Docs

[User Manual](http://docs.google.com/userguide/userguide.html)

[Guides and Tutorials](https://guides.gradle.org)

[DSL Reference](http://docs.google.com/dsl/)

[Javadoc](http://docs.google.com/javadoc/)

[Release Notes](http://docs.google.com/release-notes.html)

[Forums](https://discuss.gradle.org/)

[Training](https://gradle.org/training/)

[Try Gradle Enterprise](https://gradle.com/enterprise)

[PDF](http://docs.google.com/userguide/userguide.pdf)

* [User Manual Home](http://docs.google.com/userguide/userguide.html)
* [Release Notes](http://docs.google.com/release-notes.html)
* [Installing Gradle](http://docs.google.com/userguide/installation.html)
* [Tutorials](https://guides.gradle.org/)

### Reference

* [Groovy DSL Reference](http://docs.google.com/dsl/)
* [Gradle API Javadoc](http://docs.google.com/javadoc/)
* [Core Plugins](http://docs.google.com/userguide/plugin_reference.html)
* [Gradle & Third-party Tools](http://docs.google.com/userguide/third_party_integration.html)

### Getting Started

* [Creating New Gradle Builds](https://guides.gradle.org/creating-new-gradle-builds/)
* [Creating Build Scans](https://guides.gradle.org/creating-build-scans/)
* [Migrating From Maven](https://guides.gradle.org/migrating-from-maven/)

### Running Gradle Builds

* [Command-Line Interface](http://docs.google.com/userguide/command_line_interface.html)
* [Customizing Execution](#gjdgxs)
  + [Configuring the Build Environment](http://docs.google.com/userguide/build_environment.html)
  + [Configuring the Gradle Daemon](http://docs.google.com/userguide/gradle_daemon.html)
  + [Initialization Scripts](http://docs.google.com/userguide/init_scripts.html)
* [Directory Layout](http://docs.google.com/userguide/directory_layout.html)
* [Executing Multi-Project Builds](http://docs.google.com/userguide/intro_multi_project_builds.html)
* [Gradle Wrapper](http://docs.google.com/userguide/gradle_wrapper.html)
* [Troubleshooting](http://docs.google.com/userguide/troubleshooting.html)
* [Using Build Scans](https://docs.gradle.com/build-scan-plugin)
* [Enabling and Configuring the Build Cache](http://docs.google.com/userguide/build_cache.html)
* [Integrating Separate Gradle Builds (Composite Builds)](http://docs.google.com/userguide/composite_builds.html)

### Authoring Gradle Builds

* [Fundamentals](#30j0zll)
  + [Introducing the Basics of Build Scripts](http://docs.google.com/userguide/tutorial_using_tasks.html)
  + [Working with Tasks](http://docs.google.com/userguide/more_about_tasks.html)
  + [Learning More About Build Scripts](http://docs.google.com/userguide/writing_build_scripts.html)
  + [Working with Files](http://docs.google.com/userguide/working_with_files.html)
  + [Creating Custom Task Types](http://docs.google.com/userguide/custom_tasks.html)
  + [Using Gradle Plugins](http://docs.google.com/userguide/plugins.html)
  + [The Standard Gradle Plugins](http://docs.google.com/userguide/standard_plugins.html)
  + [Understanding the Build Lifecycle](http://docs.google.com/userguide/build_lifecycle.html)
  + [Working with Logging](http://docs.google.com/userguide/logging.html)
  + [Configuring Multi-Project Builds](http://docs.google.com/userguide/multi_project_builds.html)
* [Best Practices](#1fob9te)
  + [Authoring Maintainable Build Scripts](http://docs.google.com/userguide/authoring_maintainable_build_scripts.html)
  + [Organizing Gradle Projects](http://docs.google.com/userguide/organizing_gradle_projects.html)
  + [Optimizing Build Performance](https://guides.gradle.org/performance/)
  + [Using the Build Cache](https://guides.gradle.org/using-build-cache/)
* [Dependency Management](#3znysh7)
  + [Introduction to Dependency Management](http://docs.google.com/userguide/introduction_dependency_management.html)
  + [Dependency Management Terminology](http://docs.google.com/userguide/dependency_management_terminology.html)
  + [Dependency Types](http://docs.google.com/userguide/dependency_types.html)
  + [Repository Types](http://docs.google.com/userguide/repository_types.html)
  + [Declaring Dependencies](http://docs.google.com/userguide/declaring_dependencies.html)
  + [Declaring Repositories](http://docs.google.com/userguide/declaring_repositories.html)
  + [Inspecting Dependencies](http://docs.google.com/userguide/inspecting_dependencies.html)
  + [Managing Dependency Configurations](http://docs.google.com/userguide/managing_dependency_configurations.html)
  + [Managing Transitive Dependencies](http://docs.google.com/userguide/managing_transitive_dependencies.html)
  + [Dependency Locking](http://docs.google.com/userguide/dependency_locking.html)
  + [Troubleshooting Dependency Resolution](http://docs.google.com/userguide/troubleshooting_dependency_resolution.html)
  + [Customizing Dependency Resolution Behavior](http://docs.google.com/userguide/customizing_dependency_resolution_behavior.html)
  + [Dependency Cache Internals](http://docs.google.com/userguide/dependency_cache.html)
  + [Working with Dependencies](http://docs.google.com/userguide/working_with_dependencies.html)
* [Publishing Artifacts](http://docs.google.com/userguide/artifact_management.html)
* [C++ Projects](#2et92p0)
  + [Building Native Software](http://docs.google.com/userguide/native_software.html)
  + [Software Model Concepts](http://docs.google.com/userguide/software_model_concepts.html)
  + [Rule-based Model Configuration](http://docs.google.com/userguide/software_model.html)
  + [Implementing Model Rules in a Plugin](http://docs.google.com/userguide/rule_source.html)
  + [Extending the Software Model](http://docs.google.com/userguide/software_model_extend.html)
* [Java Projects](#tyjcwt)
  + [Building Java & JVM projects](http://docs.google.com/userguide/building_java_projects.html)
  + [Testing Java & JVM projects](http://docs.google.com/userguide/java_testing.html)
* [Advanced Techniques](#3dy6vkm)
  + [Configuring Tasks Lazily](http://docs.google.com/userguide/lazy_configuration.html)
  + [Developing Parallel Tasks](https://guides.gradle.org/using-the-worker-api/)
  + [Testing Your Build with TestKit](http://docs.google.com/userguide/test_kit.html)
  + [Using Ant from Gradle](http://docs.google.com/userguide/ant.html)
* [Sample Gradle builds](#1t3h5sf)
  + [Groovy DSL Samples](https://github.com/gradle/gradle/tree/master/subprojects/docs/src/samples)
  + [Kotlin DSL Samples](https://github.com/gradle/kotlin-dsl/tree/master/samples)

