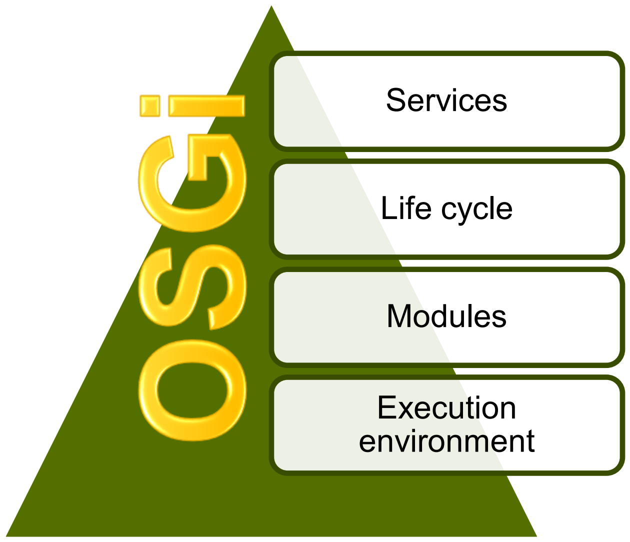
OSGi

Introduction to the framework

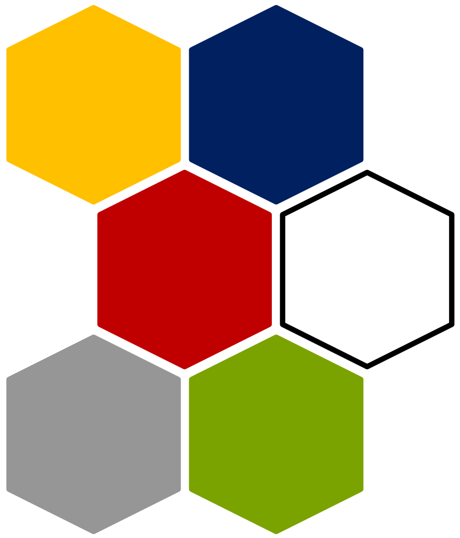


Agenda

- Modularity and versioning
- Framework characteristics
- Architecture overview
- OSGi fundamentals
- A few specifications
- High-level service support
- Common service patterns
- Practical concerns

About modularity

- Bane of non-modular code
- Versioning hell
- Core principles



What modularity means

- **Module** ≈ a deliverable code artifact with well-defined boundaries

- Additional level of encapsulation
  - Hiding details that are not explicitly made public
- Well-defined interfaces
  - Explicit definition of the public parts of a module
- Explicit dependencies
  - Requirements of a module are expressed explicitly and checked
- Versioning
  - Support for evolving modules with considerations to compatibility

What modularity brings

- Reducing system complexity by hiding the complexity of each module behind the module’s boundaries
- Reducing the possibility of misuse and abuse private code, thus improving reliability, stability and security
- More reliable composition of the system
- Unlocking performance gains (when implemented on the runtime level)

Java and modularity

- Modularity is difficult… very difficult
  - Taking modularity seriously comes at costs that seem often unacceptable until everything falls apart
- Java lacked any module system originally
  - An application was just a bunch of classes and other resources delivered usually as `.jar` files
  - Dependency management for building an application was handled by third-party tools
  - JVM knew nothing about the dependencies, let alone about conflicting or incompatible dependencies
- Is a lack of a module system a problem?

What is wrong with *classpath*?

- Consider following command:

```
java -cp app-1.0.0.jar:foo-1.0.0.jar:foo-2.0.0.jar
```

  What could happen?

- And what now:

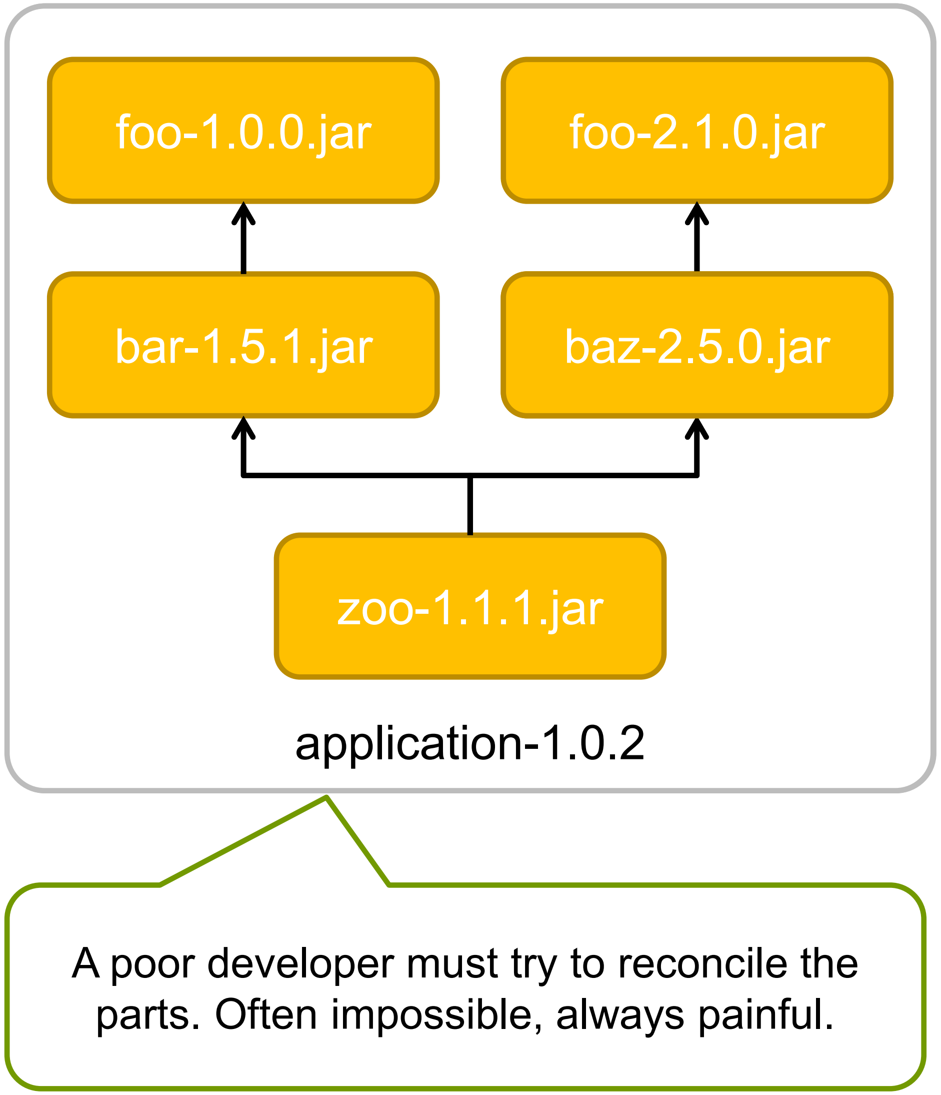
```
java -cp app-1.0.0.jar:foo-2.0.0.jar:foo-1.0.0.jar
```

- And what if…
  …there are tens or hundreds of `.jar` files?
  …some `.jar` files are embedded?
  …the application should be extensible?

Classpath hell: an often problem

- An application consists of multiple libraries usually
- A library often depends on other libraries
- A library evolves in time, thus multiple versions exist and could be used
- What if an application needs libraries that depend on different versions of the same library?

➢ This case is lost without a module system (or at least home-grown dark magic)


foo-1.0.0.jar
foo-2.1.0.jar
bar-1.5.1.jar
baz-2.5.0.jar
zoo-1.1.1.jar
application-1.0.2
A poor developer must try to reconcile the parts. Often impossible, always painful.

Lessons learned

- Probably all newly appeared languages consider modularity and version management (in a way at least)

- After all, Java got a module system as well
  - The delay caused a lot of pain and the ecosystem still has not recovered fully
  - But we’ll talk about it later, so that we could compare it with OSGi’s approach

Conclusions I

- ➢ Dependencies form a graph capturing module relations

- ▪ Mere artifact list does not capture the relations – it provides just some enumeration of the graph nodes
- ▪ The non-linear nature of the graph allows grouping dependencies in non-conflicting class spaces (with a few assumptions)

Conclusions II

- ➢ Capturing the dependency structure needs retaining the information for module system (module metadata)

- ▪ The usual and (relatively) comfortable way: explicit list of capabilities required from other modules and provided to them – common case is *imports* and *exports*

Conclusions III

- ➢ It must be possible to deduce version compatibility easily (if compatibility is solved at all… and it should be)

- ▪ Version information must be incorporated in modules and considered on their use
- ▪ Compatibility of various versions must be computable from the version information
- ▪ Probably the only reliable and “simple” way: semantic versioning

Semantic versioning in a nutshell

➢ Version information is stored as a version number (a tuple actually)

- Version number: *major.minor.micro*
- Version number has semantics (it is not arbitrary and must obey some rules)

➥ Compatibility is obvious just from comparison of version numbers

Version component	**Meaning**
micro	Indicates implementation changes that are not publicly visible (e.g., a bug fix or a security update)
minor	Indicates compatible changes of previously published parts (e.g., adding methods, classes/interfaces or packages)
major	Indicates incompatible changes of previously published parts (e.g., renaming or removing anything published earlier)

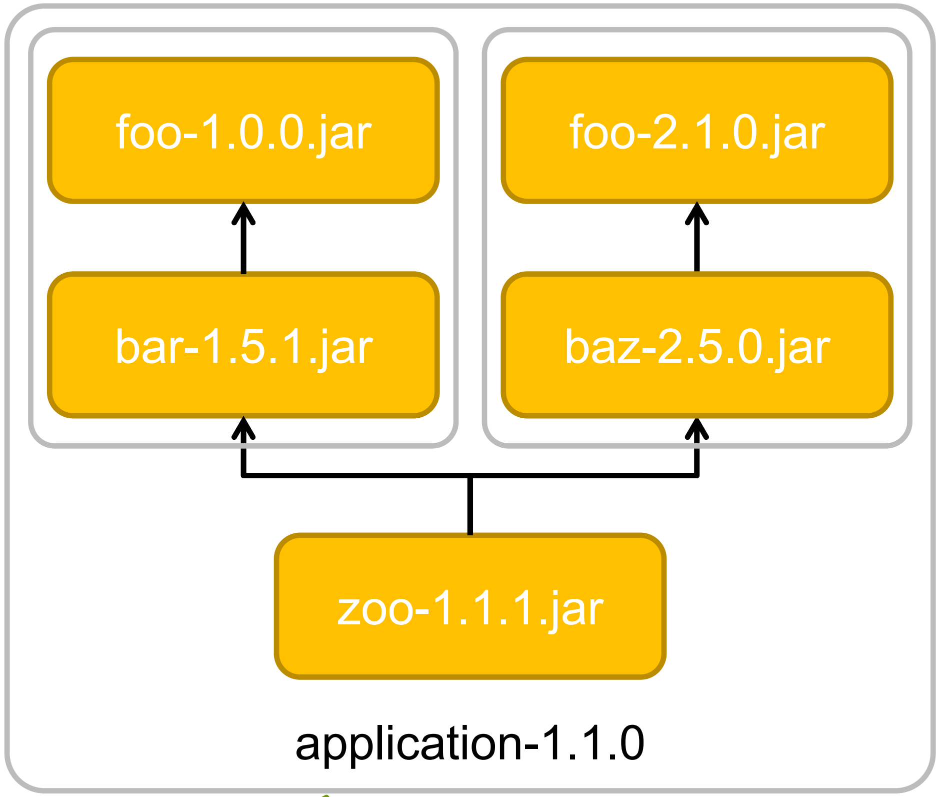
About semantic versioning

- ✓ Version information is compact
- ✓ Core rules look simple enough
- ✓ Version ranges are flexible, handling both backward and forward compatibility
- ✓ Feasible to process mechanically when the versioning information is present

- ✘ Often difficult in practice, however
  - Imperfect practices
  - Neglectful developers
  - Misunderstanding what all the rules imply and affect (we'll see some such "surprises" later)

Modular heaven: a solution?

