OSGi Service Platform

OSGi Service Platform I

Wütherich et al. (2008)

- OSGi defines a **dynamic component model** for Java, ie. components can be installed, updated and uninstalled without stopping or restarting the platform.
- Components provided by OSGi are called **bundles**.
  A bundle
  - contains an additional file with descriptive information, e.g. about provided and required interfaces.
  - can implement a **service**. Services are registered at a central **Service Registry** where other bundles can request it.
  - can be in different states (e.g. installed, active). The bundle lifecycle can be managed by the **OSGi Framework API**.

OSGi Service Platform II

Wütherich et al. (2008)

OSGi

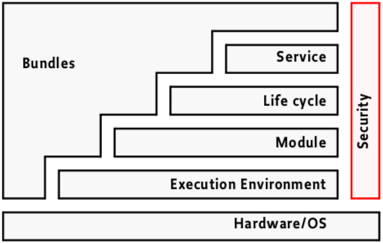
- used to stand for **O**pen **S**ervice **G**ateway **i**nitiative.
- is a **standard** defined by the OSGi Alliance (`http://www.osgi.org`).
- is used in applications ranging from mobile phones to the Eclipse IDE[1].
- is realized by open source (e.g. Eclipse Equinox) and commercial implementations.
- consists of two parts: **OSGi Framework** and **OSGi Standard Services**

[1] **I**ntegrated **D**evelopment **E**nvironment

OSGi Framework

OSGi Framework (OSGi Alliance (2010b)) I

- The OSGi Framework implements a container for bundles.
- The functionality of the framework is divided into the following layers:



Execution Environment Defines the Java environment that is needed to execute the OSGi Framework.

Module Defines a component model for Java.

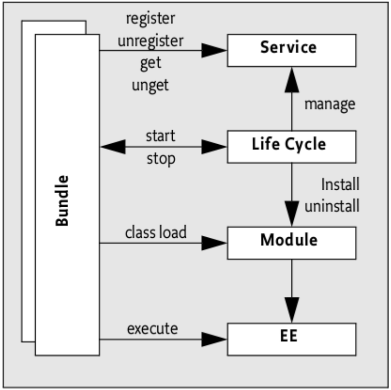
Lifecycle Defines the states of a bundle.

Service Defines a service model.

Security Defines security relevant aspects.

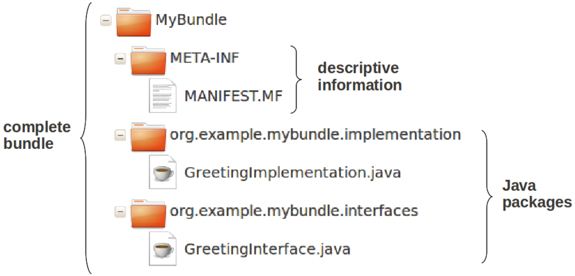
OSGi Framework (OSGi Alliance (2010b)) II

Interactions between layers:



Bundles

- A **Bundle**
  - represents a component in the OSGi Framework.
  - consists of one or more Java packages.
  - is deployed as a **J**ava **AR**chive (JAR) with additional descriptive information.



complete bundle

MyBundle

META-INF

MANIFEST.MF

descriptive information

org.example.mybundle.implementation

GreetingImplementation.java

org.example.mybundle.interfaces

GreetingInterface.java

Java packages

- The descriptive information is stored within the **bundle manifest** `MANIFEST.MF`.

Example: Bundle Manifest

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: My first bundle
Bundle-SymbolicName: org.example.mybundle
Bundle-Version: 1.0.0
```

Bundle Manifest Header	**Optional**	**Description**
`Bundle-ManifestVersion`	yes	Number corresponds to version of the OSGi specification (2 for current version).
`Bundle-Name`	yes	Defines a readable name for the bundle.
`Bundle-SymbolicName`	no	Bundle symbolic name and version must identify a unique bundle.
`Bundle-Version`	yes	Specifies the version of the bundle (default value is `0.0.0`).

Export and Import of Packages I

- By default, the classes contained in a bundle are **not** visible to classes from other bundles.
- In order to use classes of one bundle in another bundle, they must be **exported** and **imported**.
- In OSGi, only packages (and thereby the contained classes) may be exported and imported.

Export and Import of Packages II

- In order to offer a provided interface, a bundle must export the package containing the interface. Therefore the following line has to be added to the corresponding `MANIFEST.MF`:

