
        result <= 1
        if (it == 'I') result++
    }
    return result
}

assert II0I +
       IOI ==
       IO0IO
```

In case you have difficulties with the string-to-integer translation logic here, don't worry, it's an implementation detail. The main point to take away is how to use the hook method.

Where there's specialization, there may also be generalization and, actually, there is. But before we come to that in section 8.3.4, we'll enter a new dimension of dynamicity.

8.3.3 Using closures for dynamic hooks

By now, you may have the impression that MOP hook methods are very conventional. And in a way they are. They're just ordinary methods.

But if you think that this means that their behavior is guaranteed to be identical for all instances of your class, then this isn't quite so in Groovy. In fact, if you wish so, you can even change the hook logic during the lifetime of an object!

Hook methods aren't static. They're instance methods. Being that, they can work with the object's state. This state can include parameters for the hook logic. If these properties are of type `Closure`, then we have another example of parameterization with logic (see chapter 5).

The next listing maintains a `whatToDo` property of type `Closure` that's called from inside a hook method. This allows changing the hook logic at runtime, and (not shown) having multiple instances of `DynamicPretender` using different closures.

Listing 8.5 Using the closure property to change hook logic at runtime

```
class DynamicPretender {
    Closure whatToDo = { name -> "accessed $name" }
    def propertyMissing(String name) {
        whatToDo(name)
    }
}

def one = new DynamicPretender()
assert one.hello == 'accessed hello'
one.whatToDo = { name -> name.size() }
assert one.hello == 5
```

Closure property with default logic

Delegates to the closure

Changes hook behavior at runtime

In classic Java programming, the behavior of a class never changes and the behavior is the same for all objects of the class. At best, you can use a Strategy pattern⁵ to switch between objects that behave differently. The previous pattern of using a closure property to customize behavior of an object has a dynamic touch in itself, even though it's totally independent of the MOP. But in combination with the MOP, it adds a new dimension to the solution space.

NOTE All features of dynamic programming that are explained in this chapter can be combined with closure properties to open another dimension of versatility.

The hook methods that we've talked about so far apply regardless whether the call target is compiled by Groovy or Java. The next section will be about more specific handling that the MOP applies to Groovy targets.

⁵ *Design Patterns: Elements of Reusable Object-Oriented Software*, by Gamma et al. (Addison-Wesley, 1994).

8.3.4 Customizing GroovyObject methods

All classes that are compiled by Groovy implement the `GroovyObject` interface, which looks like this:

```
public interface GroovyObject {
    Object    invokeMethod(String methodName, Object args);
    Object    getProperty(String propertyName);
    void      setProperty(String propertyName, Object newValue);
    MetaClass getMetaClass();
    void      setMetaClass(MetaClass metaClass);
}
```

Again, you're free to implement any such methods in your Groovy class to your liking. If you don't, then the Groovy compiler will insert a default implementation for you. This default implementation is the same as if you'd inherit from `GroovyObjectSupport`, which basically relays all calls to the metaclass. It roughly looks like this (excerpt):

```
public abstract class GroovyObjectSupport implements GroovyObject {

    public Object invokeMethod(String name, Object args) {
        return getMetaClass().invokeMethod(this, name, args);
    }
    public Object getProperty(String property) {
        return getMetaClass().getProperty(this, property);
    }
    public void setProperty(String property, Object newValue) {
        getMetaClass().setProperty(this, property, newValue);
    }
    // more here...
}
```

We defer the explanation of the metaclass handling to the next section. For the moment, it's just a device that we can use for calling into the MOP.

NOTE You can fool the MOP into thinking that a class that was actually compiled by Java was compiled by Groovy. You only need to implement the `GroovyObject` interface or, more conveniently, extend `GroovyObjectSupport`.

As soon as a class implements `GroovyObject`, the following rules apply:

- Every access to a property calls the `getProperty()` method.⁶
- Every modification of a property calls the `setProperty()` method.
- Every call to an unknown method calls `invokeMethod()`. If the method is known, `invokeMethod()` is only called if the class implements `GroovyObject` and the marker interface `GroovyInterceptable`.

Let's use this newly acquired knowledge to play with the Groovy language rules. In Groovy, parentheses for method calls are optional for top-level statements, but only if

⁶ There is a special handling for maps in the default metaclass that makes sure that even though `Map` isn't a `GroovyObject`, every property access on a map is relayed to the respective `MapEntry`.

there's at least one argument. This is needed to distinguish method calls from property access. We cannot call `toString()` without the parentheses because `toString` would refer to the property of the name `toString`. The next listing allows us to go around this limitation. We implement `getProperty()` such that if the property exists, we return its value; if not, we assume that the parameterless method will be executed. Such a feature can be interesting when designing DSLs.

Listing 8.6 Using `getProperty` to call parameterless methods without parentheses

```
class NoParens {
    def getProperty(String propertyName) {
        if (metaClass.hasProperty(this, propertyName)) {
            return metaClass.getProperty(this, propertyName)
        }
        invokeMethod propertyName, null
    }
}

