OSGi

OSGi — Open Service Gateway Initiative

- OSGi is a Java framework for developing and deploying modular software programs and libraries.
    i. Specification for modular components called bundles, commonly referred to as plug-ins. The specification defines an infrastructure for a bundle's life cycle and determines how bundles will interact.
    ii. The second part of OSGi is a Java Virtual Machine (JVM)-level service registry that bundles can use to publish, discover and bind to services in a service-oriented architecture (SOA).
- OSGi provides a way that bundles can communicate with each other without being coupled together (interfaces, etc)

Summary

Note: *Bundle* and *plug-in* can be used interchangeably; *plug-in* is preferred by Eclipse, *bundle* is preferred by OSGi.

Most applications consist of several different parts, called modules or components, which interact with each other through an Application Programming Interface, or API. The API is considered to be the classes/methods which are accessible to other components. A component also has a set of classes/methods that are considered internal. Using an API from another component creates a dependency on the component, as it now requires the other component to be present in order to work correctly. Currently, Java doesn't provide a structured way to describe the various dependencies of a component; rather, every public class can be called from another component.

OSGi is a specification framework that defines a component and service model for creating loosely coupled modular bundles. Each bundle can define its API through a set of exported packages and also specify its own required dependencies.

Bundles

A bundle can define exports or dependencies on other bundles in its manifest file. OSGi prevents access to classes without a defined dependency (except for packages from the Java

#	Status	Description

runtime). Dependencies can be defined as either bundle or package dependencies. Bundle dependencies enable accessing all exported packages from the bundle while package dependencies enable accessing a specific package without worrying what bundle exports it (enabling a later change). The OSGi runtime **ensures that all dependencies are present before starting a plugin**, activating them if necessary.

Bundle Lifecycle

#	Status	Description
1	Installed	Bundle has been installed
2	Resolved	All bundle dependencies have been resolved
3	Starting	Bundle is starting
4	Active	Bundle has started and is running
5	Stopping	Bundle is stopping
6	Resolved	*N/A*
7	Uninstalled	Bundle has been uninstalled

NOTE: Bundles must all contain an `Activator` class that is responsible for managing the bundle's life cycle, including a `start()` and `stop()` set of methods. These methods are responsible for handling any necessary preparation or clean up during the beginning and end of the bundle's life cycle.

Declarative Services

A *service* in OSGi is defined by a standard Java class or interface, composed of the class/interface for which you want to provide a service and the implementation class for the service interface. This enables switching out the service implementation but still using the existing service interface. Services can be dynamically stopped/started, so bundles that use these services must be able to handle this behaviour, typically done by the `ServiceListener` handlers.

It is good practice to define a service with a bundle that contains only the service interface definition, with another bundle providing the service implementation. This way, the implementation of the service can always be changed at a later time. Since dependent bundles only communicate with the service through the service interface, updating the service implementation should not break existing dependent bundles. For example, if an existing implementation used a SQL database to persist data but the developers wanted to switch to

another database vendor, the only requirement would be that the new implementation correctly supports the existing service interface.

```
This is a code sample with no background color?


   //Sample Java code
   public void sampleMethod(String input) {
       if (input.equals("test")) {
           return true;
       }
       return false;
   }
```

Declarative Services remove the need for registering and consuming services programmatically. Instead, *Declarative Services* can be declared by a *Component Definition* file, which is either manually configured or rendered automatically by Eclipse annotations (which will also update the MANIFEST).

List of Lifecycle Methods (duplicate)

- Installed
    - Bundle has been installed
- Resolved
    - All bundle dependencies have been resolved
        - This can cause problems?
- Starting
- Active
    - Bundle has been started and is running

NOTE: A *Service Component Runtime* bundle **must** be installed and activated on the device if using Declarative Services. However, this bundle will likely already be installed as part of the environment.