```
Export-Package: org.example.mypackage,
                org.example.anotherpackage
```

- A bundle that requires these interfaces has to import the packages. This is done by adding the following line to the `MANIFEST.MF` of that bundle:

```
Import-Package: org.example.mypackage,
                org.example.anotherpackage
```

- The OSGi Framework resolves these dependencies by matching the imports and exports automatically as soon as both bundles are installed.

Export and Import of Packages III

- An exported package can be supplied with a version:

  ```
  Export-Package:
    org.example.mypackage;version="1.0.0"
  ```

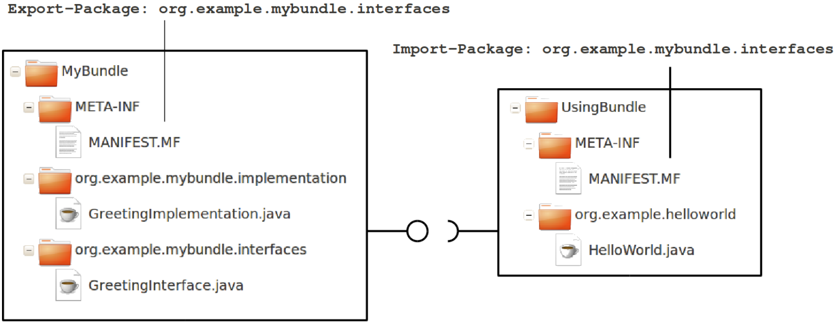
  (The default value is `0.0.0`.)

- For an imported package, a version range can be specified:

  ```
  Import-Package:
    org.example.mypackage;version="[1.1.0,1.5.0)"
  ```

  (i.e. `org.example.mypackage` can only be imported if its version number is greater than or equal to `1.1.0` and less than `1.5.0`)

Example: Bundles with provided and required interfaces I


Export-Package: org.example.mybundle.interfaces
MyBundle
META-INF
MANIFEST.MF
org.example.mybundle.implementation
GreetingImplementation.java
org.example.mybundle.interfaces
GreetingInterface.java
Import-Package: org.example.mybundle.interfaces
UsingBundle
META-INF
MANIFEST.MF
org.example.helloworld
HelloWorld.java

Example: Bundles with provided and required interfaces II

Bundle "MyBundle"

Package containing the interface:

```
package org.example.mybundle.interfaces;
public interface GreetingInterface {
  public void sayHello();
}
```

Package containing the implementation:

```
package org.example.mybundle.implementation;
import org.example.mybundle.interfaces.GreetingInterface;
public class GreetingImplementation implements
  GreetingInterface {
  public void sayHello() {
    System.out.println("Hello!");
  }
}
```

Example: Bundles with provided and required interfaces III

MANIFEST.MF of bundle "MyBundle":

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Bundle with provided interface
Bundle-SymbolicName: org.example.mybundle
Bundle-Version: 1.0.0
Export-Package: org.example.mybundle.interfaces;
                version="1.0.0"
```

Example: Bundles with provided and required interfaces IV

Bundle "UsingBundle”

Package using the interface:

```
package org.example.helloworld;
import org.example.mybundle.interfaces.GreetingInterface;
public class HelloWorld {
  public HelloWorld(GreetingInterface gi) {
    gi.sayHello();
  }
}
```

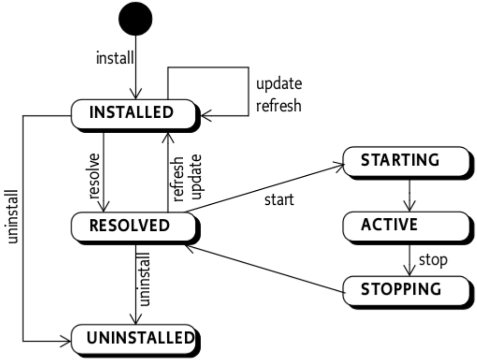
Example: Bundles with provided and required interfaces V

MANIFEST.MF of bundle "UsingBundle"