- Modules provide imports and exports
- Modules employ semantic versioning
- The modular system can ensure consistent module arrangement in runtime
- The modular system can prevent an incomplete or broken application from running (and therefore crashing after some time)
- Implementation details do not disturb dependent modules



The developer must “just” remain disciplined and supply always correct metadata.

OSGi design: service-oriented

❑ **Service** ≈ a discrete unit of functionality

- Services form additional encapsulation boundaries
- Services interact via mutually known interfaces
- Using interfaces support effective hiding of implementation details
- Hiding implementation details makes easier to avoid clashing module dependencies

➢ Application is a composition of collaborating, but well-encapsulated services

OSGi design: dynamic environment

- ❖ Support for dynamic changes of many kinds

- ➢ Deliberate design choice: not necessary and overkill for static applications, but increases the flexibility of the framework in an awesome way

- ▪ Supports resource conservation
- ▪ Supports dynamic deployment and updating the application without turning around
- ▪ Makes interactions with real environment easier: suitable, e.g., for IoT controllers

OSGi design: versioning

- Using semantic versioning strictly

- Deliberate design choice: not necessary, but convenient and coherent with other choices

- Code versioned on the package level and version information is taken into account for runtime dependencies resolution
- Enables more flexibility for service interfacing
- Reduces the probability of version conflicts
- Saves resources (code sharing possible more often)

OSGi foundations

- About OSGi framework
- Comparison with other frameworks
- Architecture overview

What is OSGi?

"OSGi technology is a set of specifications that defines a dynamic component system for Java."

osgi.org

- Mature technology (conceived in 1998)
- Evolved beyond service gateway framework into a modular system and a service platform for Java
- Specifications updated and maintained carefully (by *Open Service Gateway initiative* formerly, then by *OSGi Alliance* and now *OSGi Working Group*)
- The specifications merely define the framework, there are multiple implementations

Why OSGi?

- Provides a way how to deal with versioned code (and so far it provides much more powerful approach than JPMS)
- Provides a truly dynamic environment (it is possible to modify an application in run-time)
- Provides a mature and stable, specification-backed environment
- Enables long-term gradual evolution of complex applications
- Exhibits the design of a comprehensive modular system

You might hear about OSGi

- It is too complex/complicated
- It has a steep learning curve
- It has no documentation
- It does not work with…
- The tooling sucks
- You don’t need it

Well… we’ll see…

An OSGi container

- **OSGi container** ≈ an instance of an OSGi framework, providing the OSGi runtime for an application hosted by the container

- **Hosts a single application (usually)**
- Supports open & modular application architecture
- Provides life cycle support and other services, but lets the application choose its model
- Versatile, uses range from embedded devices to servers and integration of enterprise applications
- Embeddable into other containers or applications, offering a number of interoperability options

OSGi × JEE containers

A JEE container…

- **Hosts multiple applications and keeps them isolated**
- Needs applications as closed self-contained lumps
- Controls the applications fully, defines application model and restricts the application
- Favors server-oriented approach, usually providing many additional functionality
- Aims to centralize services and resources to improve their management and relieving applications from taking care themselves

OSGi × Spring Framework

Spring Framework…

- **Provides environment to a single application**
- Central component is the IoC container
- Additional modules support building JEE-like applications (and runs on the top of JEE)
- Ad-hoc framework, no underlying specification
- Reflection and DI are the principal workhorses
- Heavy reliance on annotations and XML descriptors which the framework can automagically process
- RAD frameworks emerged, attempting to mitigate Spring Framework complexity, e.g., Spring Boot

OSGi × Quarkus

Quarkus (and other micro-frameworks)

- **Runs a single application**
- Based on JEE standards, resembling Spring Boot
- Not a container, but embedded runtime
- Ad-hoc framework, no underlying specification
- DI and IoC are the principal workhorses, but used mostly in a static way
- Heavy reliance on annotations which the framework can automagically process
- Aiming for low footprints
- Suitable for closed-world applications, therefore supporting well compilations into native applications

OSGi × JPMS

Ready for Java 9?

JPMS (Java Platform Module System)

- **Module system for the platform, but not so much for applications** (as not so convenient and not mandatory… yet)
- Aims to unify compile and run time code resolution with higher fidelity and reliability
- Aims to increase security
- Rather static and significantly limited, not actually solving many of the problems we mentioned
  - Coarse-grained imports (whole modules only)
  - No versioning and code evolution support
  - No actual life cycle support
  - Service support limited to `ServiceLoader`

OSGi implementations

- Framework implementations
  - Apache Felix
  - Eclipse Equinox
  - Eclipse Concierge
  - Knopflerfish

- Many other projects implement additional OSGi specifications (e.g., Apache Aries)
- Distributions with many features available (e.g., Apache Karaf)

☺ OSGi-compliant code can run in any OSGi container

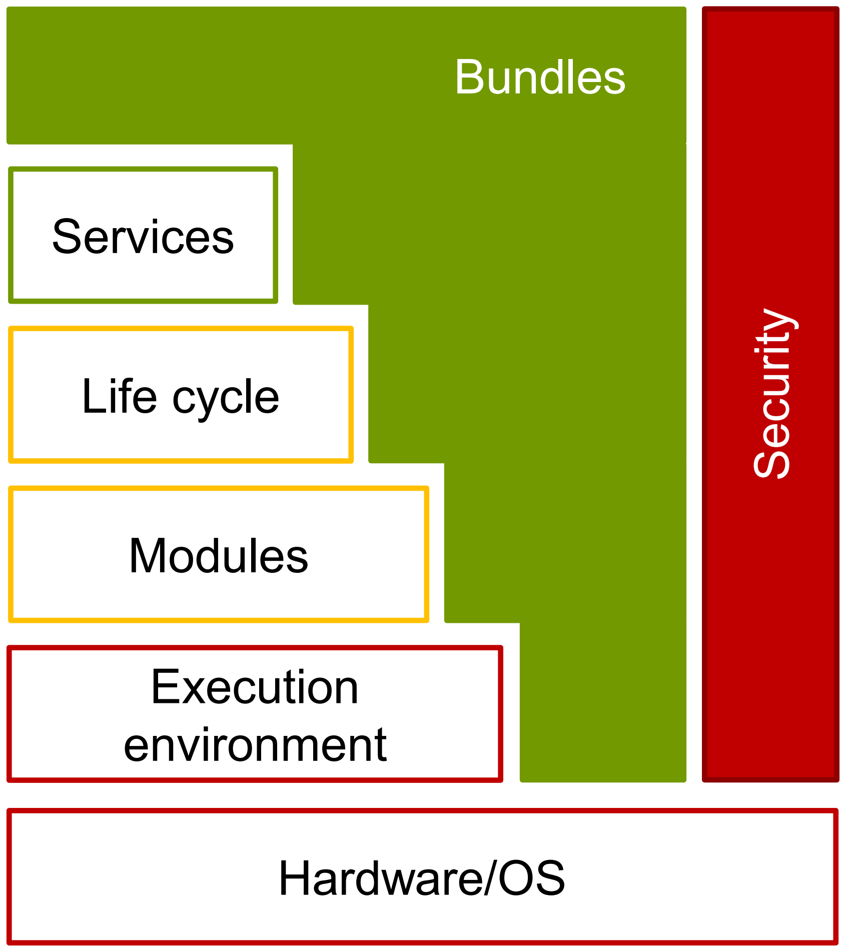
☺ Interoperability projects exist (e.g., Apache Felix Atomos)

Hands on OSGi

Let’s have a look at an OSGi Hello World application

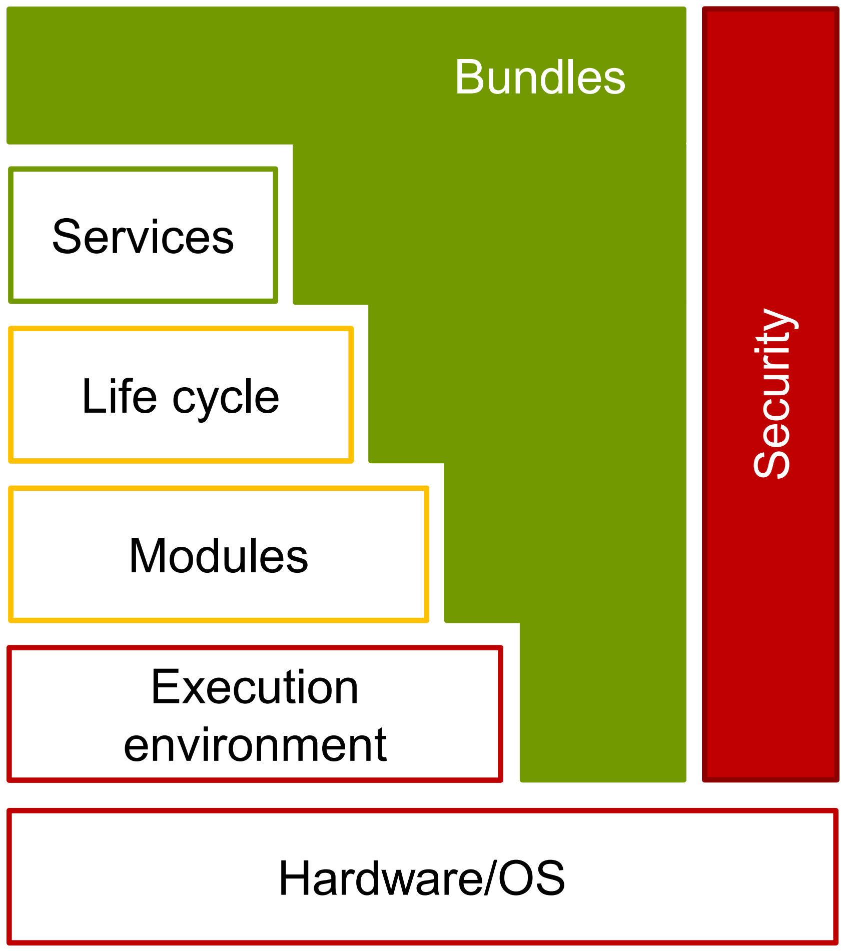
1. Have a running OSGi container
2. Create a Java project to build a bundle
3. Make the bundle display something on its *start* and *stop* actions
4. Deploy the bundle and use the container console to trigger an action