### Extending Gradle

* [Writing Custom Plugins](http://docs.google.com/userguide/custom_plugins.html)
* [Plugin Development Guides](https://gradle.org/guides/?q=Plugin+Development)

[Edit this page](https://github.com/gradle/gradle/edit/master/subprojects/docs/src/docs/userguide/)

# Rule based model configuration

Contents

[Background](#4d34og8)

[Basic Concepts](#2s8eyo1)

[Rule sources](#17dp8vu)

[Advanced Concepts](#3rdcrjn)

[The model DSL](#26in1rg)

[The model report](#lnxbz9)

[Limitations and future direction](#35nkun2)

| **⚠** | Rule based configuration [will be deprecated](https://blog.gradle.org/state-and-future-of-the-gradle-software-model). New plugins should not use this concept. |
| --- | --- |

Rule based model configuration enables *configuration logic to itself have dependencies* on other elements of configuration, and to make use of the resolved states of those other elements of configuration while performing its own configuration.

[Background](#4d34og8)

In a nutshell, the Software Model is a very declarative way to describe how a piece of software is built and the other components it needs as dependencies in the process. It also provides a new, rule-based engine for configuring a Gradle build. When we started to implement the software model we set ourselves the following goals:

* Improve configuration and execution time performance.
* Make customizations of builds with complex tool chains easier.
* Provide a richer, more standardized way to model different software ecosystems.

Gradle drastically improved configuration performance through other measures. There is no longer any need for a drastic, incompatible change in how Gradle builds are configured. Gradle’s support for building [native software](http://docs.google.com/native_software.html#native_software) and [Play Framework applications](http://docs.google.com/play_plugin.html#play_plugin) still use the configuration model.

[Basic Concepts](#2s8eyo1)

[The “model space”](#1ksv4uv)

The term “model space” is used to refer to the formal model, which can be read and modified by rules.

A counterpart to the model space is the “project space”, which should be familiar to readers. The “project space” is a graph of objects (e.g project.repositories, project.tasks etc.) having a [Project](http://docs.google.com/dsl/org.gradle.api.Project.html) as its root. A build script is effectively adding and configuring objects of this graph. For the most part, the “project space” is opaque to Gradle. It is an arbitrary graph of objects that Gradle only partially understands.

Each project also has its own model space, which is distinct from the project space. A key characteristic of the “model space” is that Gradle knows much more about it (which is knowledge that can be put to good use). The objects in the model space are “managed”, to a greater extent than objects in the project space. The origin, structure, state, collaborators and relationships of objects in the model space are first class constructs. This is effectively the characteristic that functionally distinguishes the model space from the project space: the objects of the model space are defined in ways that Gradle can understand them intimately, as opposed to an object that is the result of running relatively opaque code. A “rule” is effectively a building block of this definition.

The model space will eventually replace the project space, becoming the only “space”.

[Rules](#44sinio)

The model space is defined by “rules”. A rule is just a function (in the abstract sense) that either produces a model element, or acts upon a model element. Every rule has a single subject and zero or more inputs. Only the subject can be changed by a rule, while the inputs are effectively immutable.

Gradle guarantees that all inputs are fully “realized“ before the rule executes. The process of “realizing” a model element is effectively executing all the rules for which it is the subject, transitioning it to its final state. There is a strong analogy here to Gradle’s task graph and task execution model. Just as tasks depend on each other and Gradle ensures that dependencies are satisfied before executing a task, rules effectively depend on each other (i.e. a rule depends on all rules whose subject is one of the inputs) and Gradle ensures that all dependencies are satisfied before executing the rule.

Model elements are very often defined in terms of other model elements. For example, a compile task’s configuration can be defined in terms of the configuration of the source set that it is compiling. In this scenario, the compile task would be the subject of a rule and the source set an input. Such a rule could configure the task subject based on the source set input without concern for how it was configured, who it was configured by or when the configuration was specified.

There are several ways to declare rules, and in several forms.

[Rule sources](#17dp8vu)

One way to define rules is via a [RuleSource](http://docs.google.com/javadoc/org/gradle/model/RuleSource.html) subclass. If an object extends RuleSource and contains any methods annotated by '@Mutate', then each such method defines a rule. For each such method, the first argument is the subject, and zero or more subsequent arguments may follow and are inputs of the rule.