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Bundle with required interface
Bundle-SymbolicName: org.example.usingbundle
Bundle-Version: 1.0.0
Import-Package: org.example.mybundle.interfaces;
                version="[1.0.0,1.5.0)"
```

Bundle Lifecycle

- A bundle that is installed within the OSGi Framework can be in the states **INSTALLED**, **RESOLVED**, **STARTING**, **ACTIVE**, **STOPPING** or **UNISTALLED**.



- The lifecycle of a bundle can be managed by the API of the OSGi Framework.

Management of the bundle lifecycle I

BundleActivator

One can specify actions a bundle should perfom when it is started and stopped. To this end the following interface `BundleActivator` is to be implemented:

```
public interface BundleActivator{
   public void start(BundleContext context)
      throws Exception;
   public void stop(BundleContext context)
      throws Exception;
}
```

Management of the bundle lifecycle II

BundleActivator

The implementing class (only one per bundle allowed) must have a public, no-argument constructor.

Activator class of bundle "SomeBundle":

```
package org.example;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
public class HelloWorldActivator implements
   BundleActivator {
     public HelloWorldActivator() {}
     public void start(BundleContext context)
        throws Exception {
          System.out.println("Hello OSGi-World!");
     }
     public void stop(BundleContext context)
        throws Exception {
          System.out.println("Goodbye OSGi-World!");
     }
}
```

Management of the bundle lifecycle III

BundleActivator

MANIFEST.MF of bundle ”SomeBundle”:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Bundle with bundle activator
Bundle-SymbolicName: org.example
Bundle-Version: 1.0.0
Import-Package: org.osgi.framework;version="1.5.0"
Bundle-Activator: org.example.HelloWorldActivator
```

Management of the bundle lifecycle IV

BundleActivator

A bundle can use a BundleActivator to store the given BundleContext:

```
...
public class HelloWorldActivator implements
  BundleActivator {
    private BundleContext bundleContext;
    public void start(BundleContext context)
      throws Exception {
        this.bundleContext = context;
    }
...
}
```

Management of the bundle lifecycle V

BundleContext

- The `BundleContext` object represents the interface between all bundles and the OSGi Framework.
- This object provides methods to
  - install a new bundle:

```
public Bundle installBundle(String location)
  throws BundleException
```

  - access all installed bundles:

```
public Bundle[] getBundles()
```

  - (de-)register listeners on bundles.
  - (de-)register services a bundle provides.
  - request services of other bundles.

Management of the bundle lifecycle VI

Bundle

- Every bundle that is installed within the OSGi framework is represented by an object of type `Bundle`.
- This object provides methods to manipulate the lifecycle of the corresponding bundle:

```
public void start() throws BundleException
public void stop() throws BundleException
public void update() throws BundleException
public void uninstall() throws BundleException
```

Management of the bundle lifecycle VII

Bundle

Example:

```
...
private BundleContext bundleContext;

// called by start method of activator class
public void setBundleContext(BundleContext context) {
  this.bundleContext = context;
}

public void installAndStartABundle(String location) {
  try {
    Bundle bundle = bundleContext.installBundle(location);
    bundle.start();
    } catch (BundleException e) {
      e.printStackTrace();
    }
}
 ...
```

Management of the bundle lifecycle VIII

BundleListener

- To be able to react to a changed bundle state the interface BundleListener has to be implemented:

  ```
  public interface BundleListener
    extends EventListener{
      public void bundleChanged(BundleEvent event);
  }
  ```

- The BundleContext object provides a methods to (de-)register a BundleListener:

  ```
  public void addBundleListener(BundleListener
    listener)
  public void removeBundleListener(BundleListener
    listener)
  ```

- When the state of any bundle changes, the OSGi framework calls the method bundleChanged.

Management of the bundle lifecycle IX

`BundleListener`

Implementation of a `BundleListener`:

```
public class ReportChange implements BundleListener {
  public void bundleChanged(BundleEvent event) {
    System.out.println(event.getBundle() + "changed its state");
  }
}
```

Registration of a `BundleListener`:

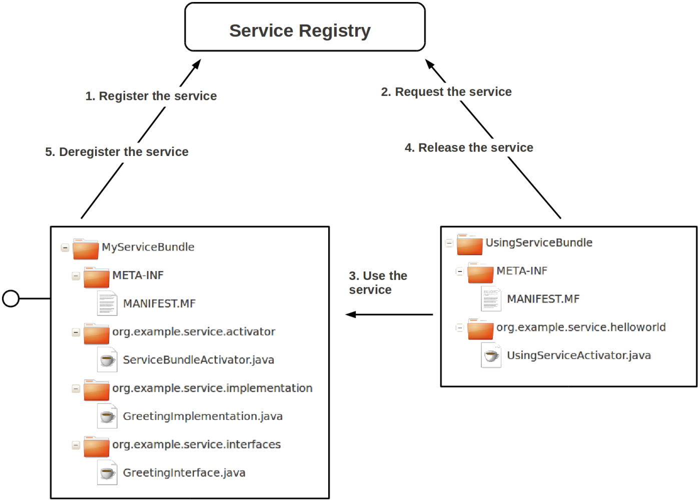
```
...
public class HelloWorldActivator implements
  BundleActivator {
    private BundleContext bundleContext;
    public void start(BundleContext context)
      throws Exception {
        this.bundleContext = context;
        ReportChange reportChange = new ReportChange();
        context.addBundleListener(reportChange);
    }
...
}
```

Services I

- A **service** is a simple Java object contained in a bundle.
- Services are registered at a central **Service Registry** where other bundles can request it.
- The **Service Registry** is part of the OSGi Framework.

Services II

To work with a service, the following steps are necessary:



- MyServiceBundle
  - META-INF
    - MANIFEST.MF
  - org.example.service.activator
    - ServiceBundleActivator.java
  - org.example.service.implementation
    - GreetingImplementation.java
  - org.example.service.interfaces
    - GreetingInterface.java

- UsingServiceBundle
  - META-INF
    - MANIFEST.MF
  - org.example.service.helloworld
    - UsingServiceActivator.java

Register the service I

- The bundle implementing the service must create the service and register this service object via the `BundleContext` at the Service Registry:

```
public ServiceRegistration registerService(String
  name, Object service, Dictionary properties)
```

- The service object is registered under a specific name (usually the name of the interface that the service implements).
- `Dictionary` is a Java class that maps keys to values. It can be used to describe properties of the service.

Register the service II

Bundle "MyServiceBundle" registers the service

Activator class of bundle "MyServiceBundle":

```
package org.example.service.activator;
...
public class ServiceBundleActivator implements BundleActivator {
  private ServiceRegistration registration;
  public void start(BundleContext context) throws Exception {
    GreetingImplementation gi = new GreetingImplementation();
    registration = context.registerService
      (GreetingInterface.class.getName(), gi, null);
  }
  public void stop(BundleContext context) throws Exception {...}
}
```

Request, use and release the service I

- The BundleContext provides methods to request and release a service:
  - Another bundle can request the registered service by its specific name:

    ```
    public ServiceReference getServiceReference
        (String name)
    ```

  - By means of the returned ServiceReference, a reference to the service object can be requested:

    ```
    public Object getService
        (ServiceReference reference)
    ```

Request, use and release the service II

  - To enable the OSGi Framework to manage which bundles are using which services, a service has to be released when it is not used any more:

    ```
    public boolean ungetService
      (ServiceReference reference)
    ```

    The returned `boolean` value is `false` if the bundle never used the service or the service was already deregistered.

- A service object can be used by different bundles at the same time.

Request, use and release the service III

”UsingServiceBundle” requests, uses and releases the service of bundle ”MyServiceBundle”

Activator class of bundle ”UsingServiceBundle”:

```
package org.example.service.helloworld;
...
public class UsingServiceActivator implements BundleActivator {
  public void start(BundleContext context) throws Exception {
    ServiceReference reference = context.getServiceReference
      (GreetingInterface.class.getName());
    GreetingInterface gi =
      (GreetingInterface)context.getService(reference);
    gi.sayHello();
    context.ungetService(reference);
  }
  public void stop(BundleContext context) throws Exception {...}
}
```

Deregister the service I

- When the service should not be available any more, the service can be deregistered by the bundle that registered the service.
- This is done by the `ServiceRegistration` object that the method `registerService` returned:
  `public void unregister()`

Deregister the service II

Bundle "MyServiceBundle" deregisters the service

Activator class of bundle "MyServiceBundle":

```
package org.example.service.activator;
...
public class ServiceBundleActivator implements BundleActivator {
  ServiceRegistration registration;
  public void start(BundleContext context)
    throws Exception {...}
  public void stop(BundleContext context) throws Exception {
    registration.unregister();
  }
}
```