Framework architecture I



- **Bundle** is an extensible application (or an application part) deployed in the OSGi container
- **Service** layer provides a dynamic, concise and consistent programming model for bundle developers

Framework architecture II



- **Life cycle** layer provides a runtime model for bundles
- **Module** layer defines a modularization model
- **Security** layer, based on Java security, is optional

Anatomy of a bundle I

- Bundle is an ordinary `.jar` file with extended `META-INF/MANIFEST.MF`
  - Additional entries provide information for the module and life cycle layers (mostly)
  - Essential module description parts: bundle identifier, package exports and imports
  - Life cycle control may be requested by providing `BundleActivator`

- There is no principal problem with libraries provided as OSGi bundles

Anatomy of a bundle II

- A bundle may contain additional metadata
  - Usually defined by a particular specification like Declarative Services
  - Stored in `OSGI-INF/`
- Documentation of a bundle may be stored in `OSGI-OPT/`

Making a bundle

- Supply the additional metadata by hand
  - Always possible, no special tooling needed
  - Difficult to maintain
  - Error-prone

- Employ a proper toolchain
  - Basically all existing tools are based on *bndlib*
    - Similar configuration, based on *bnd* syntax
    - Quite consistent results

  - Pick your choice: Bndtools, Gradle, Maven

Bundle identifiers

- Bundle symbolic name and version
  - Assigned by the bundle developer
- Bundle location
  - Determined by the operator when deploying the bundle, unique within a framework
  - Must not change when updating a bundle
- Bundle identifier
  - A unique persistent non-negative number assigned to a bundle when installed
  - System bundle has identifier 0 (zero)

Interlude

Semantic versioning

Semantic versioning in a nutshell

➢ Version information is stored as a version number (a tuple actually)

- Version number: *major.minor.micro*
- Version number has semantics (it is not arbitrary and must obey some rules)

⇘ Compatibility is obvious just from comparison of version numbers

Version component	Meaning
micro	Indicates implementation changes that are not publicly visible (e.g., a bug fix or a security update)
minor	Indicates compatible changes of previously published parts (e.g., adding methods, classes/interfaces or packages)
major	Indicates incompatible changes of previously published parts (e.g., renaming or removing anything published earlier)

We know that already

Capabilities and requirements

- **Capability describes a function of a resource**
- **Requirement asserts on the availability of a capability**

- Both have a namespace, attributes and directives
- A requirement can be satisfied when a matching capability is available
- A bundle may be used only if all its mandatory requirements are satisfied

- Dependencies on bundles and packages are a special case of a more general mechanism that OSGi uses

Dependencies and versioning

- Bundle versioning is mostly retained for backward compatibility and for build systems using artifact versioning
- Package versions are actually more important
- When package versions are not managed properly and are not specified, tooling defaults to bundle version then

Why package versioning?

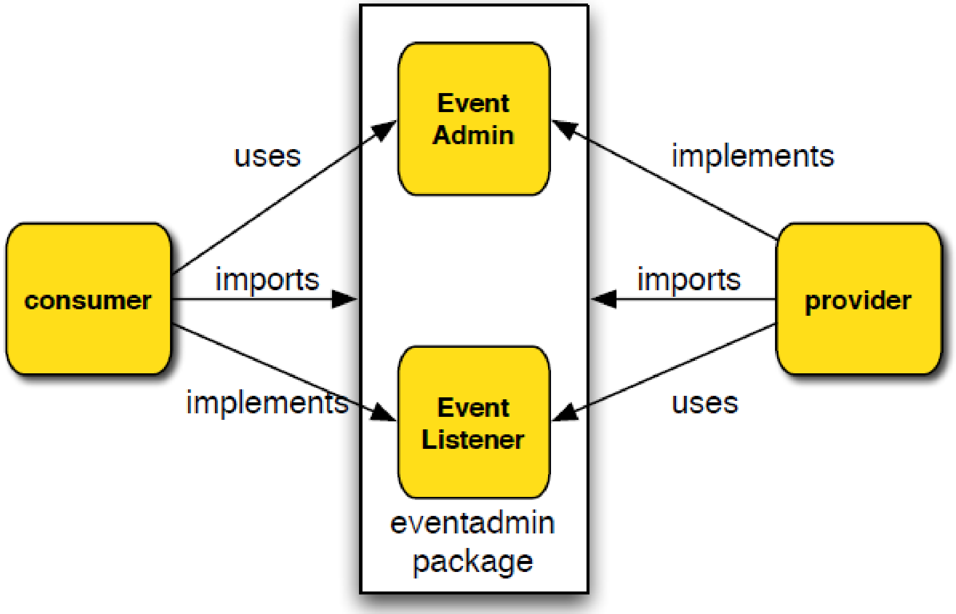
- Recall Java fundamentals
  - Package encapsulation
  - Package cohesion
  - Class-loading model
  - Acyclic dependencies (should be!)

- Keeps the track of bumping the aggregate version
- Reduces the odds of conflicting dependencies
- More flexible for refactoring of delivery units
- Not depending on bundle identifiers that may differ due to repackaging, re-exports etc.
- More efficient for runtime

Using the right (package) version

- **In compile time** use the lowest versions of the dependencies that you can use
- **In run time** supply the highest compatible versions of the dependencies
  - Unless there is a specific requirement (e.g., known bugs or dependencies that you may not use)
- Be careful when omitting a version
  - It is probably a tricky mistake in `MANIFEST.MF`

The right version depends on…



❖ Exported types form (always) an API

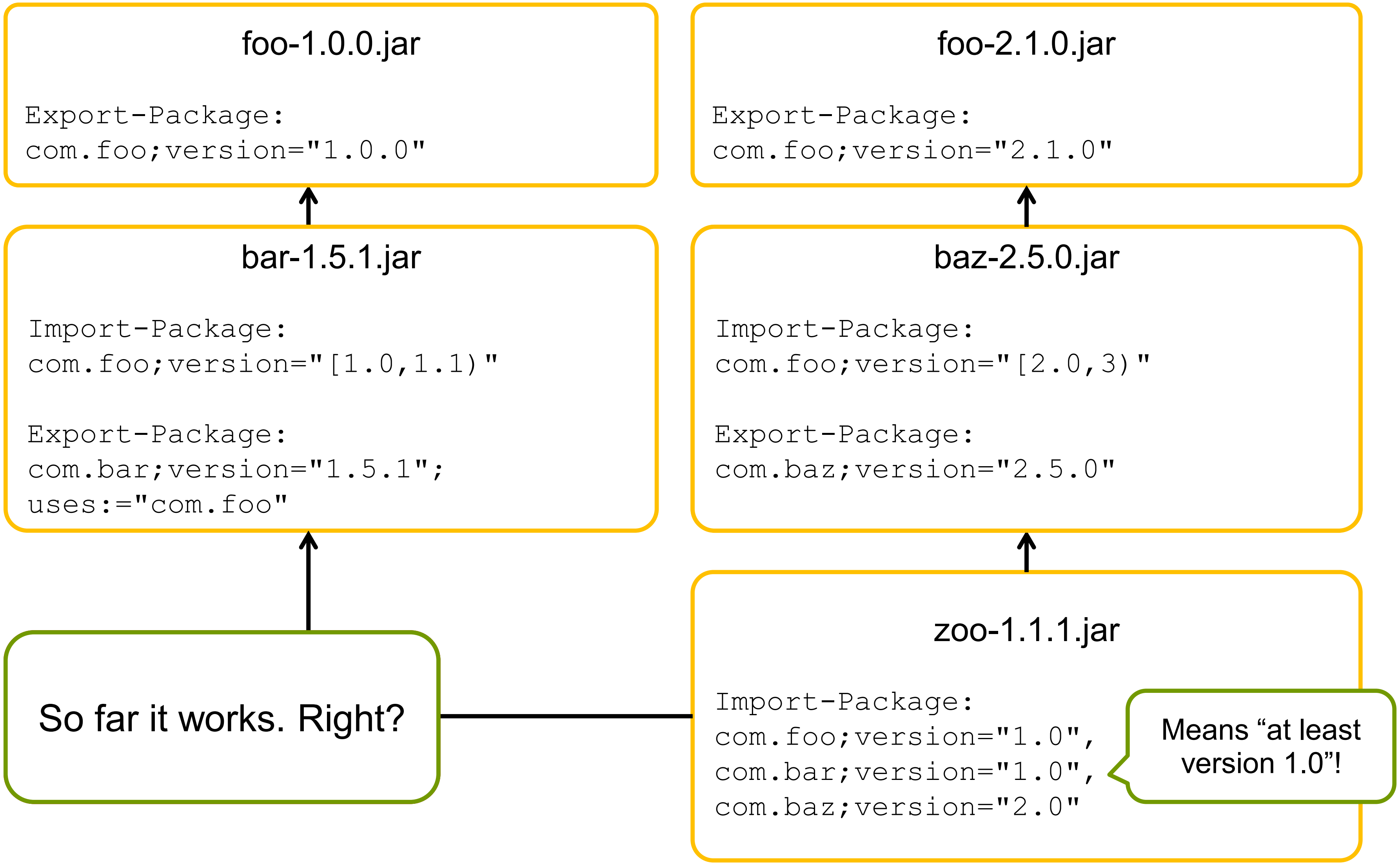
- A type acts as a **consumer type** when consumers (clients) of the API use it or implement (inherit) it
- A type acts as a **provider type** when only providers (implementations) of the API may implement (inherit) it

Type roles

- ❖ Consumer type
  - All imports may use version range up to the next major version
  - (Almost) any change in the type’s public part requires publishing a major version update
- ❖ Provider type
  - Consumer imports may use version range up to the next major version
  - Providers must import within the same minor version
  - As the result even a minor version change implies updating the providers

- Enjoy `org.osgi.annotation.versioning` to declare the roles easily

Transitive dependencies



foo-1.0.0.jar

```
Export-Package:
com.foo;version="1.0.0"
```

foo-2.1.0.jar

```
Export-Package:
com.foo;version="2.1.0"
```

bar-1.5.1.jar

```
Import-Package:
com.foo;version="[1.0,1.1)"

Export-Package:
com.bar;version="1.5.1";
uses:="com.foo"
```

baz-2.5.0.jar

```
Import-Package:
com.foo;version="[2.0,3)"

Export-Package:
com.baz;version="2.5.0"
```

zoo-1.1.1.jar

```
Import-Package:
com.foo;version="1.0",
com.bar;version="1.0",
com.baz;version="2.0"
```

Means “at least version 1.0”!

So far it works. Right?