[Example: applying a rule source plugin](#2jxsxqh)

**build.gradle**

@Managed  
interface Person {  
 void setFirstName(String name)  
 String getFirstName()  
  
 void setLastName(String name)  
 String getLastName()  
}  
  
class PersonRules extends RuleSource {  
 @Model void person(Person p) {}  
  
 //Create a rule that modifies a Person and takes no other inputs  
 @Mutate void setFirstName(Person p) {  
 p.firstName = "John"  
 }  
  
 //Create a rule that modifies a ModelMap<Task> and takes as input a Person  
 @Mutate void createHelloTask(ModelMap<Task> tasks, Person p) {  
 tasks.create("hello") {  
 doLast {  
 println "Hello $p.firstName $p.lastName!"  
 }  
 }  
 }  
}  
  
apply plugin: PersonRules

**Output of** gradle hello

> gradle hello  
  
> Task :hello  
Hello John Smith!  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

Each of the different methods of the rule source are discrete, independent rules. Their order, or the fact that they belong to the same class, do not affect their behavior.

[Example: a model creation rule](#z337ya)

**build.gradle**

@Model void person(Person p) {}

This rule declares that there is a model element at path "person" (defined by the method name), of type Person. This is the form of the [Model](http://docs.google.com/javadoc/org/gradle/model/Model.html) type rule for [Managed](http://docs.google.com/javadoc/org/gradle/model/Managed.html) types. Here, the person object is the rule subject. The method could potentially have a body, that mutated the person instance. It could also potentially have more parameters, which would be the rule inputs.

[Example: a model mutation rule](#3j2qqm3)

**build.gradle**

//Create a rule that modifies a Person and takes no other inputs  
@Mutate void setFirstName(Person p) {  
 p.firstName = "John"  
}

This [Mutate](http://docs.google.com/javadoc/org/gradle/model/Mutate.html) rule mutates the person object. The first parameter to the method is the subject. Here, a by-type reference is used as no [Path](http://docs.google.com/javadoc/org/gradle/model/Path.html) annotation is present on the parameter. It could also potentially have more parameters, that would be the rule inputs.

[Example: creating a task](#1y810tw)

**build.gradle**

//Create a rule that modifies a ModelMap<Task> and takes as input a Person  
@Mutate void createHelloTask(ModelMap<Task> tasks, Person p) {  
 tasks.create("hello") {  
 doLast {  
 println "Hello $p.firstName $p.lastName!"  
 }  
 }  
}

This [Mutate](http://docs.google.com/javadoc/org/gradle/model/Mutate.html) rule effectively adds a task, by mutating the tasks collection. The subject here is the "tasks" node, which is available as a [ModelMap](http://docs.google.com/javadoc/org/gradle/model/ModelMap.html) of [Task](http://docs.google.com/dsl/org.gradle.api.Task.html). The only input is our person element. As the person is being used as an input here, it will have been realised before executing this rule. That is, the task container effectively *depends on* the person element. If there are other configuration rules for the person element, potentially specified in a build script or other plugin, they will also be guaranteed to have been executed.

As Person is a [Managed](http://docs.google.com/javadoc/org/gradle/model/Managed.html) type in this example, any attempt to modify the person parameter in this method would result in an exception being thrown. Managed objects enforce immutability at the appropriate point in their lifecycle.

Rule source plugins can be packaged and distributed in the same manner as other types of plugins (see [Custom Plugins](http://docs.google.com/custom_plugins.html#custom_plugins)). They also may be applied in the same manner (to project objects) as [Plugin](http://docs.google.com/javadoc/org/gradle/api/Plugin.html) implementations (i.e. via [Project.apply(java.util.Map)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:apply(java.util.Map))).

Please see the documentation for [RuleSource](http://docs.google.com/javadoc/org/gradle/model/RuleSource.html) for more information on constraints on how rule sources must be implemented and for more types of rules.

[Advanced Concepts](#3rdcrjn)

[Model paths](#4i7ojhp)

A model path identifies the location of an element relative to the root of its model space. A common representation is a period-delimited set of names. For example, the model path "tasks" is the path to the element that is the task container. Assuming a task whose name is hello, the path "tasks.hello" is the path to this task.

[Managed model elements](#2xcytpi)

Currently, any kind of Java object can be part of the model space. However, there is a difference between “managed” and “unmanaged” objects.

A “managed” object is transparent and enforces immutability once realized. Being transparent means that its structure is understood by the rule infrastructure and as such each of its properties are also individual elements in the model space.

An “unmanaged” object is opaque to the model space and does not enforce immutability. Over time, more mechanisms will be available for defining managed model elements culminating in all model elements being managed in some way.

Managed models can be defined by attaching the @Managed annotation to an interface:

[Example: a managed type](#1ci93xb)

**build.gradle**

@Managed  
interface Person {  
 void setFirstName(String name)  
 String getFirstName()  
  
 void setLastName(String name)  
 String getLastName()  
}

By defining a getter/setter pair, you are effectively declaring a managed property. A managed property is a property for which Gradle will enforce semantics such as immutability when a node of the model is not the subject of a rule. Therefore, this example declares properties named *firstName* and *lastName* on the managed type *Person*. These properties will only be writable when the view is mutable, that is to say when the *Person* is the subject of a Rule (see below the explanation for rules).

