Build environment and properties

https://docs.gradle.org/current/userguide/build_environment.html

Gradle provides multiple mechanisms for configuring behavior of Gradle itself and specific projects:

- **Command-line flags** such as --build-cache. These have precedence over properties and environment variables.
- **System properties** such as systemProp.http.proxyHost=somehost.org stored in a gradle.properties file.
- **Gradle properties** such as org.gradle.caching=true that are typically stored in a gradle.properties file in a project root directory or GRADLE_USER_HOME environment variable
- **Environment variables** such as GRADLE_OPTS sourced by the environment that executes Gradle.
- **Project properties** - Gradle will set a prop property on your project object via the -P command line option

Gradle properties

- command line, as set using the -P / --project-prop environment options.
- gradle.properties in GRADLE_USER_HOME directory.
- gradle.properties in project root directory.
- gradle.properties in Gradle installation directory.

gradle.properties - often used for storing certain settings like **JVM memory configuration** and **Java home location** in version control so that an entire team can work with a consistent environment.

Examples of some useful gradle properties:

- **org.gradle.caching**=(true,false) - When set to true, Gradle will reuse task outputs from any previous build, when possible, resulting is much faster builds. Learn more about using the build cache.
- **org.gradle.daemon**=(true,false) - When set to true the Gradle Daemon is used to run the build. Default is true.
- **org.gradle.debug**=(true,false) - When set to true, Gradle will run the build with remote debugging enabled, listening on port 5005. Note that this is the equivalent of adding -agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=5005 to the JVM command line and will suspend the virtual machine until a debugger is attached. Default is false.
- **org.gradle.parallel**=(true,false) - When configured, Gradle will fork up to org.gradle.workers.max JVMs to execute projects in parallel. To learn more about parallel task execution, see the section on Gradle build performance.
- **org.gradle.workers.max**=(max # of worker processes) - When configured, Gradle will use a maximum of the given number of workers. Default is number of CPU processors. See also performance command-line options.

System properties

Using the -D command-line option, you can pass a system property to the JVM which runs Gradle. The -D option of the gradle command has the same effect as the -D option of the java command.

You can also set system properties in gradle.properties files with the prefix systemProp: systemProp.gradle.wrapperUser=myuser

Available gradle system properties:

- gradle.wrapperUser=(myuser) - Specify user name to download Gradle distributions from servers using HTTP Basic Authentication. Learn more in Authenticated wrapper downloads.
- gradle.wrapperPassword=(mypassword) - Specify password for downloading a Gradle distribution using the Gradle wrapper.
- gradle.user.home=(path to directory) - Specify the Gradle user home directory.
- https.protocols - Specify the supported TLS versions in a comma separated format. For example: TLSv1.2,TLSv1.3.

I**n a multi project build, "systemProp." properties set in any project except the root will be ignored.** That is, only the root project's gradle.properties file will be checked for properties that begin with the "systemProp." prefix.

Project properties

You can add properties directly to your Project object via the -P command line option: -Pfoo=bar

Setting a project property via a system property: -Dorg.gradle.project.foo=bar

Setting a project property via an environment variable: ORG_GRADLE_PROJECT_foo=bar

You should check for existence of optional project properties before you access them using the Project.hasProperty(java.lang.String) method

Project

https://docs.gradle.org/current/dsl/org.gradle.api.Project.html

Multimodule Project

1. Put single settings.gradle to project root
   https://github.com/ivanwolkow/tax-calculator/blob/master/settings.gradle
   These settings are read by gradle before it starts to build any module.
   Add to these settings include 'rate-provider' (for each module)

2. Put build.gradle to project root
   https://github.com/ivanwolkow/tax-calculator/blob/master/build.gradle
   Add all logic applicable to every submodule (common dependencies, common plugins,repositories etc)

3. Create directory with submodule and put build.gradle in it
   https://github.com/ivanwolkow/tax-calculator/blob/master/tax-calculator-api/build.gradle
   Create a link between submodules by declaring it as a dependency. For example: implementation project(":rate-provider"), so the rate-provider source code gets included to configurations (source set) of a submodule declaring such a dependency.

Tasks

What exactly is the task going to do?

```
./gradlew build --dry-run

:tax-calculator-api:compileJava SKIPPED
:tax-calculator-api:processResources SKIPPED
:tax-calculator-api:classes SKIPPED
:tax-calculator-api:bootJarMainClassName SKIPPED
:tax-calculator-api:bootJar SKIPPED
:tax-calculator-api:jar SKIPPED
:tax-calculator-api:assemble SKIPPED
:tax-calculator-api:compileTestJava SKIPPED
:tax-calculator-api:processTestResources SKIPPED
:tax-calculator-api:testClasses SKIPPED
:tax-calculator-api:test SKIPPED
:tax-calculator-api:check SKIPPED
:tax-calculator-api:build SKIPPED
```

Plugins

How to apply plugins:

New way:

```
plugins {
    id "org.company.myplugin" version "1.3"
}
```

Old way:

```
apply plugin: 'java'
```

The **plugins** block is the newer method of applying plugins, and they must be available in the Gradle plugin repository. The **apply** approach is the older, yet more flexible method of adding a plugin to your build.

The new plugins method does not work in multi-project configurations (subprojects, allprojects), but will work on the build configuration for each child project.

Project scope

A project has 5 method 'scopes', which it searches for methods:

- The Project object itself.
- The build file. The project searches for a matching method declared in the build file.
- The extensions added to the project by the plugins. Each extension is available as a method which takes a closure or Action as a parameter.
- The convention methods added to the project by the plugins. A plugin can add properties and method to a project through the project's Convention object.
- The tasks of the project. A method is added for each task, using the name of the task as the method name and taking a single closure or Action parameter. The method calls the Task.configure(groovy.lang.Closure) method for the associated task with the provided closure. For example, if the project has a task called compile, then a method is added with the following signature: void compile(Closure configureClosure).
- The methods of the parent project, recursively up to the root project.
- A property of the project whose value is a closure. The closure is treated as a method and called with the provided parameters. The property is located as described above.

A project has 5 property 'scopes', which it searches for properties. You can access these properties by name in your build file, or by calling the project's Project.property(java.lang.String) method. The scopes are:

- The Project object itself. This scope includes any property getters and setters declared by the Project implementation class. For example, Project.getRootProject() is accessible as the rootProject property. The properties of this scope are readable or writable depending on the presence of the corresponding getter or setter method.
- The extra properties of the project. Each project maintains a map of extra properties, which can contain any arbitrary name -> value pair. Once defined, the properties of this scope are readable and writable. See extra properties for more details.

```
project.ext.prop1 = "foo"
task doStuff {
    ext.prop2 = "bar"
}
subprojects { ext.${prop3} = false }

//Reading extra properties is done through the "ext" or through the owning object:

ext.isSnapshot = version.endsWith("-SNAPSHOT")
if (isSnapshot) {
    // do snapshot stuff
}
```

- The extensions added to the project by the plugins. Each extension is available as a read-only property with the same name as the extension.
- The convention properties added to the project by the plugins. A plugin can add properties and methods to a project through the project's Convention object. The properties of this scope may be readable or writable, depending on the convention objects.
- The tasks of the project. A task is accessible by using its name as a property name. The properties of this scope are read-only. For example, a task called *compile* is accessible as the compile property.
- The extra properties and convention properties are inherited from the project's parent, recursively up to the root project. The properties of this scope are read-only.

When reading a property, the project searches the above scopes in order, and returns the value from the first scope it finds the property in.