Transitive dependencies: rules

If an exported package depends on another package…

- Private dependencies remain private (no matter where they come from)
- Non-private dependencies (visible in the exported types) must be declared by `uses` directive to ensure compatibility consistency

➢ Fortunately, tooling can do the computations

Imports & exports can be complex

- An import may be *optional* (though usually not a good idea)
- An import may be *dynamic* (usually not a good idea either)
- An import may require a specific bundle to satisfy the import
- An import may require arbitrary mandatory attributes that the export must declare
- An export may exclude particular classes

Cookbook

API separation

Brief introduction to services

- **Service ≈** a component providing access to one or more capabilities
  - The access is provided using an interface
  - The implementation is hidden and private

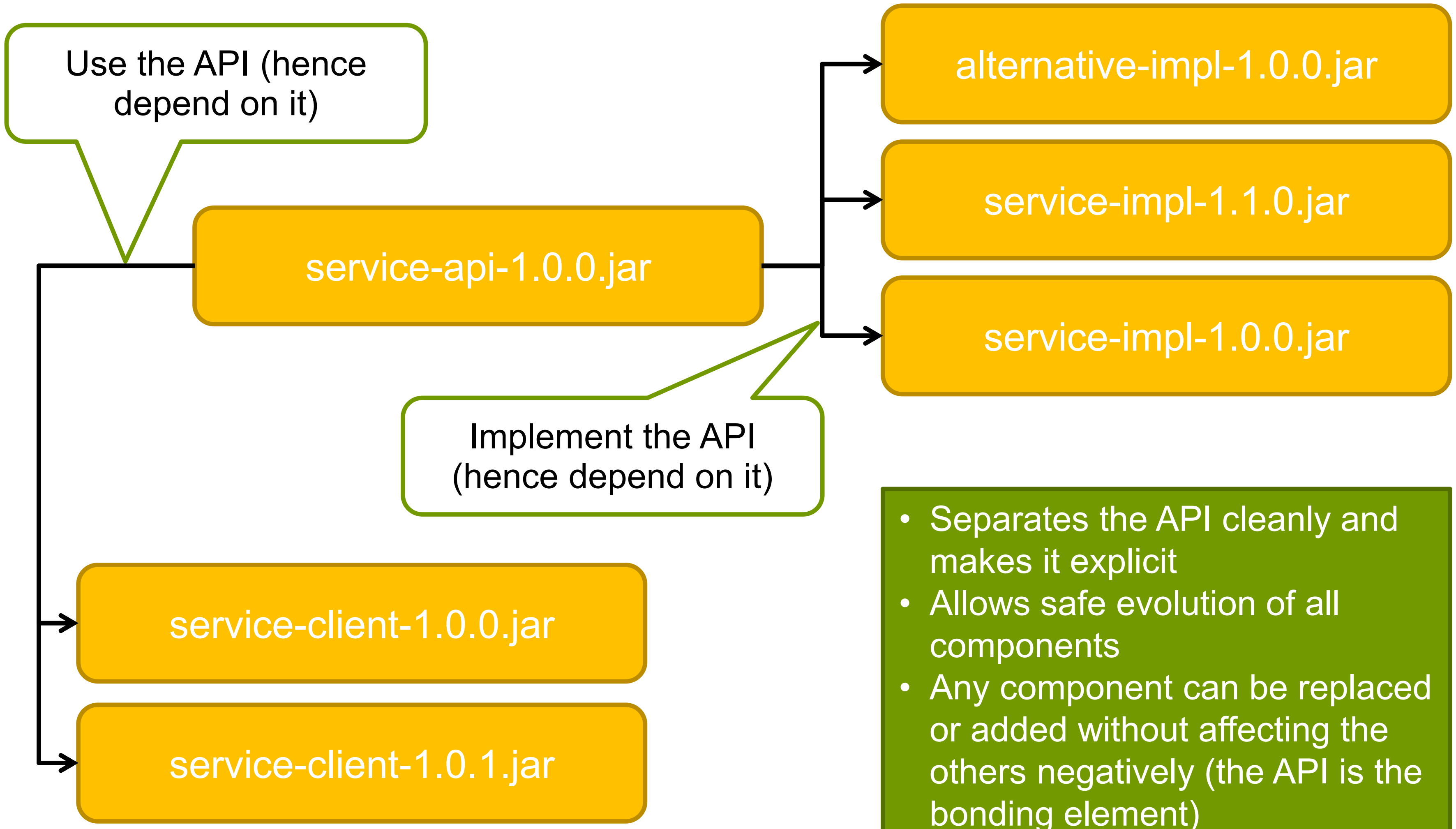
- Service providers × consumers
  - Both parties are independent (decoupled)
  - Sharing a mutually known and agreed API

API × implementation

- Service interface/API
  - Should be in a separate bundle
- Service implementation
  - A separate bundle recommended
  - Nothing related to the implementation should be exported from the bundle (public parts are already in the interface/API)

➢ Consider making the implementation OSGi-agnostic with an OSGi-enabling bridge for it
  - Better reusable elsewhere
  - Easier testing (no need to employ OSGi-specific support)
  - One of the recommended approach to mitigate dangers of stale references

Providers and consumers



- Separates the API cleanly and makes it explicit
- Allows safe evolution of all components
- Any component can be replaced or added without affecting the others negatively (the API is the bonding element)

Interlude

Class loading

How JVM deals with classes

The very first step is finding and loading the class representation

- Array classes are rather generated directly by JVM from the non-array classes (which must be found eventually)
- If the class is not an array class, JVM uses a *class loader* for getting the class’s representation

What is a class loader

- ❑ **Class loader** (CL) is an object which is responsible for finding and loading classes on behalf of JVM
  - JVM does not load **external** class representations itself, it delegates finding and loading them to CLs
  - CLs create no **run-time** class representations, they just pass the external class representations to JVM for that

- ➢ `ClassLoader` is the base for all CLs
  - It’s an abstract class, the platform libraries provide a few non-abstract implementations
  - It can be inherited for making custom CLs

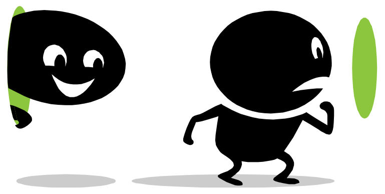
The purpose of a class loader

- Loading code and resources from various sources
  - A class file needn't reside in an actual file in the local file system, it can be retrieved from a network stream

- Customizable loading for many purposes
  - The CL decides what JVM gets
  - ⇨ Instrumenting and tailoring classes
  - ⇨ Generating classes on the fly

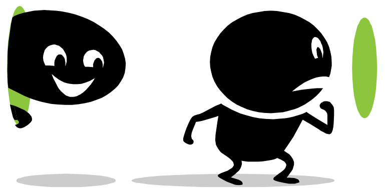
- Support for run-time type safety
- "Unloading" unnecessary classes

Class loaders and classes



- A class loader may define a class directly (it is the **defining class loader**) or delegate to another class loader (it is the **initiating class loader**)
- A class in runtime (**run-time class** or RTC for short) is determined by the pair of the class’s binary name and its defining class loader
- Each RTC belongs to a **run-time package**
- When the code of a RTC triggers loading another class, the defining class loader is used for loading that class

Run-time classes

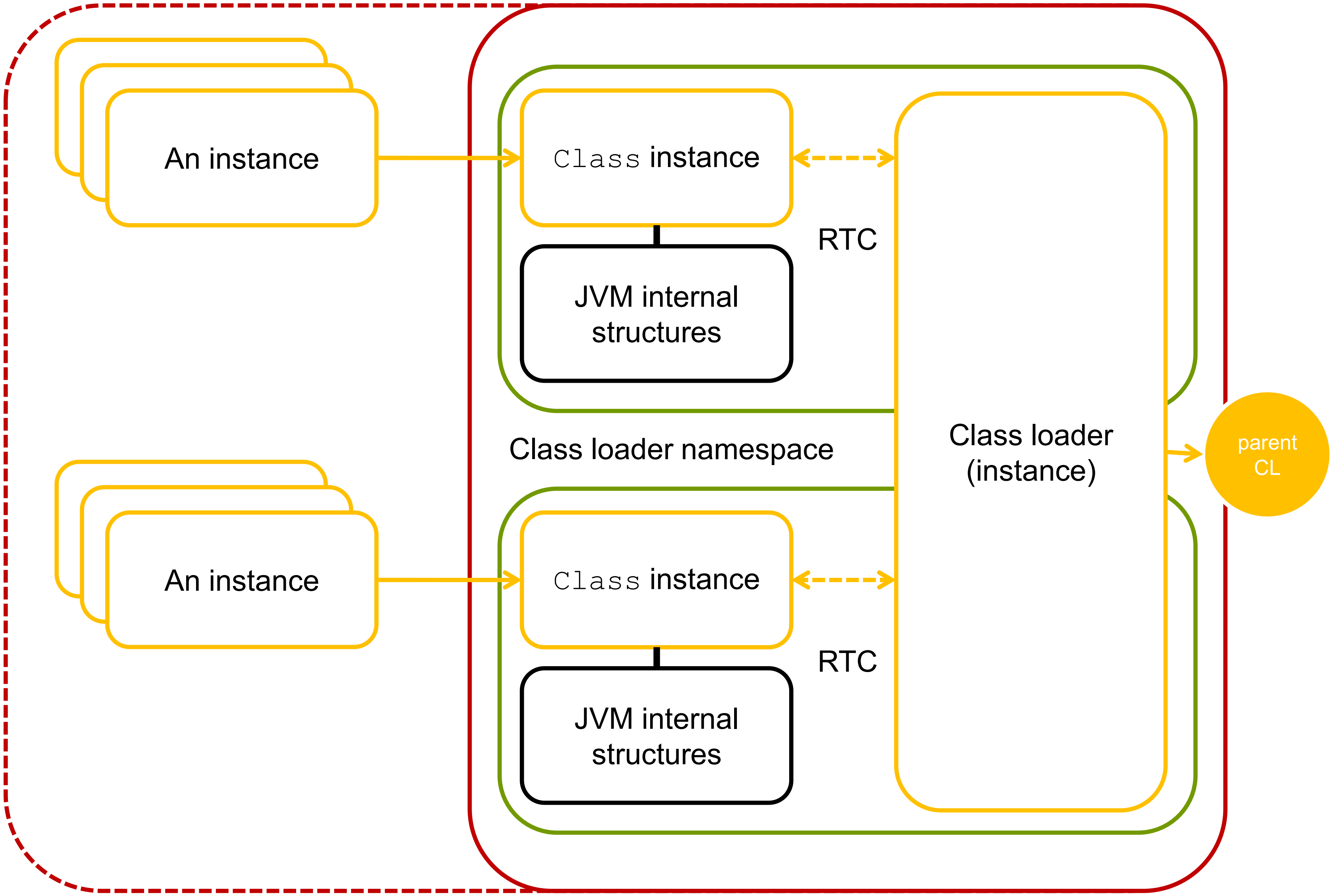


- ➢ Each class loader defines a run-time class namespace (or run-time class space)
- ➢ Loading constraints guard the consistency of run-time packages and class namespaces

↳ Loading a class by two different class loaders results in two distinct RTCs

‽ RTCs are considered in all run-time type checks (casting, loading a reference etc.)
  - The run-time view differs from the compile-time view (which is checked by the compiler)
  - ↳ Therefore compile-time static type checking does not yet guarantee run-time type compatibility

Run-time associations



Why to have run-time types?