Managed properties can be of any scalar type. In addition, properties can also be of any type which is itself managed:

| **Property type** | **Nullable** | **Example** |
| --- | --- | --- |
| String | Yes | void setFirstName(String name) String getFirstName() |
| File | Yes | void setHomeDirectory(File homeDir) File getHomeDirectory() |
| Integer, Boolean, Byte, Short, Float, Long, Double | Yes | void setId(Long id) Long getId() |
| int, boolean, byte, short, float, long, double | No | void setEmployed(boolean isEmployed) boolean isEmployed()  void setAge(int age) int getAge() |
| Another *managed* type. | Only if read/write | void setMother(Person mother) Person getMother() |
| An *enumeration* type. | Yes | void setMaritalStatus(MaritalStatus status) MaritalStatus getMaritalStatus() |
| A ManagedSet. A managed set supports the creation of new named model elements, but not their removal. | Only if read/write | ModelSet<Person> getChildren() |
| A Set or List of scalar types. All classic operations on collections are supported: add, remove, clear…​ | Only if read/write | void setUserGroups(List<String> groups) List<String> getUserGroups() |

If the type of a property is itself a managed type, it is possible to declare only a getter, in which case you are declaring a read-only property. A read-only property will be instantiated by Gradle, and cannot be replaced with another object of the same type (for example calling a setter). However, the properties of that property can potentially be changed, if, and only if, the property is the subject of a rule. If it’s not the case, the property is immutable, like any classic read/write managed property, and properties of the property cannot be changed at all.

Managed types can be defined out of interfaces or abstract classes and are usually defined in plugins, which are written either in Java or Groovy. Please see the [Managed](http://docs.google.com/javadoc/org/gradle/model/Managed.html) annotation for more information on creating managed model objects.

[Model element types](#3whwml4)

There are particular types (language types) supported by the model space and can be generalised as follows:

Table 1. Type definitions

| **Type** | **Definition** |
| --- | --- |
| Scalar | A scalar type is one of the following:   * a primitive type (e.g. int) or its boxed type (e.g Integer) * a BigInteger or BigDecimal * a String * a File * an enumeration type |
| Scalar Collection | A java.util.List or java.util.Set containing one of the scalar types |
| Managed type | Any class which is a valid managed model (i.e.annotated with @[Managed](http://docs.google.com/javadoc/org/gradle/model/Managed.html)) |
| Managed collection | A [ModelMap](http://docs.google.com/javadoc/org/gradle/model/ModelMap.html) or [ModelSet](http://docs.google.com/javadoc/org/gradle/model/ModelSet.html) |

There are various contexts in which these types can be used:

Table 2. Model type support

| **Context** | **Supported types** |
| --- | --- |
| Creating top level model elements | * Any managed type * [FunctionalSourceSet](http://docs.google.com/javadoc/org/gradle/language/base/FunctionalSourceSet.html) (when the [LanguageBasePlugin](http://docs.google.com/javadoc/org/gradle/language/base/plugins/LanguageBasePlugin.html) plugin has been applied) * Subtypes of [LanguageSourceSet](http://docs.google.com/javadoc/org/gradle/language/base/LanguageSourceSet.html) which have been registered via [ComponentType](http://docs.google.com/javadoc/org/gradle/platform/base/ComponentType.html) |
| Properties of managed model elements | The properties (attributes) of a managed model elements may be one or more of the following:   * A managed type * A type which is annotated with @[Unmanaged](http://docs.google.com/javadoc/org/gradle/model/Unmanaged.html) * A Scalar Collection * A Managed collection containing managed types * A Managed collection containing [FunctionalSourceSet](http://docs.google.com/javadoc/org/gradle/language/base/FunctionalSourceSet.html)'s (when the [LanguageBasePlugin](http://docs.google.com/javadoc/org/gradle/language/base/plugins/LanguageBasePlugin.html) plugin has been applied) * Subtypes of [LanguageSourceSet](http://docs.google.com/javadoc/org/gradle/language/base/LanguageSourceSet.html) which have been registered via [ComponentType](http://docs.google.com/javadoc/org/gradle/platform/base/ComponentType.html) |

[Language source sets](#2bn6wsx)

[FunctionalSourceSet](http://docs.google.com/javadoc/org/gradle/language/base/FunctionalSourceSet.html)s and subtypes of [LanguageSourceSet](http://docs.google.com/javadoc/org/gradle/language/base/LanguageSourceSet.html) (which have been registered via [ComponentType](http://docs.google.com/javadoc/org/gradle/platform/base/ComponentType.html)) can be added to the model space via rules or via the model DSL.

[Example: Strongly modelling sources sets](#qsh70q)

**build.gradle**

apply plugin: 'java-lang'  
  
//Creating LanguageSourceSets via rules  
class LanguageSourceSetRules extends RuleSource {  
 @Model  
 void mySourceSet(JavaSourceSet javaSource) {  
 javaSource.source.srcDir("src/main/my")  
 }  
}  
apply plugin: LanguageSourceSetRules  
  
//Creating LanguageSourceSets via the model DSL  
model {  
 another(JavaSourceSet) {  
 source {  
 srcDir "src/main/another"  
 }  
 }  
}  
  
//Using FunctionalSourceSets  
@Managed  
interface SourceBundle {  
 FunctionalSourceSet getFreeSources()  
 FunctionalSourceSet getPaidSources()  
}  
model {  
 sourceBundle(SourceBundle) {  
 freeSources.create("main", JavaSourceSet)  
 freeSources.create("resources", JvmResourceSet)  
 paidSources.create("main", JavaSourceSet)  
 paidSources.create("resources", JvmResourceSet)  
 }  
}

**Output of gradle help**

> gradle help  
  
> Task :help

| **✨** | The code for this example can be found at samples/modelRules/language-support in the ‘-all’ distribution of Gradle. |
| --- | --- |

[References, binding and scopes](#3as4poj)

As previously mentioned, a rule has a subject and zero or more inputs. The rule’s subject and inputs are declared as “references” and are “bound” to model elements before execution by Gradle. Each rule must effectively forward declare the subject and inputs as references. Precisely how this is done depends on the form of the rule. For example, the rules provided by a [RuleSource](http://docs.google.com/javadoc/org/gradle/model/RuleSource.html) declare references as method parameters.