Dynamic services I

- Services are **dynamic**, i.e. they can be registered or deregistered at any time.
- The interface `ServiceTrackerCustomizer` acts as a service listener:

```
public Object addingService
  (ServiceReference reference)
public void modifiedService
  (ServiceReference reference, Object service)
public void removedService
  (ServiceReference reference, Object service)
```

Dynamic services II

Implementation of a `ServiceTrackerCustomizer`:

```
public class ReportServiceChange
    implements ServiceTrackerCustomizer {
  private BundleContext context;
  public ReportServiceChange(BundleContext context) {
    this.bundleContext = context;
  }
  public Object addingService(ServiceReference reference) {
    System.out.println(reference.getBundle.getSymbolicName()
      + "was registered"); } }
    return context.getService(reference);
  }
  public void modifiedService(ServiceReference reference,
    Object service) {...}
  public void removedService(ServiceReference reference,
    Object service) {...}
}
```

Dynamic services III

- To register a service listener, a bundle must create a `ServiceTracker`:

```
public ServiceTracker(BundleContext context,
  String name,
  ServiceTrackerCustomizer customizer)
```

- The constructor takes the name of the service that should be monitored for changes.
- The `ServiceTracker` object calls the corresponding methods of the `ServiceTrackerCustomizer` when the service is registered, deregistered or one of its properties changes.

Dynamic services IV

Registration of a `ServiceTrackerCustomizer`:

```
...
private ServiceTracker tracker;
public void start(BundleContext context) throws Exception {
  ReportServiceChange reportServiceChange =
    new ReportServiceChange(context);
  tracker = new ServiceTracker(context,
    GreetingInterface.class.getName(), reportServiceChange);
  tracker.open(); // to start the ServiceTracker
  }
...
```

OSGi Standard Services

OSGi Standard Services

- **OSGi Standard Services** (OSGi Alliance (2010a)):
  - are based on the OSGi Framework
  - offer an API for different recurring problems
- Over 20 OSGi Standard Services are defined:
  - **Declarative Services**
  - Event Admin Service
  - Http Service
  - . . .

Declarative Services I

- In large applications, the service model of the OSGi Framework has some drawbacks:
  - **Start-up time**:
    Instantiation and registration of many services takes too much time.
  - **Memory usage**:
    For every registered service, all associated classes and objects are loaded in memory.
  - **Complexity**:
    Because services can be registered and deregistered at any time, the programming model is complex.

Declarative Services II

- **Declarative Services** address these problems by introducing **service components** which
  - are not activated until the service provided by the service component is requested for the first time.
  - are not activated until all services required by the service component are available.

Service Component

A **service component** is defined in a bundle and consists of

- a **component class**:
  - simple Java class
  - must have a public, no-argument constructor
  - can implement the methods `activate(ComponentContext)` and `deactivate(ComponentContext)` to specify actions that should be performed when the component is (de-)activated
- a **component description**
  - description of the component as an XML document
  - additional line in `MANIFEST.MF`: `Service-Component: OSGI-INF/component-description.xml`

The **Service Component Runtime** creates service components and manages their lifecycle.

Example: A simple service component

Component class:

```
package org.example.simplecomponent;
import org.osgi.service.component.ComponentContext;
public class SimpleComponent {
   protected void activate(ComponentContext context) {
      System.out.println("activate");
   }
   protected void deactivate(ComponentContext context) {
      System.out.println("deactivate");
   }
}
```

Component description:

```
<?xml version="1.0"?>
<component name="simpleComponent">
  <implementation class=
    "org.example.simplecomponent.SimpleComponent"/>
</component>
```

Delayed Component I

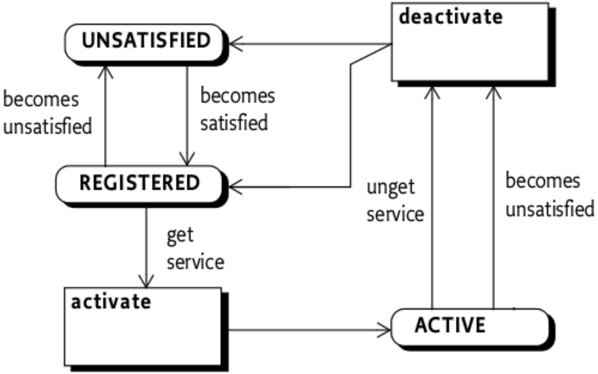
- An instance of a service component can be registered as an OSGi service.
- This is done by adding the XML element `service` to the component description:

```
<service>
<provide interface="...">
</service>
```

- `<provide interface="...">` is used to specify the name the service should be registered under.
- A service component that provides a service is not activated until the service is requested for the first time.
- Such a service component is called a **delayed component**.