✓ Ensuring type-safe linkage

- A class name does not guarantee its uniqueness (e.g. two versions of a class)
- The source (defining CL) can be used to ensure the uniqueness (JVM prevents re-loading a class)

✓ Prevention from subverting the type system

- Even a malicious CL or a compromised class source can subvert only the particular class namespace, but not the whole JVM
- Essential and trusted classes (from trusted sources) are then safe

Class unloading

- ❑ **Unloading a class** ≈ releasing the RTC (i.e. the appropriate `Class` instance) and related JVM structures, reclaiming occupied memory

JVM **may** unload classes

- A class may be **unloaded** if and only if its defining CL may be reclaimed by the garbage collector
- No live references may exist (including those which might be resurrected by finalizers)
- Once a class is unloaded, it can’t be loaded again
  - The identity of the CL and `Class` instances is lost – therefore the previous state can never be restored

Class reloading

- ❑ **Reloading a class** ≈ creating a new RTC (i.e. the appropriate `Class` instance) for a class which was already loaded by the particular CL
  - ‽ Loading a class with a different CL instance is not reloading
    - – Because a different RTC is created

JVM **must not** reload a class

- ▪ Loading/initializing a class may have external side effects
  - • Reloading a class is not transparent in general
  - ⇘ Therefore it is forbidden to reload a class
- ▪ Reloading a class is even not possible anyway: the defining CL is lost as a class may be unloaded after its CL is released definitely
- ▪ Therefore when a web/application container “reloads” an application, it actually forgets the application’s CL namespace branch, creates a new one and starts it again (relying on JVM’s unloading capabilities)

Class loading model

- Core platform classes (e.g., `Object`, `String`, or `ClassLoader` itself) are loaded by the **bootstrap CL** (a.k.a. primordial CL)
- Platform uses additional CLs (since Java 9 defined more precisely than before)
- Applications may create custom CLs

- Class loaders often delegate
  - The most common model uses delegation to parent, making a tree CL hierarchy
  - Applications use other models sometimes

- When targeting an environment, it is necessary to know its class loading model

Under the hood

Class loading model

Bundles and class loaders

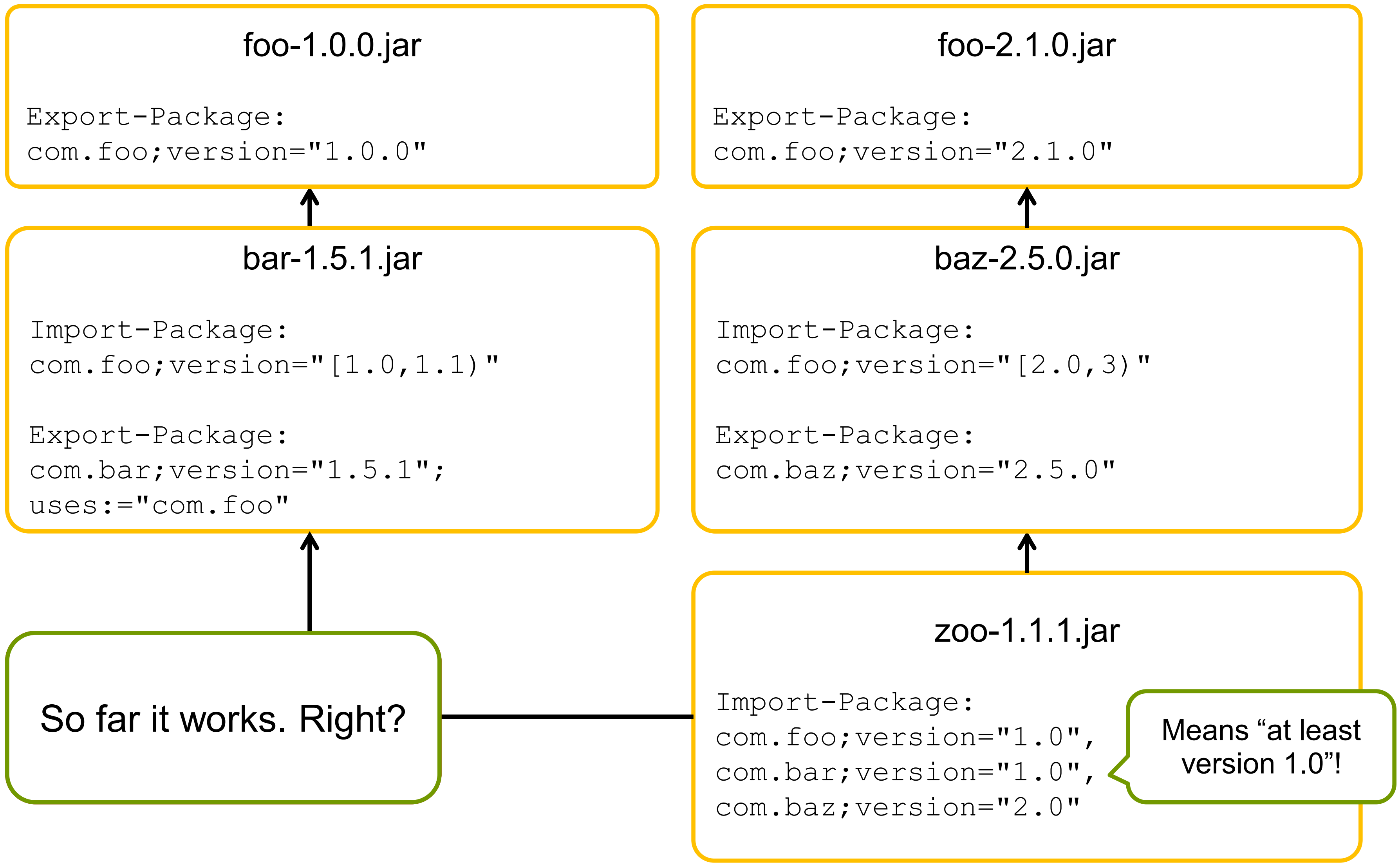
- ➢ Each bundle has a single class loader
  - **Bundle space** consists of the bundle's `.jar` and additional closely coupled `.jar` files (e.g., *fragments*)
- ➢ There are special class loaders
  - Boot class path
    - – Provides `java.*` packages (and perhaps more, depending on the framework implementation)
  - Framework class path
    - – Contains the framework implementation
  - External class loaders belonging under OSGi Connect
    - – OSGi Connect makes accessible content living outside of the framework itself

Delegation model

- Class loaders form a class loading delegation network
- **Class space** contains all classes reachable from a given bundle's class loader
  - The parent class loader (packages from the boot class path)
  - Imported packages
  - Required bundles
  - The bundle's class path (private packages)
  - Attached fragments

- A class space must be consistent, such that it **never** contains two classes with the same fully qualified name
- Separate class spaces may contain classes with the same fully qualified name
- Multiple versions of a class in a JVM instance are supported

Dependencies: recap with CLs



foo-1.0.0.jar

```
Export-Package:
com.foo;version="1.0.0"
```

foo-2.1.0.jar

```
Export-Package:
com.foo;version="2.1.0"
```

bar-1.5.1.jar

```
Import-Package:
com.foo;version="[1.0,1.1)"

Export-Package:
com.bar;version="1.5.1";
uses:="com.foo"
```

baz-2.5.0.jar

```
Import-Package:
com.foo;version="[2.0,3)"

Export-Package:
com.baz;version="2.5.0"
```

zoo-1.1.1.jar

```
Import-Package:
com.foo;version="1.0",
com.bar;version="1.0",
com.baz;version="2.0"
```

Dynamic imports

- Sometimes a bundle may need to load a class by name which is not known in compile time
  - This is not the usual OSGi way as OSGi offers for more superior approach to composing dynamic applications
  - Usually applies to OSGi-fied libraries that were developed for different class loading model

- `DynamicImport-Package` header can help
  - Used as the last resort when standard class loading mechanism are not able to locate the class
  - Tricky, it may have unexpected consequences

Fragment bundles

❑ **Fragment bundles** have no own class loader

- Instead, a fragment bundle is attached to one or more **host bundles**
- Fragments may have no activator either and a particular attachment shares the life cycle with the host bundle

➢ Key use case: localization resources

- Development and shipping independently from the main bundle
- Supports `java.util.ResourceBundle` (thanks to the same class loader)

‽ Often abused whenever something must be injected in a bundle's class space

- Usually a very wrong idea

Expert corner

Class loading troubles

How to crash and burn

- Wrong imports or exports (or lying about them)
  - Arguably the worst sin as it undermines everything that module layer attempts to provide

- Optional imports
  - Safe use requires the code to be explicitly ready to miss some dependencies… which is rare and alien to OSGi world
  - Often abused just to mute resolution problems that are not correctly understood… which merely turns them in a ticking bomb (even if it works – now)

- Dynamic imports
  - Last resort, or rather a desperate attempt, to cope with non-OSGi approach to locate a class
  - Often appears as `DynamicImport-Package: *` (which removes any compatibility safeguards and targeted class loading delegation)

Thread context class loader

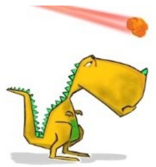
❑ **Thread Context Class Loader** (TCCL) is associated with a thread (a thread-local variable: `Thread::getContextClassLoader`)

▪ Introduced originally because of RMI and for JEE
  - But soon exploited by many other frameworks and libraries as first or last resort for loading the right class

✘ There is no actual specification how to use it
  - Seemingly every library/framework uses it… somehow (except for OSGi)
  - Clashes may occur

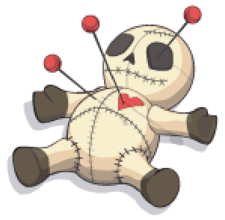
? Ad hoc solution suitable for some applications only
  - 😐 Application entry points are well-defined in JEE and under the container's control
  - ✘ Hardly usable in modular systems where the context boundaries are not so clear and entry points needn't be defined – hence their concepts do not require TCCL at all, on contrary TCCL clashes with them

How to have a problem



- Using `java.util.ServiceLoader`
  - Much weaker than OSGi service support
  - Hardly usable in such a complex class loading model: basically doomed to miss the service, thus needing hinting where the service load from (beware of TCCL!)
  - Fortunately, there is Service Loader Mediator (in OSGi Compendium)
- Meeting other kinds of various SPIs
  - Reverse engineering often works to find out what kind of a hack must be taken to mimic the expectations of the SPI
  - Often TCCL is involved
  - Outstanding example: `java.sql.DriverManager`