A reference is either “by-path” or “by-type”.

A “by-type” reference identifies a particular model element by its type. For example, a reference to the [TaskContainer](http://docs.google.com/javadoc/org/gradle/api/tasks/TaskContainer.html) effectively identifies the "tasks" element in the project model space. The model space is not exhaustively searched for candidates for by-type binding; rather, a rule is given a scope (discussed later) that determines the search space for a by-type binding.

A “by-path” reference identifies a particular model element by its path in model space. By-path references are always relative to the rule scope; there is currently no way to path “out” of the scope. All by-path references also have an associated type, but this does not influence what the reference binds to. The element identified by the path must however by type compatible with the reference, or a fatal “binding failure” will occur.

[Binding scope](#1pxezwc)

Rules are bound within a “scope”, which determines how references bind. Most rules are bound at the project scope (i.e. the root of the model graph for the project). However, rules can be scoped to a node within the graph. The [ModelMap.named(java.lang.String, java.lang.Class)](http://docs.google.com/javadoc/org/gradle/model/ModelMap.html#named-java.lang.String,%20java.lang.Class-) method is an example of a mechanism for applying scoped rules. Rules declared in the build script using the model {} block, or via a RuleSource applied as a plugin use the root of the model space as the scope. This can be considered the default scope.

By-path references are always relative to the rule scope. When the scope is the root, this effectively allows binding to any element in the graph. When it is not, then only the children of the scope can be referenced using "by-path" notation.

When binding by-type references, the following elements are considered:

* The scope element itself.
* The immediate children of the scope element.
* The immediate children of the model space (i.e. project space) root.

For the common case, where the rule is effectively scoped to the root, only the immediate children of the root need to be considered.

[Binding to all elements in a scope matching type](#49x2ik5)

Mutating or validating all elements of a given type in some scope is a common use-case. To accommodate this, rules can be applied via the @Each annotation.

In the example below, a @Defaults rule is applied to each FileItem in the model setting a default file size of "1024". Another rule applies a [RuleSource](http://docs.google.com/javadoc/org/gradle/model/RuleSource.html) to every DirectoryItem that makes sure all file sizes are positive and divisible by "16".

[Example: a DSL example applying a rule to every element in a scope](#2p2csry)

**build.gradle**

@Managed interface Item extends Named {}  
@Managed interface FileItem extends Item {  
 void setSize(int size)  
 int getSize()  
}  
@Managed interface DirectoryItem extends Item {  
 ModelMap<Item> getChildren()  
}  
  
class PluginRules extends RuleSource {  
 @Defaults void setDefaultFileSize(@Each FileItem file) {  
 file.size = 1024  
 }  
  
 @Rules void applyValidateRules(ValidateRules rules, @Each DirectoryItem directory) {}  
}  
apply plugin: PluginRules  
  
abstract class ValidateRules extends RuleSource {  
 @Validate  
 void validateSizeIsPositive(ModelMap<FileItem> files) {  
 files.each { file ->  
 assert file.size > 0  
 }  
 }  
  
 @Validate  
 void validateSizeDivisibleBySixteen(ModelMap<FileItem> files) {  
 files.each { file ->  
 assert file.size % 16 == 0  
 }  
 }  
}  
  
model {  
 root(DirectoryItem) {  
 children {  
 dir(DirectoryItem) {  
 children {  
 file1(FileItem)  
 file2(FileItem) { size = 2048 }  
 }  
 }  
 file3(FileItem)  
 }  
 }  
}

| **✨** | The code for this example can be found at samples/modelRules/ruleSourcePluginEach in the ‘-all’ distribution of Gradle. |
| --- | --- |

[The model DSL](#26in1rg)

In addition to using a RuleSource, it is also possible to declare a model and rules directly in a build script using the “model DSL”.

| **💡** | The model DSL makes heavy use of various Groovy DSL features. Please have a read of [Groovy DSL basics](http://docs.google.com/writing_build_scripts.html#groovy-dsl-basics) for an introduction to these Groovy features. |
| --- | --- |

The general form of the model DSL is:

model {  
 «rule-definitions»  
}

All rules are nested inside a model block. There may be any number of rule definitions inside each model block, and there may be any number of model blocks in a build script. You can also use a model block in build scripts that are applied using apply from: $uri.

There are currently 2 kinds of rule that you can define using the model DSL: configuration rules, and creation rules.

[Configuration rules](#147n2zr)

You can define a rule that configures a particular model element. A configuration rule has the following form:

model {  
 «model-path-to-subject» {  
 «configuration code»  
 }  
}

Continuing with the example so far of the model element "person" of type Person being present, the following DSL snippet adds a configuration rule for the person that sets its lastName property.

[Example: DSL configuration rule](#3o7alnk)

**build.gradle**

model {  
 person {  
 lastName = "Smith"  
 }  
}

A configuration rule specifies a path to the subject that should be configured and a closure containing the code to run when the subject is configured. The closure is executed with the subject passed as the closure delegate. Exactly what code you can provide in the closure depends on the type of the subject. This is discussed below.