class PropUser extends NoParens {
    boolean existingProperty = true
}

def user = new PropUser()
assert user.existingProperty
assert user.toString() == user.toString()
```

1 Properties have priority

2 Dynamic invocation

3 Subclass for feature sharing

4 Look, Ma, no parentheses!

This example uses the metaclass and so leads us slowly into the topic of the next section where we'll explore this concept in more detail.

When we ① check whether a known property is requested, we ask the `metaClass` (that is, we call the `getMetaClass()` method) if it has such a property. In case it has, we ask the `metaClass` for its value. Note that we cannot simply use `this."$propertyName"` because this would call `getProperty()` again, leading to endless recursion.

To eventually execute the method ②, we call the default implementation of the `invokeMethod()` hook, which relays the call to the metaclass.

We see in ③ that subclasses can share this `NoParens` feature. Subclassing is generally not a good way of sharing features but it works. We'll discuss this further and provide better alternatives at a later time.

We assert ④ that omitting parentheses really works with selecting the ubiquitously available `toString()` method as our test candidate. Existing properties remain untouched.

Implementing `get/setProperty` can often improve the elegance of an API. Just consider Groovy maps. They relay property access like `map.a` to map content access like `map['a']` and you can do the equivalent with your own objects.

NOTE Once you've implemented `getProperty()`, every property will be found and thus `propertyMissing()` will no longer be called.

So far, you've seen various means of dynamic programming that require access to the source code of the target class and the possibility to apply modifications to it. We call

this approach *intrusive*. You may be glad to hear that there's also a *nonintrusive* approach, which is the topic of our next section.

8.4 *Modifying behavior through the metaclass*

By now you should feel at ease with the situation that all method calls that originate from Groovy code are routed through the MOP. If the last sentence still sounds odd to you, consider rereading section 8.2 and doing more experiments around the provided examples until you've gained enough confidence to proceed.

With `methodMissing` and `propertyMissing` you've seen examples of hook methods that the MOP invokes when it cannot find the requested method or property. In this section, we'll explore how Groovy tries to locate those and how you can use the lookup mechanism for the purposes of customizing the object's behavior.

8.4.1 *MetaClass knows it all*

For every class *A* in the class loader, Groovy maintains a metaclass—an object of type `MetaClass`. This metaclass maintains the collection of all methods and properties of *A*, starting with the bytecode information of *A* and adding additional methods that Groovy knows about per default (`DefaultGroovyMethods`).

Generally, all instances of class *A* share the same metaclass. But Groovy also supports having per-instance metaclasses—that is, different instances of *A* may refer to different metaclasses. We'll revisit this situation later.

You can easily ask any metaclass for its metainformation (recall seeing the information in figure 1.6, which displayed the Groovy Object Browser).

Listing 8.7 inspects the capabilities of `MetaClass` by asking `String` for its metaclass and calling various methods on it. We inspect the availability of methods with `respondsTo`, list all properties, list all methods from the bytecode, list all `metaMethods` that Groovy added dynamically, and call `invokeMethod`, `invokeStaticMethod`, and `invokeConstructor` to show dynamic invocation.

Listing 8.7 `MetaClass` is key to Groovy reflection and dynamic method invocation

```
MetaClass mc = String.metaClass
final Object[] NO_ARGS = []
assert 1 == mc.respondsTo("toString", NO_ARGS).size()
assert 3 == mc.properties.size()
assert 74 == mc.methods.size()
assert 176 == mc.metaMethods.size()
assert "" == mc.invokeMethod("", "toString", NO_ARGS)
assert null == mc.invokeStaticMethod(String, "println", NO_ARGS)
assert "" == mc.invokeConstructor(NO_ARGS)
```

Numbers may vary depending on Java version.

There are more methods and more variants in `MetaClass`, but those in the previous listing give a good overview of what it does in general: providing means of *reflection* and *dynamic invocation*.

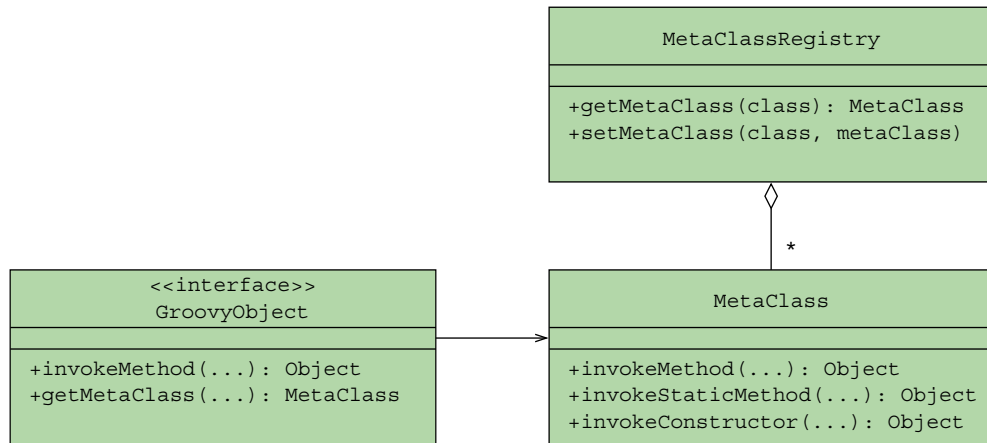


Figure 8.2 A UML class diagram of the `GroovyObject` interface that refers to an instance of class `MetaClass`, where `MetaClass` objects are also aggregated by the `MetaClassRegistry` to allow class-based retrieval of metaclasses in addition to `GroovyObject`'s optional object-based retrieval.

Calling a method means calling the metaclass

You can assume that Groovy never calls methods directly in the bytecode but always through the object's metaclass. At least, this is how it looks to you as a programmer.

Behind the scenes there are optimizations going on that technically circumvent the metaclass, but only when it's safe to do so.

Even the MOP hook methods that you've seen in earlier sections make no exception. If you provide your own implementation of let's say `invokeMethod`, then this method is added to your object's metaclass at class loading time and later invoked from there.⁷

All this should look to you as a pretty simple rule and you may ask what's so special about it. The trick is that a metaclass can change at runtime and that an object may also change its metaclass. Let's first investigate how Groovy finds metaclasses.

8.4.2 How to find the metaclass and invoke methods

You've seen that all `GroovyObjects` have a `metaClass` property (`setMetaClass` and `getMetaClass` methods). That makes it easy to find the metaclass for them. You simply ask the object with `obj.metaClass`.

If you don't provide a custom implementation of the `metaClass` property accessor methods, the default implementation looks up the metaclass in the so-called `MetaClassRegistry`. The registry maintains a map of classes and their metaclasses. Figure 8.2 displays the connection among `GroovyObject`, `MetaClass`, and `MetaClassRegistry`.

⁷ If you add hook methods like `invokeMethod` to a superclass or interface, you need to previously call `ExpandableMetaClass.enableGlobally()` if you want that hook method to apply further down the inheritance hierarchy.

Objects that don't inherit from `GroovyObject` aren't asked for the `metaClass` property. Their metaclass is retrieved from the `MetaClassRegistry`.

FOR THE GEEKS The default metaclass can be changed from the outside without touching any application code. Let's assume you have a class `Custom` in package `custom`. Then you can change its default metaclass by putting a metaclass with the name `groovy.runtime.metaclass.custom.CustomMetaClass` on the classpath. This device has been proven useful when inspecting large Groovy codebases in production.

Putting all this together is a bit of a challenge. The following snippet is a sketch in pseudocode to keep the level of detail manageable while still revealing the core of the logic. Important methods from the metaclass are shown in bold italics, hook methods as underlined. Note that `invokeMethod` appears twice: with two parameters as a hook method and with three parameters in `MetaClass`.

At the very beginning, you decide whether you have a `GroovyObject` and, if not, look for the metaclass in the registry and use it to invoke the requested method:

```
// MOP pseudo code
def mopInvoke(Object obj, String method, Object args) {
    if (obj instanceof GroovyObject) {
        return groovyObjectInvoke(obj, method, args)
    }
    registry.getMetaClass(obj.class).invokeMethod(obj, method, args)
}
```

If you have a `GroovyObject`, you use the `metaClass` property to find the metaclass but you also have to care for the special handling around `GroovyInterceptable` and unknown methods (see section 8.2):

```
def groovyObjectInvoke(Object obj, String method, Object args){
    if (obj instanceof GroovyInterceptable) {
        return obj.metaClass.invokeMethod(method, args)
    }
    if (! obj.metaClass.respondsTo(method, args)) {
        return obj.metaClass.invokeMethod(method, args)
    }
    obj.metaClass.invokeMethod(obj, method, args)
}
```

You may ask why `methodMissing` doesn't appear in the preceding code. This case is handled in the default metaclass:

```
// Default meta class pseudo code
def invokeMethod(Object obj, String method, Object args) {
    if (obj.metaClass.respondsTo(method, args)) {
        return methodCall(obj, method, args)
    }
}
```

```

    if (methodMissingAvailable(obj)) {
        return obj.metaClass.methodMissing(method, args)
    }
    throw new MissingMethodException()
}

```

Don't forget that all the previous is pseudocode, that actual implementation differs quite a bit, mostly for performance reasons. Also, the code is supposed to have Java semantics; that is, all method calls and property access are direct and don't go through the MOP itself. Otherwise, you'd run into endless recursion.

The mechanics of the MOP may appear complex but for usual cases you can assume that all method calls go through the metaclass and the default metaclass is in place. This raises the question what other metaclasses are available and why you'd want to use them.

8.4.3 Setting other metaclasses

Groovy comes with a number of metaclasses:

- The default metaclass `MetaClassImpl`, which is used in the vast majority of cases
- The `ExpandoMetaClass`, which can expand the state and behavior
- A `ProxyMetaClass`, which can decorate a metaclass with interception capabilities
- Additional metaclasses that are used internally and for testing purposes

Let's look at `ProxyMetaClass` as an example of how to use a customized metaclass. A `ProxyMetaClass` wraps an existing metaclass that it relays all method calls to. When doing so, it provides the ability to execute customized logic before and after each method call. That customized logic is captured in a so-called `Interceptor`. With Groovy comes a `TracingInterceptor` that simply logs all method access to a writer, effectively providing a trace of all method calls. The following listing configures such a `ProxyMetaClass` with a `TracingInterceptor` and assigns this metaclass to an arbitrary Groovy object that should be subject to tracing.

Listing 8.8 Assigning a `ProxyMetaClass` to a `GroovyObject` for tracing method calls

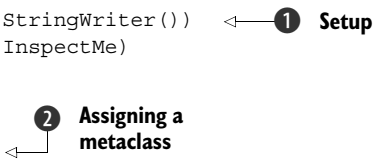
```

class InspectMe {
    int outer(){
        return inner()
    }
    private int inner(){
        return 1
    }
}

def tracer = new TracingInterceptor(writer: new StringWriter())
def proxyMetaClass = ProxyMetaClass.getInstance(InspectMe)
proxyMetaClass.interceptor = tracer

InspectMe inspectMe = new InspectMe()
inspectMe.metaClass = proxyMetaClass

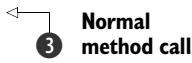
```



```

assert 1 == inspectMe.outer()
assert "\n" + tracer.writer.toString() == ""
before InspectMe.outer()
    before InspectMe.inner()
    after  InspectMe.inner()
after  InspectMe.outer()
""

```



Our self-testing code requires a small change to the default. In ❶ we set up the Tracing-Interceptor to not print to `System.out` but to use a `StringWriter` that we can later inspect for its content.

In ❷ we assign our metaclass to the object under inspection. We do have a per-instance metaclass this way.

When we call any method on our object under inspection as in ❸, then all method calls and returns are traced, even in private methods. Note that this doesn't require any change in `InspectMe` nor is that class in any way aware of the tracing. It's all controlled from the outside. This is what we call *nonintrusive*.

Interceptors are more than aspects

Interceptors may remind one or the other reader of aspect-oriented programming (AOP) and the `TracingInterceptor` suggests this connotation. But interceptors can do much more: they can redirect to a different method, change the arguments, suppress the method call, and even change the return value!

Oftentimes, you may want to use the proxy metaclass only temporarily and set the metaclass back to the original afterwards. In such a case you can put the proxy-using code inside a closure and give it to the `use` method like this:

```

proxyMetaClass.use(inspectMe){
    inspectMe.outer() // proxy in use
}
// proxy is no longer in use

```

Manually setting the metaclass of a Groovy object works as expected and working through the example has confirmed your understanding of the MOP. But Groovy wouldn't be Groovy if it would leave you behind with only the low-level devices.

In the next sections you'll see ways of working with the MOP on a higher level of abstraction to make it more accessible, more flexible, and more convenient to work with for specialized use cases.

8.4.4 Expanding the metaclass

Groovy has, since its early days, a class called `Expando`. It's a tiny class with few methods but one interesting characteristic: it can expand its state and behavior. The following listing uses an `Expando` as a boxer who can take some hits but will eventually fight back.

Listing 8.9 An Expando can extend the state and behavior at runtime

```
def boxer = new Expando()

boxer.takeThis = 'ouch!'
boxer.fightBack = { times -> takeThis * times }

assert boxer.fightBack(3) == 'ouch!ouch!ouch!'
```

A new state is assigned to not-yet-existing properties, analogous to what you've seen for maps.

A new behavior is assigned to not-yet-existing properties as closures. After the assignment, it can be called as if it was a method.

The reason for explaining the Expando class here is that there's an ExpandoMetaClass in Groovy that, as you may have guessed, is a metaclass that works like an Expando. You can register a new state (properties) and new behavior (methods) in the metaclass by using property assignments.

Listing 8.10 introduces the concept with an example that adds a new method called `low()` to `java.lang.String`. It does the same as `toLowerCase()` but is shorter and the spelling is easier to remember. We don't need to set the `ExpandoMetaClass` explicitly. Groovy automatically replaces the default metaclass with an `ExpandoMetaClass` when we apply any modification to it.

Listing 8.10 Adding `low()` to `java.lang.String` via `ExpandoMetaClass`