How to be cursed by everybody



The most dangerous thing to use in OSGi:
`Class::forName`

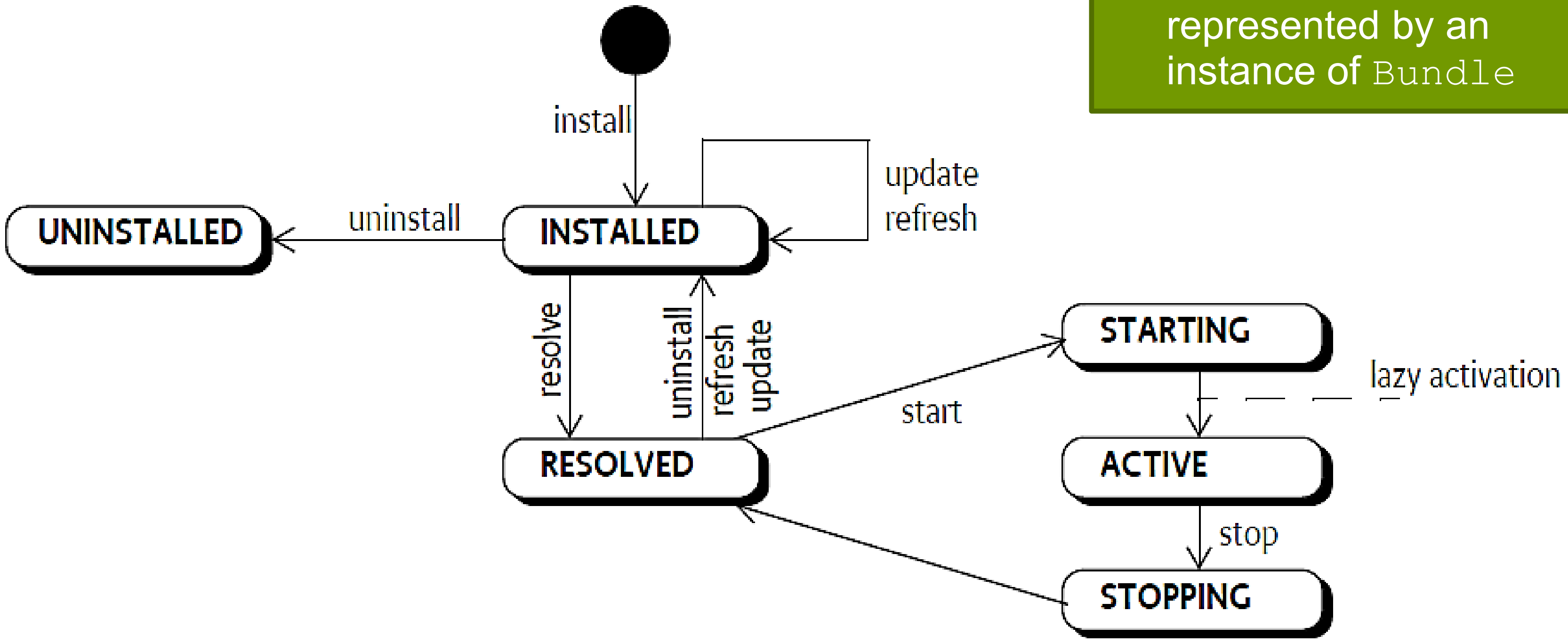
- No place for it in OSGi, it offers better means
- If you really have to use it, understand all details of OSGi class loading model (or you are doomed and everybody will curse you)
- Especially dreadful when used with TCCL
- Actually… have you already tried to make any SPI work in OSGi? They all do this at the end!

OSGi core

- Bundle and framework life cycles
- Service layer
- Trackers
- Service development cookbook

Bundle life cycle

- An installed bundle is represented by an instance of `Bundle`



- A bundle class loader must be (re)created after resolving the bundle and may be even deferred until the bundle must be activated

Bundle activation

- Bundle may be used without activation
  - Being `RESOLVED` is good enough for loading resources, including classes (typical for libraries)
  - Activation occurs when the bundle is started (explicitly or due an activation policy)

- When activated, a new `BundleActivator` instance must be created (if any present)
- A new `BundleContext` is created as well for each bundle activation
- A `BundleActivator` instance is never reused, its `BundleContext` becomes invalid when the bundle stops

BundleContext

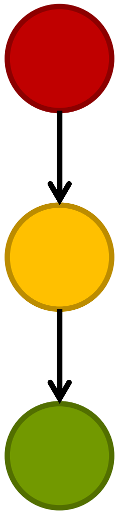
- Represents the execution environment for an active bundle
  - Therefore an instance remains valid as long as the bundle incarnation remains active
- Impersonates the bundle
  - Therefore avoid borrowing outside of the bundle context
- Provides means for interacting with the framework, e.g., registering services

About services

- **Service** ≈ a component providing access to one or more capabilities
  - The access is provided using an interface
  - The implementation is hidden and private
  - OSGi services have a dynamic life cycle

- Any object may become an OSGi service
  - Usually, a service is defined by a Java interface and its implementations are provided as service instances

Why services?

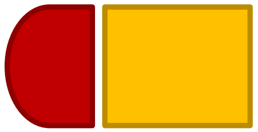


Core design principles:
focus + delegation + composition

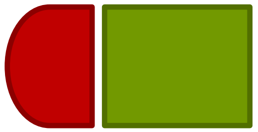
- Focused components do just one thing (and do it properly)
- All other things are delegated to other components
- Complex components are composed from simpler ones

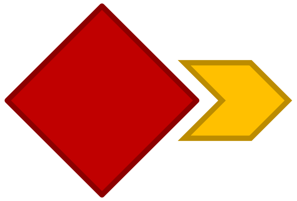
Prerequisite for flexibility & extensibility

OSGi services concept follows these ideas and enables both flexible and extensible, dynamic solutions

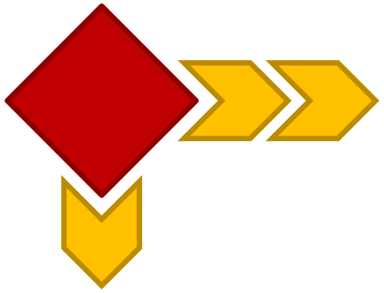


Flexibility ≈ any component can be replaced with another having the same interface





Extensibility ≈ the number of cooperating components is not fixed and is unknown in advance

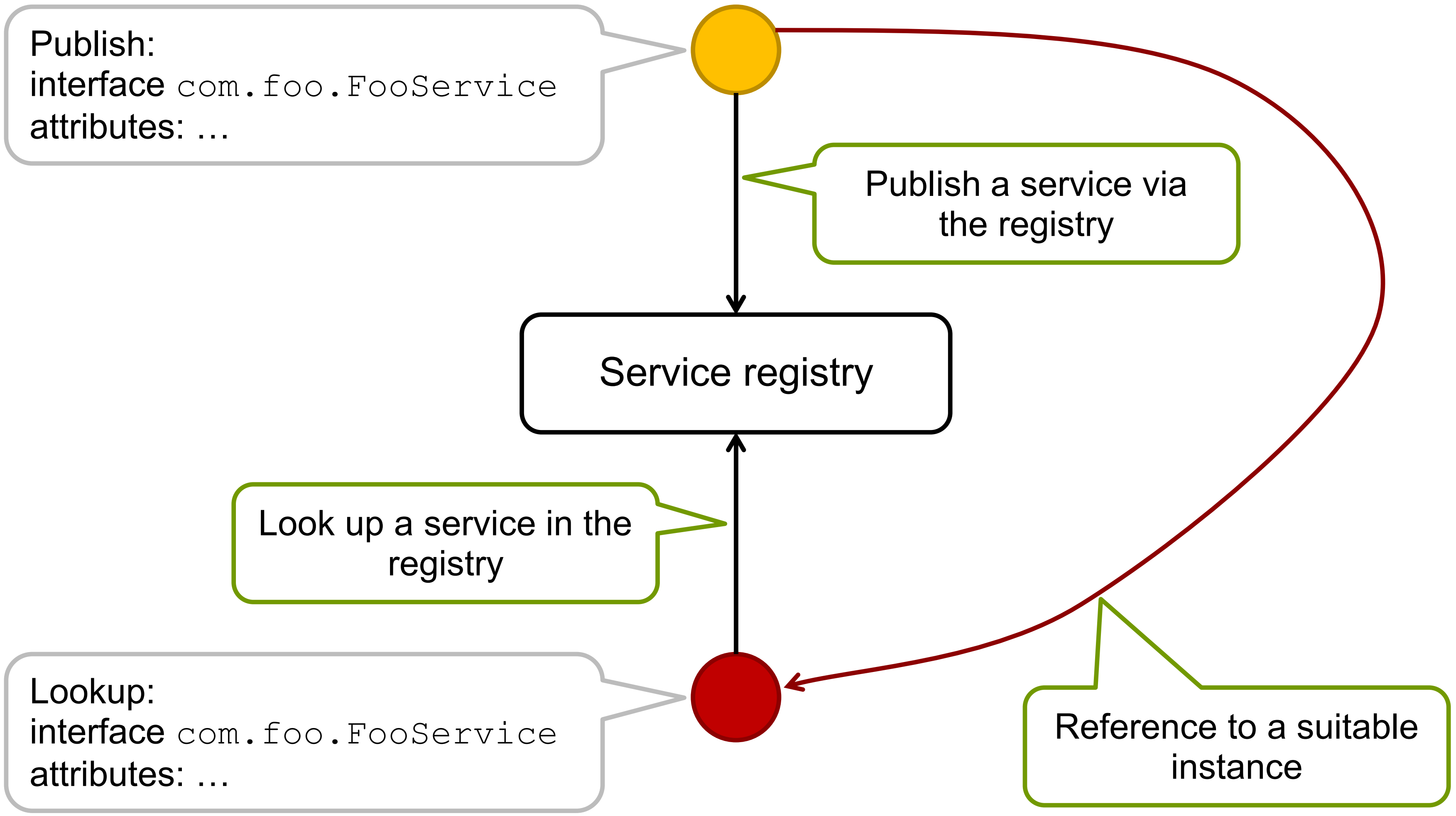


Hands on services

Let’s implement and use a service

1. Assuming a service implementation exists, look it up and use
2. How does it work? Well…
3. Implement the service that should have been used
4. How does it work now?
5. Let’s contemplate for a while

Service registry: the corner stone


Publish:
interface `com.foo.FooService`
attributes: …
Publish a service via the registry
Service registry
Look up a service in the registry
Lookup:
interface `com.foo.FooService`
attributes: …
Reference to a suitable instance

Exercise aftermath: considerations

- What if no service is available?
- What if the service in use becomes unavailable?
- What about repeated lookup costs?
- Why to indicate that a service is used no longer?
- What if a service registration is never cancelled (and its bundle stops)?
- What if there are multiple services available?
- Do `BundleActivator` methods have to be synchronized?

More questions? Please?