You should note that the configuration code is not executed immediately but is instead executed only when the subject is required. This is an important behaviour of model rules and allows Gradle to configure only those elements that are required for the build, which helps reduce build time. For example, let’s run a task that uses the "person" object:

[Example: Configuration run when required](#23ckvvd)

**build.gradle**

model {  
 person {  
 println "configuring person"  
 lastName = "Smith"  
 }  
}

**Output of** gradle showPerson

> gradle showPerson  
configuring person  
  
> Task :showPerson  
Hello John Smith!  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

You can see that before the task is run, the "person" element is configured by running the rule closure. Now let’s run a task that does not require the "person" element:

[Example: Configuration not run when not required](#ihv636)

**Output of** gradle somethingElse

> gradle somethingElse  
  
> Task :somethingElse  
Not using person  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

In this instance, you can see that the "person" element is not configured at all.

[Creation rules](#32hioqz)

It is also possible to create model elements at the root level. The general form of a creation rule is:

model {  
 «element-name»(«element-type») {  
 «initialization code»  
 }  
}

The following model rule creates the "person" element:

[Example: DSL creation rule](#1hmsyys)

**build.gradle**

model {  
 person(Person) {  
 firstName = "John"  
 }  
}

A creation rule definition specifies the path of the element to create, plus its public type, represented as a Java interface or class. Only certain types of model elements can be created.

A creation rule may also provide a closure containing the initialization code to run when the element is created. The closure is executed with the element passed as the closure delegate. Exactly what code you can provide in the closure depends on the type of the subject. This is discussed below.

The initialization closure is optional and can be omitted, for example:

[Example: DSL creation rule without initialization](#41mghml)

**build.gradle**

model {  
 barry(Person)  
}

You should note that the initialization code is not executed immediately but is instead executed only when the element is required. The initialization code is executed before any configuration rules are run. For example:

[Example: Initialization before configuration](#2grqrue)

**build.gradle**

model {  
 person {  
 println "configuring person"  
 println "last name is $lastName, should be Smythe"  
 lastName = "Smythe"  
 }  
 person(Person) {  
 println "creating person"  
 firstName = "John"  
 lastName = "Smith"  
 }  
}

**Output of** gradle showPerson

> gradle showPerson  
creating person  
configuring person  
last name is Smith, should be Smythe  
  
> Task :showPerson  
Hello John Smythe!  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

Notice that the creation rule appears in the build script *after* the configuration rule, but its code runs before the code of the configuration rule. Gradle collects up all the rules for a particular subject before running any of them, then runs the rules in the appropriate order.

[Model rule closures](#vx1227)

Most DSL rules take a closure containing some code to run to configure the subject. The code you can use in this closure depends on the type of the subject of the rule.

| **💡** | You can use the [model report](#lnxbz9) to determine the type of a particular model element. |
| --- | --- |

In general, a rule closure may contain arbitrary code, mixed with some type specific DSL syntax.

[ModelMap<T> subject](#3fwokq0)

A [ModelMap](http://docs.google.com/javadoc/org/gradle/model/ModelMap.html) is basically a map of model elements, indexed by some name. When a ModelMap is used as the subject of a DSL rule, the rule closure can use any of the methods defined on the [ModelMap](http://docs.google.com/javadoc/org/gradle/model/ModelMap.html) interface.

A rule closure with ModelMap as a subject can also include nested creation or configuration rules. These behave in a similar way to the creation and configuration rules that appear directly under the model block.

Here is an example of a nested creation rule:

[Example: Nested DSL creation rule](#1v1yuxt)

**build.gradle**

model {  
 people {  
 john(Person) {  
 firstName = "John"  
 }  
 }  
}

As before, a nested creation rule defines a name and public type for the element, and optionally, a closure containing code to use to initialize the element. The code is run only when the element is required in the build.

Here is an example of a nested configuration rule:

[Example: Nested DSL configuration rule](#4f1mdlm)

**build.gradle**

model {  
 people {  
 john {  
 lastName = "Smith"  
 }  
 }  
}

As before, a nested configuration rule defines the name of the element to configure and a closure containing code to use to configure the element. The code is run only when the element is required in the build.

ModelMap introduces several other kinds of rules. For example, you can define a rule that targets each of the elements in the map. The code in the rule closure is executed once for each element in the map, when that element is required. Let’s run a task that requires all of the children of the "people" element:

[Example: DSL configuration rule for each element in a map](#2u6wntf)

**build.gradle**

model {  
 people {  
 john(Person) {  
 println "creating $it"  
 firstName = "John"  
 lastName = "Smith"  
 }  
 all {  
 println "configuring $it"  
 }  
 barry(Person) {  
 println "creating $it"  
 firstName = "Barry"  
 lastName = "Barry"  
 }  
 }  
}

**Output of** gradle listPeople

> gradle listPeople  
creating Person 'people.barry'  
configuring Person 'people.barry'  
creating Person 'people.john'  
configuring Person 'people.john'  
  
> Task :listPeople  
Hello Barry Barry!  
Hello John Smith!  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

Any method on [ModelMap](http://docs.google.com/javadoc/org/gradle/model/ModelMap.html) that accepts an [Action](http://docs.google.com/javadoc/org/gradle/api/Action.html) as its last parameter can also be used to define a nested rule.

[@Managed type subject](#19c6y18)

When a managed type is used as the subject of a DSL rule, the rule closure can use any of the methods defined on the managed type interface.