```
assert String.metaClass =~ /MetaClassImpl/
String.metaClass.low = {-> delegate.toLowerCase() }
assert String.metaClass =~ /ExpandoMetaClass/

assert "DiErK".low() == "dierk"
```

Note that our closure uses the delegate reference to refer to the actual `String` instance that the closure is called upon. The closure must also have the right number of parameters. The usual rules for closure parameters apply—type markers are optional, and you can use default values, varargs, and so forth. Because our method will not have any parameters we use an empty parameter list `{-> ...}`.

The next listing adds a new property (`myProp`) and a new method (`test`) to the metaclass of `MyGroovy1`—a class that's written in Groovy. Note that the dynamic `test` method refers to the dynamic property `myProp`. These dynamic features are only available for objects of type `MyGroovy1` that have been constructed *after* the metaclass modification.

Listing 8.11 Modifying the metaclass of a class (Groovy and Java)

```
class MyGroovy1 { }

def before = new MyGroovy1()

MyGroovy1.metaClass.myProp = "MyGroovy prop"
MyGroovy1.metaClass.test = {-> myProp }
```



```

try {
    before.test()
    assert false, "should throw MME"
} catch(mme) { }

```

← **Not available**

```

assert new MyGroovy1().test() == "MyGroovy prop"

```

Here we've changed the metaclass of a class and thus for all instances of that class. In the following listing we do the very same but only on a single instance. Only the `myGroovy` instance gets the new dynamic features because we only modify a per-instance metaclass.

Listing 8.12 Modifying the metaclass of a Groovy instance

```

class MyGroovy2 { }

def myGroovy = new MyGroovy2()

myGroovy.metaClass.myProp = "MyGroovy prop"
myGroovy.metaClass.test = { -> myProp }

try {
    new MyGroovy2().test()
    assert false, "should throw MME"
} catch(mme) { }

```

← **Not available**

```

assert myGroovy.test() == "MyGroovy prop"

```

Per-instance metaclasses are very valuable because they allow fine-grained control over where and how dynamic features are added.

Imagine a large development team where accidentally two developers modify the same metaclass with the same method names for different reasons. The last modification wins and may compromise the logic of the developer who did the first change.⁸

With per-instance metaclasses such clashes are easier to avoid. The next listing uses per-instance metaclasses even for such a ubiquitous Java object as a `String` while avoiding clashes.

Listing 8.13 Modifying the metaclass of a Java instance

```

def myJava = new String()

myJava.metaClass.myProp = "MyJava prop"
myJava.metaClass.test = { -> myProp }

try {
    new String().test()
    assert false, "should throw MME"
} catch(mme) { }

```

← **Not available**

```

assert myJava.test() == "MyJava prop"

```

⁸ This situation is often called “monkey patching,” referring to programmers who use programming constructs that they’ve seen elsewhere without fully understanding what they do: “monkey see, monkey do.”

So far, we've asked classes and objects for their metaclass every single time when we did a modification. Listing 8.14 introduces a new so-called *builder* style for doing multiple changes at once. We use it to encode and decode strings by moving every character up and down the alphabet with the respective methods, a metaclass property to capture how many characters to shift up or down, and property accessor methods to work more conveniently with the code and the original.

If you've ever seen Stanley Kubrick's motion picture *A Space Odyssey*, you may remember the super-intelligent computer HAL. It turns out that this is an encoded version of IBM. Well, things could have been worse for that company if the writer Arthur C. Clarke had chosen a different shift distance for the encoding...

Listing 8.14 Decoding *A Space Odyssey* with a metaclass builder

```
def move(string, distance) {
  string.collect { (it as char) + distance as char }.join()
}

String.metaClass {
  shift = -1
  encode {-> move delegate, shift }
  decode {-> move delegate, -shift }
  getCode {-> encode() }
  getOrig {-> decode() }
}

assert "IBM".encode() == "HAL"
assert "HAL".orig      == "IBM"

def ibm = "IBM"
ibm.shift = 7
assert ibm.code == "PIT"
```

Note that we can change the shift distance on a per-instance basis by setting the respective property.

NOTE Modifying the metaclass of the `String` class will affect all future `String` instances.

In all the preceding examples, we've added new instance methods to all instances of a class or to only a specific instance of a class. The following listing adds a static method to `java.lang.Integer` by using the `static` keyword. We can now ask the `Integer` class (as opposed to an `Integer` object) for the answer to “life, the universe, and everything.”

Listing 8.15 Adding a static method to a class

```
Integer.metaClass.static.answer = {-> 42}

assert Integer.answer() == 42
```

When talking about objects, we also have to consider inheritance. Listing 8.16 adds a new method `toTable()` dynamically to a superclass and asserts that it's transparently available in its subclass.

We can even modify the metaclass of interfaces and all classes that implement this interface share the new behavior.

Listing 8.16 Metaclass changes for superclasses and interfaces

```
class MySuperGroovy { }
class MySubGroovy extends MySuperGroovy { }

MySuperGroovy.metaClass.added = { -> true }

assert new MySubGroovy().added()

Map.metaClass.toTable = { ->
    delegate.collect{ [it.key, it.value] }
}

assert [a:1, b:2].toTable() == [
    ['a', 1],
    ['b', 2]
]
```

Note that we call `toTable()` on a literally declared map, which is of type `LinkedHashMap`. Even though we've added the new method to the metaclass of the `java.util.Map` interface, it's available for all instances of its subtypes.

We mop⁹ up the metaclass topic with an example that should illustrate that we can add any kind of method dynamically, even operator methods and MOP hook methods.

Listing 8.17 adds the `>>>` operator to strings with the `rightShiftUnsigned` operator method to split the string by words and push them to the right. It then replaces names with nicknames by calling a method of the to-be-replaced name with the replacement as the argument. To make this possible for every conceivable name, it adds the `methodMissing` hook to `String`.

Listing 8.17 Metaclass injection of operator and MOP hook methods

```
String.metaClass {
    rightShiftUnsigned = { prefix ->
        delegate.replaceAll(/~/\w+/) { prefix + it }
    }
    methodMissing = { String name, args->
        delegate.replaceAll name, args[0]
    }
}

def people = "Dierk,Guillaume,Paul,Hamlet,Jon"
people >>>= "\n      "
people      = people.Dierk('Mittie').Guillaume('Mr.G')
```

⁹ Pun intended but we'll try not to wring out the analogy too far.