Delayed Component II

Lifecycle of a delayed component:



A service component is satisfied as soon as its dependencies can be resolved.

Example: A service component as a service I

Service Interface:

```
package org.example.simplecomponent;
public class SimpleService {
   public void sayHello();
}
```

Component class:

```
package org.example.simplecomponent;
import org.osgi.service.component.ComponentContext;
public class SimpleComponent implements SimpleService {
   public void sayHello() {
      System.out.println("Hello!");
   }
}
```

Example: A service component as a service II

Component description:

```
<?xml version="1.0"?>
<component name="simpleComponent">
  <implementation class=
    "org.example.simplecomponent.SimpleComponent"/>
  <service>
    <provide interface=
      "org.example.simplecomponent.SimpleService"/>
  </service>
</component>
```

Immediate Component I

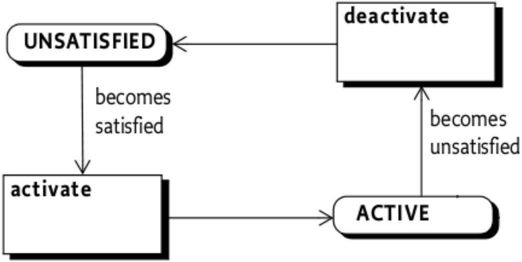
- A service component can use services registered by other bundles or service components.
- This is done by adding the XML element `reference` to the component description:

```
<reference
  name="..."
  interface="..."
  bind="..."
  unbind="..."
/>
```

- `name`: The local name of the reference.
- `interface`: The name the service is registered under.
- `bind`: The name of the method that is used to assign the service to the component.
- `unbind`: The name of the method that is used to remove the service from the component.

Immediate Component II

- A service component that uses services is activated as soon as all requested services are available.
- Such a service component is called an **immediate component**.
- Lifecycle of an immediate component:



Example: A service component uses a service I

Component class:

```
package org.example.hellocomponent;
import org.osgi.service.component.ComponentContext;
public class HelloComponent {
  private SimpleService service;
  protected void setService(SimpleService service) {
    this.service = service;
  }
  protected void unsetService(SimpleService service) {
    this.service = null;
  }
  protected void activate(ComponentContext componentContext) {
    sayHello();
  }
}
```

Example: A service component uses a service II

Component description:

```
<?xml version="1.0"?>
<component name="helloComponent">
  <implementation class=
    "org.example.hellocomponent.HelloComponent"/>
  <reference
    name="SimpleService"
    interface="org.example.simplecomponent.SimpleComponent"
    bind="setService"
    unbind="unsetService"
  />
</component>
```

Advantages of service components

- **Delayed activation of services**:
  Services that are provided by service components will be registered at the Service Registry when the implementing bundle is started. But no service instance is created until the service is requested for the first time. This reduces **start-up time** and **memory usage**.
- **Resolution of service dependencies**:
  The Service Component Runtime resolves all service dependencies. It instantiates and activates a service component that uses services not until all necessary services are available. Therefore, no service listeners have to be implemented. This reduces **complexity**.

What have we learned? I

- OSGi defines a dynamic component model for Java.
- In OSGi, components are called bundles. Bundles
  - consist of Java packages and an additional file with descriptive information (e.g. about exports and imports).
  - have a lifecycle that can be controlled by the OSGi Framework API.
  - can implement services which are registered at the Service Registry where other bundles can request them
- OSGi Standard Services offer an API for different recurring problems, like Declarative Services which reduce start-up time, memory usage and complexity when working with services.

What have we learned? II

	Description	**Management**	**Listener**
Bundle	OSGi component	`BundleActivator` (`start` and `stop` methods), `BundleContext`, `Bundle`	`BundleListener`
Service	Java object contained in a bundle	Service Registry	`ServiceTracker-Customizer`, `ServiceTracker`
Service Component	Java object and component description contained in a bundle	Service Component Runtime, `activate` and `deactivate` methods	
Delayed Component	Service component which provides a service	Service Component Runtime, `activate` and `deactivate` methods	
Immediate Component	Service component which uses services	Service Component Runtime, `activate` and `deactivate` methods	