A rule closure can also configure the properties of the element using nested closures. For example:

[Example: Nested DSL property configuration](#3tbugp1)

**build.gradle**

model {  
 person {  
 address {  
 city = "Melbourne"  
 }  
 }  
}

| **✨** | Currently, the nested closures do not define rules and are executed immediately. Please be aware that this behaviour will change in a future Gradle release. |
| --- | --- |

[All other subjects](#28h4qwu)

For all other types, the rule closure can use any of the methods defined by the type. There is no special DSL defined for these elements.

[Automatic type coercion](#nmf14n)

Scalar properties in managed types can be assigned CharSequence values (e.g. String, GString, etc.) and they will be converted to the actual property type for you. This works for all scalar types including `File`s, which will be resolved relative to the current project.

[Example: a DSL example showing type conversions](#37m2jsg)

**build.gradle**

enum Temperature {  
 TOO\_HOT,  
 TOO\_COLD,  
 JUST\_RIGHT  
}  
  
@Managed  
interface Item {  
 void setName(String n); String getName()  
  
 void setQuantity(int q); int getQuantity()  
  
 void setPrice(float p); float getPrice()  
  
 void setTemperature(Temperature t)  
 Temperature getTemperature()  
  
 void setDataFile(File f); File getDataFile()  
}  
  
class ItemRules extends RuleSource {  
 @Model  
 void item(Item item) {  
 def data = item.dataFile.text.trim()  
 def (name, quantity, price, temp) = data.split(',')  
 item.name = name  
 item.quantity = quantity  
 item.price = price  
 item.temperature = temp  
 }  
  
 @Defaults  
 void setDefaults(Item item) {  
 item.dataFile = 'data.csv'  
 }  
  
 @Mutate  
 void createDataTask(ModelMap<Task> tasks, Item item) {  
 tasks.create('showData') {  
 doLast {  
 println """  
Item '$item.name'  
 quantity: $item.quantity  
 price: $item.price  
 temperature: $item.temperature"""  
 }  
 }  
 }  
}  
  
apply plugin: ItemRules  
  
model {  
 item {  
 price = "${price \* (quantity < 10 ? 2 : 0.5)}"  
 }  
}

| **✨** | The code for this example can be found at samples/modelRules/modelDslCoercion in the ‘-all’ distribution of Gradle. |
| --- | --- |

In the above example, an Item is created and is initialized in setDefaults() by providing the path to the data file. In the item() method the resolved File is parsed to extract and set the data. In the DSL block at the end, the price is adjusted based on the quantity; if there are fewer than 10 remaining the price is doubled, otherwise it is reduced by 50%. The GString expression is a valid value since it resolves to a float value in string form.

Finally, in createDataTask() we add the showData task to display all of the configured values.

[Declaring input dependencies](#1mrcu09)

Rules declared in the DSL may *depend* on other model elements through the use of a special syntax, which is of the form:

$.«path-to-model-element»

Paths are a period separated list of identifiers. To directly depend on the firstName of the person, the following could be used:

$.person.firstName

[Example: a DSL rule using inputs](#46r0co2)

**build.gradle**

model {  
 tasks {  
 hello(Task) {  
 def p = $.person  
 doLast {  
 println "Hello $p.firstName $p.lastName!"  
 }  
 }  
 }  
}

| **✨** | The code for this example can be found at samples/modelRules/modelDsl in the ‘-all’ distribution of Gradle. |
| --- | --- |

In the above snippet, the $.person construct is an input reference. The construct returns the value of the model element at the specified path, as its default type (i.e. the type advertised by the [Model Report](#lnxbz9)). It may appear anywhere in the rule that an expression may normally appear. It is not limited to the right hand side of variable assignments.

The input element is guaranteed to be fully configured before the rule executes. That is, all of the rules that mutate the element are guaranteed to have been previously executed, leaving the target element in its final, immutable, state.

Most model elements enforce immutability when being used as inputs. Any attempt to mutate such an element will result in a runtime error. However, some legacy type objects do not currently implement such checks. Regardless, it is always invalid to attempt to mutate an input to a rule.

[Using ModelMap<T> as an input](#2lwamvv)

When you use a [ModelMap](http://docs.google.com/javadoc/org/gradle/model/ModelMap.html) as input, each item in the map is made available as a property.

[The model report](#lnxbz9)

The built-in [ModelReport](http://docs.google.com/dsl/org.gradle.api.reporting.model.ModelReport.html) task displays a hierarchical view of the elements in the model space. Each item prefixed with a + on the model report is a model element and the visual nesting of these elements correlates to the model path (e.g. tasks.help). The model report displays the following details about each model element:

Table 3. Model report - model element details

| **Detail** | **Description** |
| --- | --- |
| Type | This is the underlying type of the model element and is typically a fully qualified class name. |
| Value | Is conditionally displayed on the report when a model element can be represented as a string. |
| Creator | Every model element has a creator. A creator signifies the origin of the model element (i.e. what created the model element). |
| Rules | Is a listing of the rules, excluding the creator rule, which are executed for a given model element. The order in which the rules are displayed reflects the order in which they are executed. |

[Example: Model task output](#111kx3o)

**Output of gradle model**

> gradle model  
  
> Task :model  
  
------------------------------------------------------------  
Root project  
------------------------------------------------------------  
  