```
assert people == '''
    Mittie,
    Mr.G,
    Paul,
    Hamlet,
    Jon'''
```

Some takeaways and rules of thumb for metaclasses:

- All method calls from Groovy code go through a metaclass.
- Metaclasses can change for all instances of a class or per a single instance.
- Metaclass changes affect all future instances in all running threads.
- Metaclasses allow nonintrusive changes to both Groovy and Java code as long as the caller is Groovy. We can even change access to final classes like `java.lang.String`.
- Metaclass changes can take the form of property accessors (pretending property access), operator methods, `GroovyObject` methods, or MOP hook methods.
- `ExpandoMetaClass` makes metaclass modifications more convenient.
- Metaclass changes are best applied only once, preferably at application startup time.

The last point directly leads us to another concept of dynamic programming in Groovy. `ExpandoMetaClass` isn't designed for easily removing a once dynamically added method or undoing any other change. For such temporary changes, Groovy provides category classes.

8.4.5 Temporary MOP modifications using category classes

Metaclasses are the main workhorses for dynamic programming in Groovy but sometimes you don't need their full power and would prefer an alternative that's small and focused and confined to the current thread and a small piece of code. This is exactly what category classes are. We'll look into how you use existing category classes, what benefits they bring, and how to write your own ones.

Using a category class is trivial. Groovy adds a `use` method to `java.lang.Object` that takes two parameters: a category class (or any number thereof) and a closure:

```
use CategoryClass, {
    // new methods are available
}
// new methods are no longer available
```

While the closure is executed, the MOP is modified as defined by the category. After the closure execution is finished, the MOP is reset to its old state.

Listing 8.18 leads us through two examples of using a category: a `TimeCategory` that's part of Groovy and the `java.util.Collections` class.

`TimeCategory` allows simplified working with date, time, and duration for both, easier definition and easier calculation. If you have an appointment in two weeks, you can find the date with `2.weeks.from.today`.

Collections is the unmodified class from the JDK. It contains a number of static helper methods.

Listing 8.18 How to use existing categories like `TimeCategory` and `Collections`

```
import groovy.time.TimeCategory

def janFirst1970 = new Date(0)
use TimeCategory, {
    Date    xmas = janFirst1970 + 1.year - 7.days
    assert xmas.month == Calendar.DECEMBER
    assert xmas.date  == 25
}

use Collections, {
    def list    = [0, 1, 2, 3]
    list.rotate 1
    assert list == [3, 0, 1, 2]
}
```

Inside the closures, we have new properties on numbers (`1.year`), new operator methods for calculating dates, and a new `rotate` method on lists. Outside the closures, no such feature is visible. Note that `janFirst1970` was constructed before the `use` closure.

Category classes are by no means special. Neither do they implement a certain interface nor do they inherit from a certain class. They aren't configured or registered anywhere! They just happen to contain static methods with at least one parameter.

When a class is used as an argument to the `use` method, it becomes a category class and every static method like

```
static ReturnType methodName(Receiver self, optionalArgs) {...}
```

becomes available on the receiver as if the `Receiver` had an instance method like

```
ReturnType methodName(optionalArgs) {...}
```

As always, an example says it better than any explanation. Listing 8.19 defines a class `Marshal` with static methods to marshal and unMarshal an integer to and from a string. The string version may be used for sending the integer to a remote machine. When we use the `Marshal` category class, we can call `marshal()` on an integer and `unMarshal()` on a string.

Listing 8.19 Running a category to marshal and unMarshal integers to/from strings

```
class Marshal {
    static String marshal(Integer self) {
        self.toString()
    }
    static Integer unMarshal(String self) {
        self.toInteger()
    }
}
```

```

use Marshal, {
  assert 1.marshal() == "1"
  assert "1".unMarshal() == 1
  [Integer.MIN_VALUE, -1, 0, Integer.MAX_VALUE].each {
    assert it.marshal().unMarshal() == it
  }
}

```

Naming the receiver object `self` is just a convention. You can use any name you want. Groovy's design decision of using static methods to implement category behavior has a few beneficial effects.

- You're much less likely to run into concurrency issues, because there's less shared state.
- You can use a plethora of classes as categories even if they've been implemented without knowing about Groovy. `Collections` was just an example of many classes with static methods that reside in widely used helper libraries.
- They can easily be created in Groovy, Java, or any other JVM language that produces classes and static methods.

Category classes are a good place to collect methods that work conjointly on different types, such as `Integer` and `String`, to accomplish a feature like marshaling.

Key characteristics of using category classes are:

- The `use` method applies categories to the runtime scope of the closure (as opposed to the lexical scope). That means you can extract code from the closure into a method and call the method from inside the closure.
- Category use is confined to the current thread.
- Category use is nonintrusive.
- If the receiver type refers to a superclass or even an interface, then the method will be available in all subclasses/implementors without further configuration.
- Category method names can well take the form of property accessors (pretending property access), operator methods, and `GroovyObject` methods. MOP hook methods cannot be added through a category class.¹⁰
- Category methods can override method definitions in the metaclass.
- Where performance is crucial, use categories with care and measure their influence.
- Categories cannot introduce a new state in the receiver object; they cannot add new properties with a backing field.

The last point reveals that even though categories are a great tool for combining behavior into reusable features they do have their limitations when it comes to sharing state.

¹⁰ This is a restriction as of Groovy 2.4. The feature may become available in later versions.

In addition to the way that you've seen here, there are two more ways to bring in category methods: as extension methods and through the `@Category` annotation.

8.4.6 **Writing extension modules**

Extension modules can be seen as categories that are always visible: you don't need to call `use` to enable the methods. Just like Groovy enriches the JDK classes with custom methods, you can make your categories globally visible and make them behave like methods from the GDK. One of the most interesting use cases for this is that you can bundle such extension modules into their own JAR file and make them available to other programs just by adding the JAR file to your classpath.

Converting a category into an extension module is straightforward. Imagine that you want to use the `Marshal` category defined in listing 8.19 without having to explicitly use the category. To achieve that, you only need two steps:

- 1 Write the `Marshal` class into its own source file (we'll place it in `regina/Marshal.groovy`).
- 2 Write an extension module descriptor and make it available on a classpath.

The first step is straightforward, but what is the descriptor file? You need to create a file named `org.codehaus.groovy.runtime.ExtensionModule` and ensure it's found in the `META-INF/services` folder of your JAR. This file is used internally by Groovy to load your extension module and make the category transparently available. The descriptor file consists of four entries:

```
moduleName=regina-marshall
moduleVersion=1.0
extensionClasses=regina.Marshal
staticExtensionClasses=
```

The `moduleName` and `moduleVersion` entries are used by Groovy when the runtime is initialized. If two versions of a module of the same name are found on the classpath, the module will not be loaded and an error will be thrown. The `extensionClasses` entry is a comma-separated list of category-like classes. This means that you can define multiple categories in a single extension module. Here there's only one extension class, so the line only contains the fully qualified name of the category class.

Interestingly, extension modules allow you to define static extension methods (add static methods to existing classes). In that case, the static methods must be defined in a separate class, but are written in the same way.

As you can see, bundling extension modules is very easy: the minimum that's required is a descriptor file. One could decide, for example, to write an extension module for the very famous `StringUtils` class from Apache Commons.¹¹ In that case, you wouldn't need to put the `StringUtils` class into your JAR file. All that's needed is a descriptor file, which makes the `StringUtils` class available as an extension module!

¹¹ Commons Lang provides helper utilities for the `java.lang` API, including `String` manipulation methods missing from `java.lang.String`. See <http://commons.apache.org/proper/commons-lang/>.