Exercise aftermath: summary

- Service registration
  - Bound to a registering bundle and limited by its life cycle (i.e., registrations do not outlive the registering bundle)
  - Associated with properties that can be observed by service clients (and used in lookup filters)
  - Represented with a `ServiceRegistration` instance for controlling the registration (and service properties)

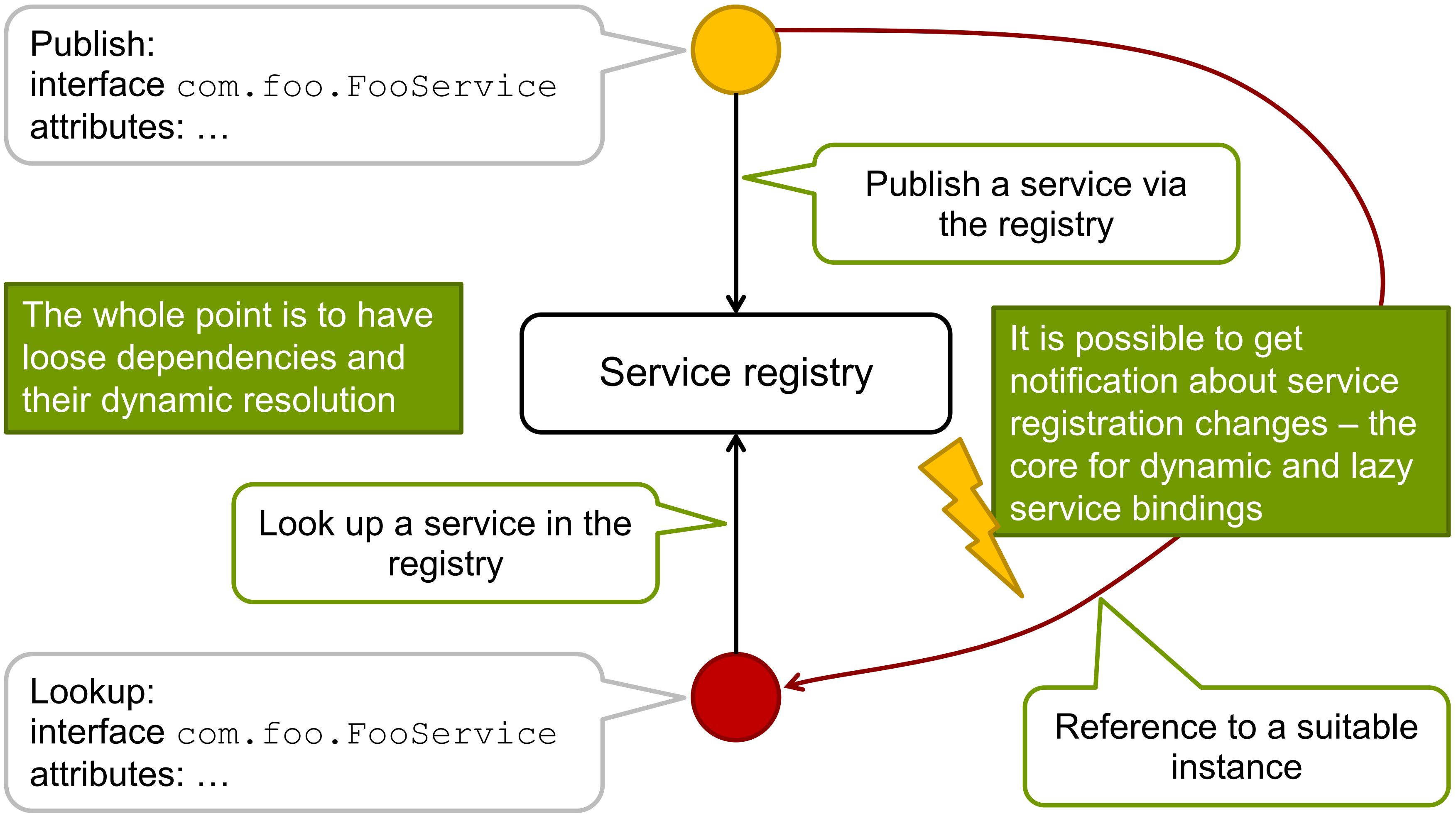
- Service use
  - Related to `ServiceReference` instances that allows accessing the service object and the service properties
  - Employs reference counting and tracks bundles that use a service

- Lookup provides a `ServiceReference` rather than the service object directly (which enables more advanced uses)
- Service ranking solves usually the case of multiple services

Service registry: a closer look

- Service registry exists implicitly, there is no dedicated interface
- All interactions via `BundleContext`, which identifies the calling bundle and thus allows associate various requests to particular bundles
- A service can be registered and unregistered (almost) any time, service properties can change as well
- Changes are propagated via events and they are handled by listeners

Service registry: the whole picture


Publish:
interface `com.foo.FooService`
attributes: …
Publish a service via the registry
The whole point is to have loose dependencies and their dynamic resolution
Service registry
It is possible to get notification about service registration changes – the core for dynamic and lazy service bindings
Look up a service in the registry
Lookup:
interface `com.foo.FooService`
attributes: …
Reference to a suitable instance

Framework and events

- All major operations fire events that the framework distributes to appropriate listeners
  - Event-driven approach is the core of the framework architecture
  - Employing events and listeners allows extending the framework significantly without any actual core changes (see later, e.g., Declarative Services)
- Listeners must (usually) count with various concurrency-related issues
  - Not suitable as the regular programming model, but it is possible to build more comfortable tools on the top of the event-driven model

Tracker API

- ➢ Trackers provide a safer and more comfortable approach than listeners
  - While employing the listeners, they hide most of the nasty and difficult event processing
  - Available for services and bundles, see `ServiceTracker` and `BundleTracker`

- ‽ Tracker callbacks are still a minor challenge
  - Callback code must avoid holding locks
  - Callbacks must be re-entrant
  - Callbacks are synchronous… and should be fast

Launching a framework instance

- Framework instance
  - Represented by an instance of `Framework`
  - Represented as a `Bundle`, known as the system bundle, with reserved bundle identifier 0 (zero)
  - Besides custom launchers `FrameworkFactory` can make a new instance

- Launching and managing a framework
  - All required operations are available in the `Bundle` interface already – thus nothing special is needed

Support for embedding

- Lifecycle management is defined and specified well, and it can be controlled programmatically
- A framework instance is a naturally isolated world, self-contained and independent on outer environment
- OSGi Connect specification defines powerful means to connect outer environment with the framework in a controlled way when convenient or necessary

⇨ Embedding the framework and controlling its lifecycle within other containers and various environments is well supported (but still laborious)

Framework life cycle

- A framework instance starts when created, initialized and started by a launcher
- A framework instance shuts down when its system bundle stops

- Hosted bundles have defined activation policies and start level to determine when they shall start

Framework start levels

- ➢ Start levels are changed sequentially and step-by-step from the active start level towards the requested start level
- ➢ Framework events can be used to monitor the start level changes
- ➢ Bundles are started (according to their activation policy) when the active start level reaches their start level
- ➢ Bundles are stopped when the active start level drops below their start level

❢ Do not rely on start levels too much

Configuration support

- Configuration Admin
- Handling events
- Data conversion

Persistence support

- A bundle may get a persistent data area
  - See `Bundle::getDataFile`
  - There is even a specialized specification for user preferences (see PreferencesService specification)
  - Occasionally useful, not intended nor designed well for tasks like storing and managing configuration

- Configuration Admin specification adds support for configuration management
  - Concerns managing and providing configuration dictionaries (sets of key-value pairs) for services
  - However, not part of the core specification

Configuration targets

- **PID** (persistent identity) ≈ a primary key for objects needing a configuration dictionary

  - Configuration targets use PIDs to bind associated configuration dictionaries
  - Configuration Admin provides configuration targets with the configuration dictionaries and notifies them about updates (besides `*ConfigurationListener`)

- `ManagedService`
  - Receives a single configuration dictionary

- `ManagedServiceFactory`
  - May receive any number of configuration dictionaries

Configuration data origin

- 😐 Implementation-specific for a long time
  - Programmatically created configurations are stored… somewhere (depending on the container implementation)
  - Some implementations define locations and data format to be used (e.g., Karaf uses `etc/*.cfg` files for singletons)

- ➢ Configurator specification provides additional and portable support… finally
  - Portable JSON-based configuration format
  - Mechanism for loading initial configurations

`ConfigurationPlugin`

- ❑ Intercepts configuration objects before delivering

- ➢ Allows configuration processing on the fly
  - Decrypting passwords and other sensitive information stored in an encrypted form
  - Replacing placeholders

- ‽ Not working for Declarative Services in past (so some older implementations might get into problems)

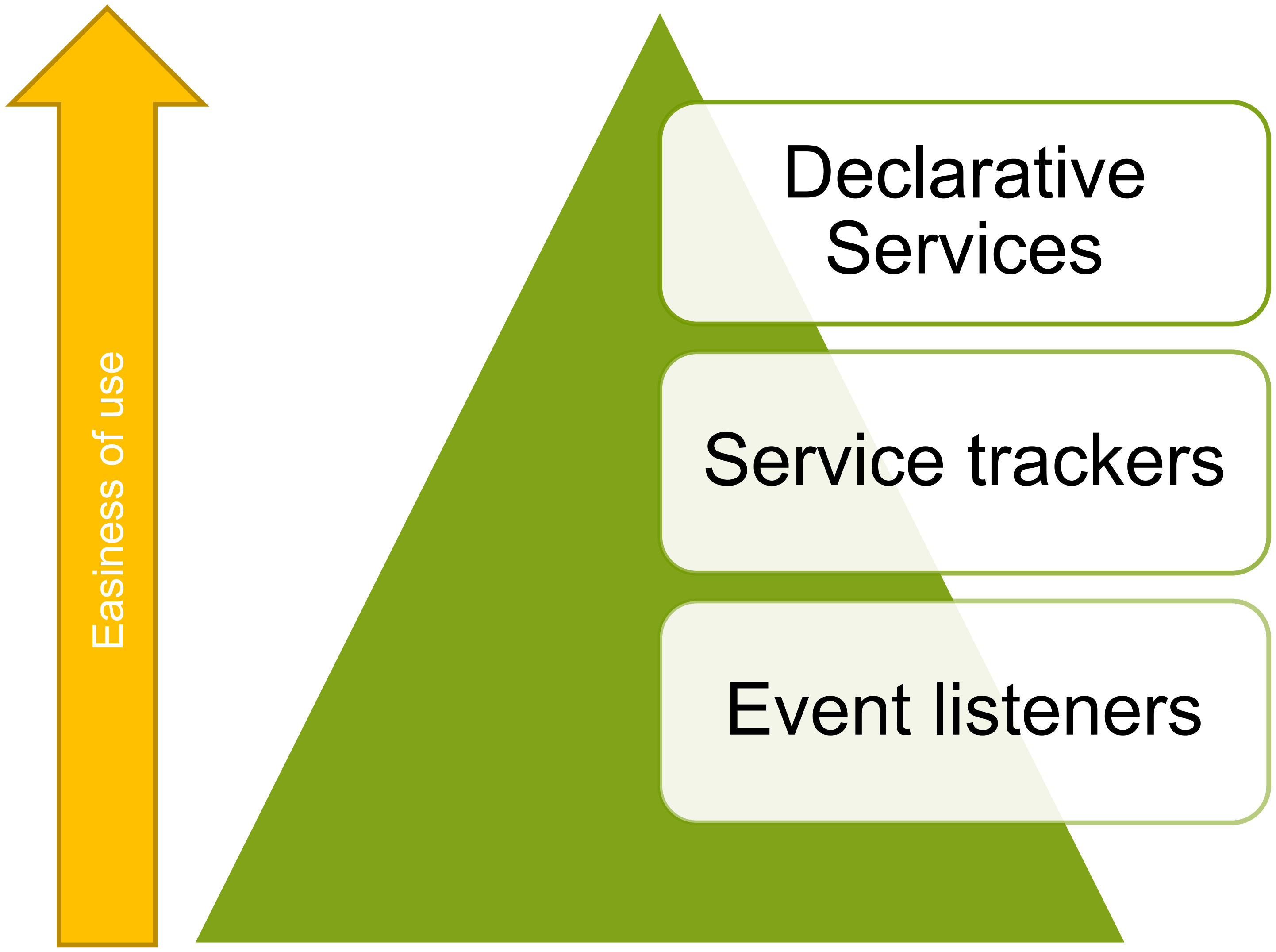
Processing configuration

- Configuration objects provide just key-value pairs
- MetaType Service specification defines means for better configuration processing
  - Very generic, heavy-weight and complex
  - Fortunately, quite usable when combined with Declarative Services
- Converter specification offers a more direct, but a very versatile alternative
  - General-purpose extensible library for converting basically anything to anything
  - Many standard conversions available out of the box, thus useful for processing any configuration data too