+ person  
 | Type: Person  
 | Creator: PersonRules#person(Person)  
 | Rules:  
 ⤷ person { ... } @ build.gradle line 97, column 3  
 ⤷ PersonRules#setFirstName(Person)  
 + age  
 | Type: int  
 | Value: 0  
 | Creator: PersonRules#person(Person)  
 + children  
 | Type: org.gradle.model.ModelSet<Person>  
 | Creator: PersonRules#person(Person)  
 + employed  
 | Type: boolean  
 | Value: false  
 | Creator: PersonRules#person(Person)  
 + father  
 | Type: Person  
 | Value: null  
 | Creator: PersonRules#person(Person)  
 + firstName  
 | Type: java.lang.String  
 | Value: John  
 | Creator: PersonRules#person(Person)  
 + homeDirectory  
 | Type: java.io.File  
 | Value: null  
 | Creator: PersonRules#person(Person)  
 + id  
 | Type: java.lang.Long  
 | Value: null  
 | Creator: PersonRules#person(Person)  
 + lastName  
 | Type: java.lang.String  
 | Value: Smith  
 | Creator: PersonRules#person(Person)  
 + maritalStatus  
 | Type: MaritalStatus  
 | Creator: PersonRules#person(Person)  
 + mother  
 | Type: Person  
 | Value: null  
 | Creator: PersonRules#person(Person)  
 + userGroups  
 | Type: java.util.List<java.lang.String>  
 | Value: null  
 | Creator: PersonRules#person(Person)  
+ tasks  
 | Type: org.gradle.model.ModelMap<org.gradle.api.Task>  
 | Creator: Project.<init>.tasks()  
 | Rules:  
 ⤷ PersonRules#createHelloTask(ModelMap<Task>, Person)  
 + buildEnvironment  
 | Type: org.gradle.api.tasks.diagnostics.BuildEnvironmentReportTask  
 | Value: task ':buildEnvironment'  
 | Creator: Project.<init>.tasks.buildEnvironment()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + components  
 | Type: org.gradle.api.reporting.components.ComponentReport  
 | Value: task ':components'  
 | Creator: Project.<init>.tasks.components()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + dependencies  
 | Type: org.gradle.api.tasks.diagnostics.DependencyReportTask  
 | Value: task ':dependencies'  
 | Creator: Project.<init>.tasks.dependencies()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + dependencyInsight  
 | Type: org.gradle.api.tasks.diagnostics.DependencyInsightReportTask  
 | Value: task ':dependencyInsight'  
 | Creator: Project.<init>.tasks.dependencyInsight()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + dependentComponents  
 | Type: org.gradle.api.reporting.dependents.DependentComponentsReport  
 | Value: task ':dependentComponents'  
 | Creator: Project.<init>.tasks.dependentComponents()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + hello  
 | Type: org.gradle.api.Task  
 | Value: task ':hello'  
 | Creator: PersonRules#createHelloTask(ModelMap<Task>, Person) > create(hello)  
 | Rules:  
 ⤷ copyToTaskContainer  
 + help  
 | Type: org.gradle.configuration.Help  
 | Value: task ':help'  
 | Creator: Project.<init>.tasks.help()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + init  
 | Type: org.gradle.buildinit.tasks.InitBuild  
 | Value: task ':init'  
 | Creator: Project.<init>.tasks.init()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + model  
 | Type: org.gradle.api.reporting.model.ModelReport  
 | Value: task ':model'  
 | Creator: Project.<init>.tasks.model()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + projects  
 | Type: org.gradle.api.tasks.diagnostics.ProjectReportTask  
 | Value: task ':projects'  
 | Creator: Project.<init>.tasks.projects()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + properties  
 | Type: org.gradle.api.tasks.diagnostics.PropertyReportTask  
 | Value: task ':properties'  
 | Creator: Project.<init>.tasks.properties()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + tasks  
 | Type: org.gradle.api.tasks.diagnostics.TaskReportTask  
 | Value: task ':tasks'  
 | Creator: Project.<init>.tasks.tasks()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + wrapper  
 | Type: org.gradle.api.tasks.wrapper.Wrapper  
 | Value: task ':wrapper'  
 | Creator: Project.<init>.tasks.wrapper()  
 | Rules:  
 ⤷ copyToTaskContainer

[Limitations and future direction](#35nkun2)

*The rule engine that was part of the Software Model will be deprecated.* Everything under the model block will be ported as extensions to the current model. Native users will no longer have a separate extension model compared to the rest of the Gradle community, and they will be able to make use of the new variant aware dependency management. For more information, see the [blog post](https://blog.gradle.org/state-and-future-of-the-gradle-software-model) on the state and future of the software model.

Docs

* [User Manual](http://docs.google.com/userguide/userguide.html)
* [DSL Reference](http://docs.google.com/dsl/)
* [Release Notes](http://docs.google.com/release-notes.html)
* [Javadoc](http://docs.google.com/javadoc/)

News

* [Blog](https://blog.gradle.org/)
* [Newsletter](https://newsletter.gradle.com/)
* [Twitter](https://twitter.com/gradle)

Products

* [Build Scans](https://gradle.com/build-scans)
* [Build Cache](https://gradle.com/build-cache)
* [Enterprise Docs](https://gradle.com/enterprise/resources)

Get Help

* [Forums](https://discuss.gradle.org/c/help-discuss)
* [GitHub](https://github.com/gradle/)
* [Training](https://gradle.org/training/)
* [Services](https://gradle.org/services/)

Subscribe for important Gradle updates and news

Subscribe

By entering your email, you agree to our [Terms](https://gradle.org/terms/) and [Privacy Policy](https://gradle.org/privacy/), including receipt of emails. You can unsubscribe at any time.

© [Gradle Inc.](https://gradle.com) 2018 All rights reserved.

[Careers](https://gradle.com/careers) | [Privacy](https://gradle.org/privacy) | [Terms of Service](https://gradle.org/terms) | [Contact](https://gradle.org/contact/)