```

moduleName=apache-commons-stringutils
moduleVersion=3.2
extensionClasses=org.apache.commons.lang3.StringUtils

```

The use of the category format makes extension modules very appealing because it's a common pattern to find utility classes in the form of static methods. Commons Lang is just one example. Extension modules have an interesting advantage compared to categories: while the latter are totally dynamic, the former are statically bound (they're known when Groovy initializes), making them compatible with type checking and static compilation (see chapter 10 for details).

8.4.7 Using the @Category annotation

With @Category, you write your class as if it were an instance class but the annotation adjusts it to have the required format needed for categories, meaning, methods are made static and the `self` parameter, such as you saw in listing 18.9, is automatically added. The following listing shows how we might rewrite listing 18.9 to use the category annotation.

Listing 8.20 Using @Category to create your own category

```

@Category(Integer)
class IntegerMarshal {
    String marshal() {
        toString()
    }
}

@Category(String)
class StringMarshal {
    Integer unMarshal() {
        this.toInteger()
    }
}

use ([IntegerMarshal, StringMarshal]) {
    assert 1.marshal() == "1"
    assert "1".unMarshal() == 1
}

```

Diagram annotations:

- 1** Specifies the type of self: Points to `@Category(Integer)`.
- 2** Implicit this: Points to `toString()` in the `marshal()` method of `IntegerMarshal`.
- 3** Explicit this: Points to `this.toInteger()` in the `unMarshal()` method of `StringMarshal`.
- 4** List variant of use: Points to `[IntegerMarshal, StringMarshal]` in the `use` block.

The @Category annotation can only be used for creating categories associated with a single class; therefore we split our category into two. First up is our category methods for Integer, though in this case there's only one. We place that method in an IntegerMarshal class and annotate it with @Category(Integer) ❶. Methods aren't written as static methods but in instance form with no `self` parameter required. Implicit ❷ and explicit ❸ references to `this` are automatically changed into `self` reference. Using our category class is the same as before, though in this case we now have two category classes so we use the list variant of the method ❹.

That finishes our discussion of categories. Next up are Mixins, which are the final topic in our dynamic programming tour.

8.4.8 Merging classes with Mixins

Have you ever noticed that in Java many interface names (Appendable, Adjustable, Activatable, Callable, Cloneable, Closeable, and many more) end with “-able”? That’s because they refer to an *ability*.

An object may have many abilities and so its class may implement many interfaces, but reusing implementations of any such ability is restricted to only one superclass. In Java and Groovy alike, you can only inherit once even though you can implement many interfaces.

NOTE Using inheritance for reuse of an ability implementation is often frowned upon. Instead of implementation reuse, it’s considered good object-oriented design to only use inheritance if there’s a true is-a relationship between the subclass and superclass.

If you have a superclass A with a subclass B then any object of class B isn’t only a B, it also *is* an A! The definitions of A and B typically reside in different files.¹² The situation looks as if A and B would be merged when constructing an instance of B. They share both state and behavior.

This class merging by inheritance is pretty restricted in Java.

- You cannot use it when inheritance has already been used for other purposes.
- You cannot merge (inherit from) more than one class.
- It’s intrusive. You have to change the class definition.
- You cannot do it with final classes.

Groovy provides a feature called Mixin that addresses exactly these limitations. This feature comes in two flavors. The first flavor is the `@Mixin` class annotation. This feature is now deprecated because the more powerful trait mechanism (see section 7.3.4) provides a better alternative, but we’ll describe it anyway for users of older versions of Groovy where traits didn’t exist. The second flavor is the `mixIn` method, which is available on any class or metaclass. We’ll discuss this after `@Mixin`.

The following listing uses the `@Mixin` class annotation to mix reusable state and behavior into a test case that uses inheritance to be recognized by the testing framework.

Listing 8.21 Mixing a feature into a test case by using the `@Mixin` annotation

```
@Mixin(MessageFeature)
class FirstTest extends GroovyTestCase {
    void testWithMixinUsage() {
        message = "Called from Test"
        assertEquals "Called from Test"
    }
}
```

¹² Because of Java’s late binding, you cannot even be sure that the A that was available at compile time is the same A that’s used at runtime. Java is much more dynamic than many might assume.

```
class MessageFeature {
    def message
    void assertMessage(String msg) {
        assertEquals msg, message
    }
}
```

Note that you can execute listing 8.21 as a script and it will run the test case with the bundled JUnit. Test frameworks for both unit and functional tests tend to use inheritance a lot even though this is no longer considered good framework design. Inheritance makes it more difficult to nicely factor out common state and behavior. With Mixins, you can circumvent this restriction. They make a good companion for unit tests with JUnit and functional tests with Canoo WebTest.

Using the @Mixin annotation is intrusive. You have to change the code of the class that receives the new features. Listing 8.22, in contrast, works nonintrusively. It calls the mixin method on the ArrayList type to mix in two different features that “sieve” factors of 2 or any other number from a list of numbers. Such a feature is helpful when implementing the Sieve of Eratosthenes¹³ to efficiently find prime numbers.

Listing 8.22 Mixing in multiple sieve features nonintrusively

```
class EvenSieve {
    def getNo2() {
        removeAll { it % 2 == 0 }
        return this
    }
}
class MinusSieve {
    def minus(int num) {
        removeAll { it % num == 0 }
        return this
    }
}