Advanced OSGi

- Declarative Services
- Blueprint Container
- CDI Container
- Typical service patterns

Declarative Services (DS)



Declarative Services: about

❑ Declarative model complementing the imperative model of the core API

✓ Less code to write
✓ Footprint-friendly
✓ Lazy & improving startup time
✓ Yet powerful (there is just few occasions when it might be easier using trackers)
✓ Backward-compatible with simpler models

😐 Not the core specification, requiring SCR (Service Component Runtime) support installed

Declarative Services: components

- **Component** ≈ a class with a descriptor declares the life cycle methods and dependencies

- Component features
  - Activation state (enabled × disabled)
  - Activation policy (immediate × delayed)
  - Reference cardinalities and binding policies
  - Configuration dependencies

- Dependencies are injected
  - Using methods, constructor or fields
  - Bindings may change, depending on policies

Declarative Services: details

- SCR uses XML descriptors
  - It is possible to use POJOs without touching the code at all
- Annotations available
  - Comfortable and generally preferred, but contaminate code a bit
  - Class retention policy used though, turned during build with tools into the descriptors

Blueprint Container: about

- Specification based on Spring DM
  - Dependency injection framework
  - XML descriptors defining *beans* and *managers* and binding them together
  - Providing bridges to OSGi service registry (so that Blueprint beans can use other services and vice versa)

- Alternative to Declarative Services?
  - Yes, and no: depends on the situation

Blueprint Container: considerations

- ✓ More familiar to JEE/Spring developers
- ✓ Supports more straight-forward integration
  - Especially with enterprise stuff (e.g., Camel)

- 😐 Hiding more the dynamic nature of OSGi
- ✘ Not “native” for OSGi
- ✘ Specification not further developed
- ✘ XML-based descriptors only, annotations are implementation-specific

CDI integration: about

- Standard for Java dependency injection
  - Probably the most used Java technology
  - Many container/runtime implementations
  - Depending heavily on annotations, evolved from the original JEE XML-based specifications
- CDI integration specification for OSGi created quite recently to support CDI-based code within OSGi

CDI integration: considerations

- ✓ Familiar to all JEE/Spring developers
- ✓ Supports straight-forward integration
- ✓ Can be used for bundle-private services

- 😐 CDI is not as dynamic as OSGi, therefore interaction between CDI and OSGi may have limitations or impose unexpected overhead
- 😐 OSGi-specific annotations needed in some cases to improve the integration

- ✘ Not “native” for OSGi

Extender pattern

- **Extender** ≈ a facility acting on behalf of other bundles and providing additional features

- Common pattern to extend the framework
- The major use case for bundle tracking
- Framework core API provides all required support
  - Surprisingly, nothing special is needed for most cases
  - Advanced core parts (like Weaving Hook Service) may grant additional super-powers to extenders

- Declarative Services, CDI integration or Blueprint Container are notable examples of using the extender model

Whiteboard pattern

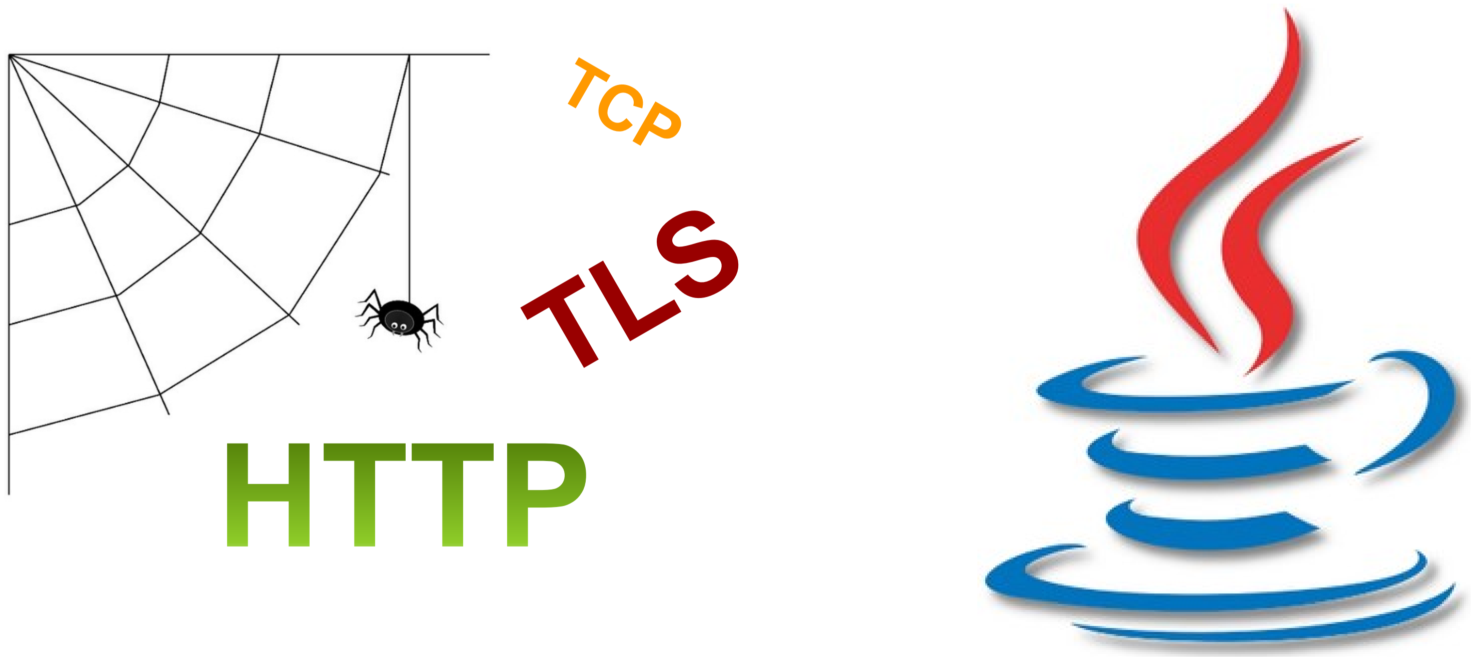
❑ Replacement for listeners
  - Instead of registering listeners and managing their life cycles (which is complex), OSGi service registry is used

▪ Event targets register self as OSGi services
▪ Event sources track the services only

➢ Surprisingly simple
  - Use `ServiceTracker`
  - Use service references of appropriate cardinalities with Declarative Services or reference lists with Blueprint Container

Embedding HTTP services

- HttpService specification
- Whiteboard-based specifications
- Web Applications specification



Http Service and Http Whiteboard

- Http Service
  - Defines the API for registering servlets and resources and mapping HTTP contexts for them
  - ✘ Very low-level
  - ✘ Very limited

- Http Whiteboard
  - Uses whiteboard pattern to register servlets, filters and resources
  - Service properties (and/or annotations) define the mapping etc.
  - 😐 More powerful, easier to use, but the specification is more complex as well

JAX-RS Whiteboard

- Uses whiteboard pattern to register JAX-RS applications and resources
- Resembles Http Whiteboard, just aiming to JAX-RS

✓ Thanks to the nature of JAX-RS, more compact and simpler than Http Whiteboard

😐 Not very simple though, many degrees of freedom, e.g., multiple whiteboards

😐 May bring some surprises (e.g., different default life cycle of JAX-RS resources)

😐 There might be yet gaps and unsettled matters

Web Applications

- Defines Web Application Bundle (WAB) and transformation from WAR

✓ Allows to use basically unchanged WARs

✓ Allows to use the full power of OSGi in WABs

✓ Quite well supported and arguably best usable for regular Web applications

Conclusions

What we have learnt

- Concepts of proper modularity
- Versioning and compatibility concerns
- Service-oriented design
- OSGi core basics
- OSGi services in a more comfortable way
- HTTP support overview

You might hear about OSGi

- It is too complex/complicated
- It has a steep learning curve
- It has no documentation
- It does not work with…
- The tooling sucks
- You don't need it

Debunked?

So… shall we use OSGi?

Not an easy decision…

- ➢ Recall the beginning: why OSGi?
- ➢ Read more details on OSGi Alliance site: https://www.osgi.org/developer/benefits-of-using-osgi/ (use Wayback Machine to get it)

- ➢ TL;DR? Alright, then suitable for:
  - Complex and/or long-living projects
  - Dynamic environments
  - Flexible requirements

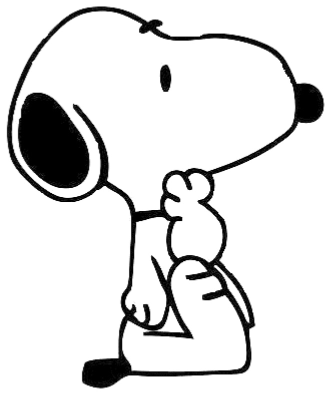
Still hesitating to use OSGi?

Maybe your case is different…

- OSGi might not be suitable for:
  - Small and disposable projects (especially when not familiar with OSGi yet)
  - Environments, technologies or libraries that might be too difficult to integrate into OSGi
  - Some non-technical requirements excluding OSGi (e.g., requiring a particular platform or technology)

It was quite a lot of things…

Any remaining questions?



References

- Java VM & language specification
  https://docs.oracle.com/javase/specs/
- OSGi Alliance
  http://www.osgi.org/
- OSGi Documentation
  https://docs.osgi.org/
- OSGi Semantic Versioning Technical Whitepaper
  https://docs.osgi.org/whitepaper/semantic-versioning/
- Semantic Versioning
  https://semver.org/

Good bye!

And happy coding