 Extension points

Jenkins provides a [set of extension points](https://wiki.jenkins-ci.org/display/JENKINS/Extension+points), which are interfaces or abstract classes that model a part of a Jenkins functionality. Plugins can extend those extention points in order to contribute with their implementation. They can also define their own extension points.

The incredible huge amount of open source plugins available in the [Jenkins GitHub account](https://github.com/jenkinsci) provide many exemples that can get you started to create your own plugin.

Those interfaces define contracts of what need to be implemented, and Jenkins allows plugins to contribute those implementations. In general, all you need to do to register an implementation is to mark it with @Extension. Creating a new extension point is quite easy too.

ust as Jenkins core defines a set of extension points, your plugin can define new extension points so that other plugins can contribute implementations. This page shows how to do that.

Extension points may be defined in either of two ways:

1. Singleton pattern
2. Describable/Descriptor pattern (builds on singleton pattern)

# Singleton Pattern

First, you define your contract in terms of an abstract class. (You can do this in interfaces, too, but then you'll have hard time adding new methods in later versions of your plugin.) This class should implement the marker [hudson.ExtensionPoint](http://javadoc.jenkins-ci.org/byShortName/hudson.ExtensionPoint) interface, to designate that it is an extension point.

For convenience you can also define a static method conventionally named all. The returned ExtensionList allows you discover all the implementations at runtime.

The following code shows this in a concrete example:

/\*\*

\* Extension point that defines different kinds of animals

\*/

public abstract class Animal implements ExtensionPoint {

...

/\*\*

\* All registered {@link Animal}s.

\*/

public static ExtensionList<Animal> all() {

return Jenkins.getActiveInstance().getExtensionList(Animal.class); // getActiveInstance() starting with Jenkins 1.590, else getInstance()

}

}

In addition to these basic ingredients, your extension point can implement additional interfaces, extend from another base class, define all sorts of methods, etc. Because the extension implementations will be singletons, usually the default no-argument constructor suffices.

Also, this system is only necessary when you need to discover the implementations at runtime. So if your extension point consists of a group of several classes and interfaces, normally only the entry point needs to follow this convention. Such an example can be seen in [hudson.scm.ChangeLogParser](http://javadoc.jenkins-ci.org/byShortName/hudson.scm.ChangeLogParser) in the core — ChangeLogParser implementations are always created by [hudson.scm.SCM](http://javadoc.jenkins-ci.org/byShortName/hudson.scm.SCM) implementations, so ChangeLogParser is not an extension point.

### Enumerating implementations at runtime

The following code shows how you can list up all the Animal implementations contributed by other plugins.

for (Animal a : Animal.all()) {

System.out.println(a);

}

There are other convenience methods to find a particular instance, and so on. See [hudson.ExtensionList](http://javadoc.jenkins-ci.org/byShortName/hudson.ExtensionList) for more details.

### Implementing extension points

Implementing an extension point defined in a plugin is no different from implementing an extension point defined in the core. See [hudson.Extension](http://javadoc.jenkins-ci.org/byShortName/hudson.Extension) for more details.

@Extension

public class Lion extends Animal { ... }

# Describable/Descriptor pattern

If you are going to define a new extension point that follows the [hudson.model.Describable](http://javadoc.jenkins-ci.org/byShortName/hudson.model.Describable)/[hudson.model.Descriptor](http://javadoc.jenkins-ci.org/byShortName/hudson.model.Descriptor) pattern, the convention is bit different. This pattern is used when users should be able to configure zero or more instances of some things you define.

In this case the singleton is a Descriptor: a description of the kind of thing a user can create and configure. There is a matching Describable class which represents the actual things being created and configured. Confusingly, the convention is to place the ExtensionPoint marker on the Describable class, even though @Extension will be put on implementations of the Descriptor.

For this, first you define your Describable subtype and Descriptor subtype.

public abstract class Food extends AbstractDescribable<Food> implements ExtensionPoint {

...

@Override public FoodDescriptor getDescriptor() {

return (FoodDescriptor) super.getDescriptor();

}

}

public abstract class FoodDescriptor extends Descriptor<Food> {

protected FoodDescriptor() {

}

// optional constructor specifying class (the default one usually suffices however):

protected FoodDescriptor(Class<? extends Food> clazz) {

super(clazz);

}

}

An extension for Food would look like this:

public class MyFoodImpl extends Food {

private final boolean tasty;

@DataBoundConstructor public MyFoodImpl(boolean tasty) {

this.tasty = tasty;

}

public boolean isTasty() {return tasty;}

...

@Extension

    public static class DescriptorImpl extends FoodDescriptor {

        @Override

        public String getDisplayName() {

            return "MyFood";

        }

    }

}

Typically you will also need a src/main/resources/…/MyFoodImpl/config.jelly providing its configuration form. The exact Jelly views needed for a describable vary according to how the extension point is used.