ArrayList.mixin EvenSieve, MinusSieve

assert (0..10).toList().no2 - 3 - 5 == [1, 7]
```

You see that we can mix in multiple classes (EvenSieve, MinusSieve) with property accessor methods (getNo2) and operator methods (minus).

The surprising part is how easily the sieve classes implement their feature methods as if they were of type ArrayList themselves, which they aren’t. Even the return value this refers to the actual ArrayList instance when the method is called from the ArrayList, but not when you’re looking at this from inside the feature method.

Mixins are often compared with multiple inheritance but they’re of a different nature. In the first place, our ArrayList doesn’t become a subtype of MinusSieve.

¹³ The Sieve of Eratosthenes is an ancient algorithm for finding all prime numbers up to any given limit. See http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes.

Any instance of test will fail. There's no is-a relationship and no polymorphism. You can use enforced type coercion with the `as` operator, though.

Unlike many models of multiple inheritance, the mixing in of new features always happens in traceable sequence and, in case of conflicts, the latest addition wins. Mixins work like metaclass changes in that respect.

To sum it up, here are the important characteristics of Mixins:

- You can instantiate objects from a blend of many classes. The object's state and behavior encompasses all properties and methods of all mixed classes.
- There's an intrusive use with the `@Mixin` class annotation and a nonintrusive use with the `mixin` method on classes. Both alternatives happen at runtime (as opposed to compile time). `@Mixin` happens at class construction time in a static initializer.
- Mixins are visible in all threads.
- There are no restrictions on what methods to mix in. Property accessors, operator methods, `GroovyObject` methods, and even MOP hook methods all work fine.
- You can mix into superclasses and interfaces.
- A Mixin can override a method of a previous Mixin but not methods in the metaclass.
- There's no per-instance Mixin. You can only mix into classes and metaclasses. To achieve the effect of a per-instance Mixin, you can mix into a per-instance metaclass.
- Mixins cannot easily be undone.

In general, Mixins are designed for sharing features while not modifying any existing behavior of the receiver. Features can build on top of each other and merge and blend with the receiver. And remember traits from section 7.3.4, which provide an alternative approach to some of the problems which Mixins solve.

MOP priorities

It's always good advice to keep things simple. With dynamic programming one can easily go overboard by doing too much, such as using category classes, metaclass changes, and Mixins in combination. If you do anyway, then categories are looked at first, then the metaclass, and finally the Mixins:

```
category class > meta class > mixin
```

But this only applies to methods that are defined for the same class and have the same parameter types. Otherwise, the rules for method dispatch by class/superclass/interface take precedence.

NOTE In case of multiple method definitions, a category class shadows a previously applied category class. Changes to an `ExpandoMetaClass` override previously added methods in that metaclass. Later applied Mixins shadow previously applied Mixins.

That's it for the technical description of Groovy's dynamic programming devices. We discussed quite a number of different concepts for you to understand and remember. Their real value will become apparent when you use them in practice and the following use cases may give you some inspiration for when and how to try dynamic programming yourself.

8.5 Real-world dynamic programming in action

After having seen the various means of dynamic programming in Groovy you may ask yourself how this applies to real-world projects. If you haven't seen much dynamic programming in your career so far, you may even ask whether it's valuable at all because, apparently, you've been able to live without it so far.

This section presents five scenarios that we've derived from working experience with Groovy. They're taken from real codebases with minor modifications. We'll always start with explaining the task so that you can take it as an exercise to come up with your own solution. Then we'll present a solution and talk about the design rationale. We start simple and proceed to the more complex.

8.5.1 Calculating with metrics

We always do silly mistakes when calculating with measurements that have a different order of magnitude. How many nanoseconds are there in a second? Hmm, we must concede that we'd rather look it up than guessing.

But Groovy can help us. Let's take meters, centimeters, and millimeters as a simple example. If we could simply write `"1.m + 20.cm - 8.mm"`, that would be much easier than calculating with 1,192 millimeters.

The task is to make this possible. Calculations will be done in millimeters. The feature will be ubiquitously available.

The following listing addresses the requirements by adding the respective property accessor methods.

Listing 8.23 Metric calculations that avoid common magnitude mistakes

```
Number.metaClass {
    getMm = { delegate }
    getCm = { delegate * 10.mm }
    getM  = { delegate * 100.cm }
}

assert 1.m + 20.cm - 8.mm == 1.192.m
```

We chose a metaclass modification as the vehicle to introduce the new getters for the remainder of the program. We add them to the `Number` interface to not only accommodate `Integers` but also `Doubles`, `Floats`, and so on.

Note that from inside one new feature method we can call the others. Specifying that one meter is 100 cm is more obvious than trying to specify a meter in terms of millimeters.

A solution like the example can be found in many DSLs. You'll find more examples in chapter 18.

8.5.2 *Replacing constructors with factory methods*

New objects are usually constructed by using the `new` keyword and a constructor as in `new Integer(42)`. Many question this language design and you often hear the advice to favor factory methods over direct constructor calls.

The task is to change the Groovy language so that every class can be constructed by a static factory method called `make` with the same parameters as the respective constructor (for example, `Integer.make(42)` will replace `new Integer(42)`).

The next listing goes for a solution that's essentially a one-liner, even though it's typeset on three lines for better reading. It tests itself with factory methods that take zero, one, and two parameters.

Listing 8.24 Introducing static factory methods to all classes

```
import java.awt.Dimension

Class.metaClass.make = { Object[] args ->
    delegate.metaClass.invokeConstructor(*args)
}

assert new HashMap() == HashMap.make()
assert new Integer(42) == Integer.make(42)
assert new Dimension(2, 3) == Dimension.make(2, 3)
```

Quite obviously, we have to introduce the `make` method on some metaclass. But which one? It will be available on every class—on every instance of `java.lang.Class`. Therefore, we add it to the metaclass of the `Class` class.¹⁴

Invoking the constructor is done dynamically; that is, on the metaclass of the current `Class` object, which we refer to as the `delegate`. To allow any number of parameters we use varargs (`Object[]`) in the closure parameter list and spread all arguments over the `invokeConstructor` argument list with the spread operator (`*args`).

This has been a tiny change and we've seemingly changed all classes in the system! That's the true power of dynamic programming. Try this with a static language.

Our example has a number of real-world uses. The Ruby language, for example, solely relies on this approach to constructing objects. Tammo Freese first explored the solution when he, Johannes Link, and I (Dierk) designed our “Groovy in a Day” workshop.

8.5.3 *Fooling IDEs for fun and profit*

Imagine you had a set of components that you have to connect. One component's output channel will be connected to another component's input channel. Let's call the process of defining the connections *wiring*.

¹⁴ If you've gone cross-eyed by now, don't worry. That's a healthy reaction. Rereading and understanding the last paragraph will improve your nerd level at the possible risk of compromising your common sense.

You could do the wiring by maintaining a list of pairs where every pair reflects one connection between a source and a target component. But you don't get much IDE support when you create such pairs.

The task is to allow an approach to wiring that gives you IDE support and checks for assignable types such that only channels of assignable types are wired. All components should remain untouched in the wiring process.

The next listing comes up with a solution that fools your IDE into thinking that there would be property assignments while you actually intercept the assignment and only register the call for the wiring. Depending on the quality of your IDE support, it will check the assignment statements for assignable types and will suggest only those.

Listing 8.25 Temporarily faking property assignments for configuration purposes

```
interface ChannelComponent {}
class Producer implements ChannelComponent {
    List<Integer> outChannel
}
class Adaptor implements ChannelComponent {
    List<Integer> inChannel
    List<String> outChannel
}
class Printer implements ChannelComponent {
    List<String> inChannel
}

class WiringCategory {
    static connections = []
    static setInChannel(ChannelComponent self, value){
        connections << [target:self, source:value]
    }
    static getOutChannel(ChannelComponent self){
        self
    }
}

Producer producer = new Producer()
Adaptor adaptor = new Adaptor()
Printer printer = new Printer()

use WiringCategory, {
    adaptor.inChannel = producer.outChannel
    printer.inChannel = adaptor.outChannel
}

assert WiringCategory.connections == [
    [source: producer, target: adaptor],
    [source: adaptor, target: printer]
]
```

Intercepts assignments

Fakes assignments

Because the components will remain untouched, you use a category class for the scope of the wiring. The assignments are intercepted by overriding the respective property getter and setter methods nonintrusively on the common interface of all components.

The solution is a simplified version of the wiring in the PillarOne project (www.pillarone.org). PillarOne is an open source project for risk calculation in the insurance industry. It makes heavy use of Groovy for specifying risk models made from wired components.

8.5.4 Undoing metaclass modifications

Modifying a metaclass is simple. Undoing such a modification can be a bit involved, though. The task is to try various approaches and to start with an experiment that modifies the `size()` method of `String` such that it returns twice the actual value by referring to the old implementation. Later you want to set the `size()` method back to the original behavior.

The next listing searches the metaclass of `String` for the `MetaMethod` of `size()` and stores it for later reference. A `MetaMethod` has an `invoke` method that takes the receiver object as the first parameter.

Listing 8.26 Method aliasing and undoing metaclass modifications

```
MetaClass oldMetaClass = String.metaClass
MetaMethod alias = String.metaClass.metaMethods
    .find { it.name == 'size' }
String.metaClass {
    oldSize = { -> alias.invoke delegate }
    size    = { -> oldSize() * 2 }
}

assert "abc".size()    == 6
assert "abc".oldSize() == 3

if (oldMetaClass.is(String.metaClass)) {
    String.metaClass {
        size    = { -> alias.invoke delegate }
        oldSize = { -> throw new UnsupportedOperationException() }
    }
} else {
    String.metaClass = oldMetaClass
}

assert "abc".size() == 3
```

1 Stores old metaclass

2 Stores MetaMethod

3 Reverses modification

4 Resets metaclass

When overriding a method on the metaclass, there's nothing like "super" that we'd have used in subclasses to refer to an original implementation in a superclass. As a replacement, we introduce a new method `oldSize()` as an alias for the **2** old method so that we can refer to it.

Undoing that modification comes in two flavors: doing a **3** reverse modification or **4** setting the metaclass instance back to the original instance **1** in case the instance has changed. If before the modification the default metaclass was in use, then it was changed into an `ExpandoMetaClass` with the first modification and we can reset to the old metaclass. Otherwise, we've already started with an `ExpandoMetaClass` and only modified that instance.

Resetting the metaclass instance is the cleaner way but it's only available if there were no changes to the metaclass of `String` before we started. The code in listing 8.26 is again a simplified version of metaclass handling in the `PillarOne` project.

8.5.5 The Intercept/Cache/Invoke pattern

The `methodMissing` hook method is a cornerstone of the MOP. Some people even define dynamic programming by the availability of such a method. But it comes at a cost. Because Groovy first tries all other possibilities of finding a suitable method before it finally calls `methodMissing`, this requires some time. It's also very common that a method that has been called once will be called again.

The task is to step into `methodMissing` at most once for every distinct method call. For example, we want to support methods of the form `findBy<propertyName>(value)` that searches any collective datatype for items that have a property of that name with the given value. We seek an optimized and nonintrusive version of listing 8.2.

The following listing searches a list of maps for planets with a given name or average distance from Earth in astronomical units (rounded).

Listing 8.27 The Intercept/Cache/Invoke pattern for finding by property value

```
ArrayList.metaClass.methodMissing = { String name, Object args ->
    assert name.startsWith("findBy")
    assert args.size() == 1
    Object.metaClass."$name" = { value ->
        delegate.find { it[name.toLowerCase()-'findby'] == value }
    }
    delegate."$name" (args[0])
}

def data = [
    [name: 'moon',    au: 0.0025],
    [name: 'sun',     au: 1      ],
    [name: 'neptune', au: 30     ],
]

assert data.findByName('moon')
assert data.findByName('sun')
assert data.findByAu(1)
```

We add the `methodMissing` hook to the metaclass of `ArrayList`. Whenever we enter the hook method we add a new method of the requested name to our metaclass ①. For this new method, the missing hook method will never be called again, because it's no longer missing. We've *synthesized* a new method.

We also need to execute the synthesized method, which we do in ②.

The Intercept/Cache/Invoke pattern was invented by Graeme Rocher, the project lead of the Grails web platform. It's a core part of the Grails infrastructure. The productive version is a bit more elaborate than our example, mainly to work nicely in highly concurrent environments, but the general approach is the same.

8.6 **Summary**

We hope that by now you've gained a good overview of the concepts that allow dynamic programming with Groovy. These language capabilities may have been new to you and thus unfamiliar and maybe even daunting.

But even if they appear like magic, they're all easily explained by the fact that Groovy sees the world through the glasses of the MOP. The MOP itself offers many alternatives for adapting it to new necessities.

We can use the MOP hook methods intrusively or apply nonintrusive changes by switching metaclasses, modifying metaclasses, using categories, or mixing in a new state and behavior. All these devices come in combination with the Groovy method dispatch, property handling, operator methods, `GroovyObject` methods, and inheritance. The pervasive use of closures adds another dimension of dynamically changing behavior at runtime.

Once you've experienced the merits of dynamic programming, you'll find it unwieldy to go back to a static language.

You may be surprised to hear that the topic of dynamic programming isn't over, yet. What we've covered so far is the runtime aspect of it. But there are also compile-time aspects that we'll explore in the next chapter.