Learning to Ignore OSGi

January 25, 2011

Richard S. Hall

Modularity

- This presentation is ***not*** about modularity
- It assumes we all know what modularity is and agree it is ***a good thing***
    - If you feel otherwise please leave or stop reading
- This presentation is about using OSGi as a means to achieve modularity

The reality

The reality

- I asked a developer on a 250-plus bundle OSGi project, “*How much have you read about OSGi?*” The answer?

The reality

- I asked a developer on a 250-plus bundle OSGi project, “*How much have you read about OSGi?*” The answer?
  - “*I've **never** read any OSGi documentation at all.*”

The reality

- I asked a developer on a 250-plus bundle OSGi project, “*How much have you read about OSGi?*” The answer?
  - “*I've* ***never*** *read any OSGi documentation at all.*”
- Clearly, this can't be the approach of the average corporate developer, can it?

The reality

- I asked a developer on a 250-plus bundle OSGi project, “*How much have you read about OSGi?*” The answer?
  - “*I've **never** read any OSGi documentation at all.*”
- Clearly, this can't be the approach of the average corporate developer, can it?
  - Shortly after the above, I read the following on an OSGi-oriented mailing list:
    “*I represent the mainstream corporate developer who only wants to consume OSGi **but not understand it**.*”

Reality check

- We have people who think they can use a technology in projects (or even base projects on it) with ***little or no*** understanding of it

Reality check

- We have people who think they can use a technology in projects (or even base projects on it) with ***little or no*** understanding of it
  - *This seems like it has a debatable value proposition, but...*

Reality check

- We have people who think they can use a technology in projects (or even base projects on it) with ***little or no*** understanding of it
  - *This seems like it has a debatable value proposition, but...*
- Ok, fine, this presentation will help you ***learn to ignore*** OSGi...

The first step

The first step in learning to ignore OSGi is...

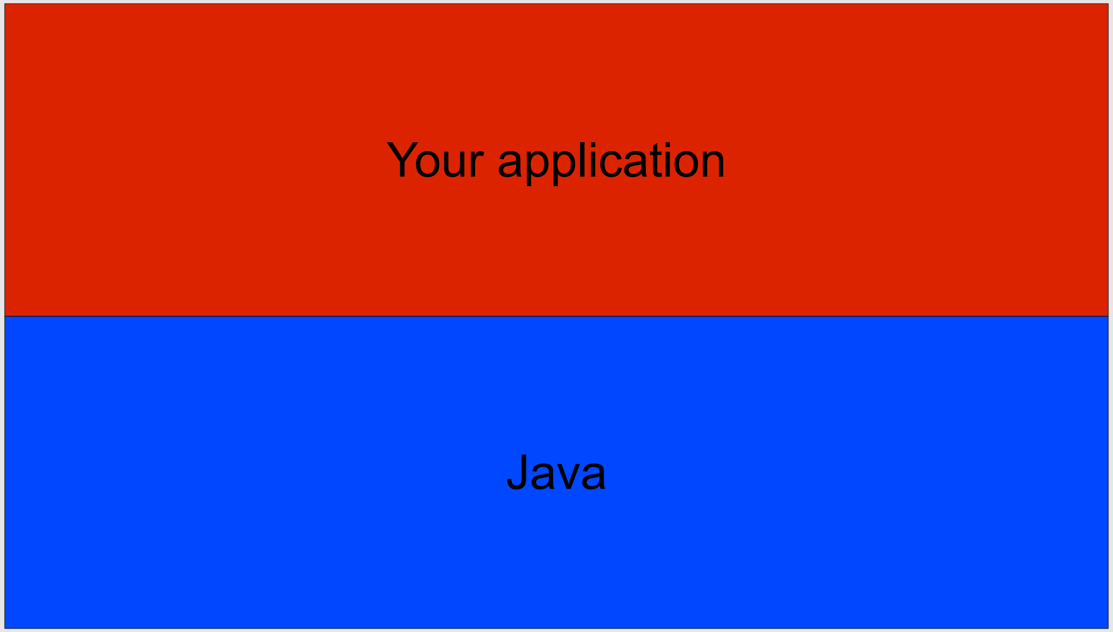
The first step

The first step in learning to ignore OSGi is...

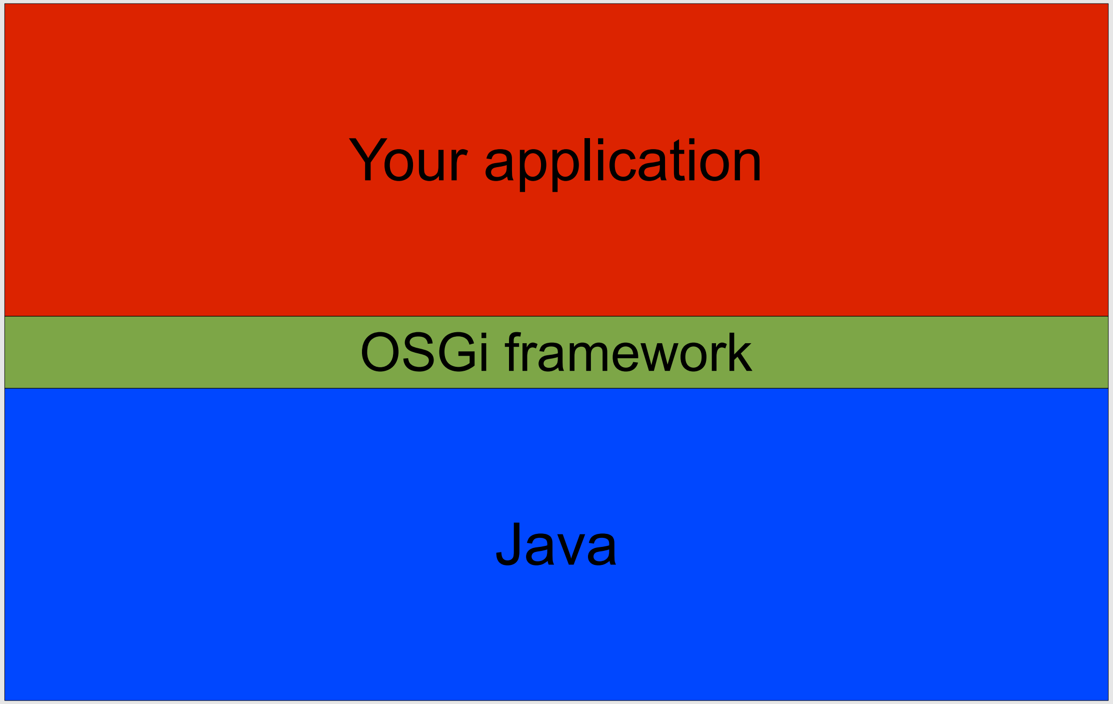
Accept the fact that you ***can't*** completely ignore OSGi!

Why?

Why?



Why?



OSGi adds a layer to enforce modularity
by ***limiting*** type visibility

Say, “What?”

To clarify, let's review how type visibility is handled in standard Java...

Say, “What?”

Class path

rt.jar
bar.jar
baz.jar
foo.jar
…
zoo.jar

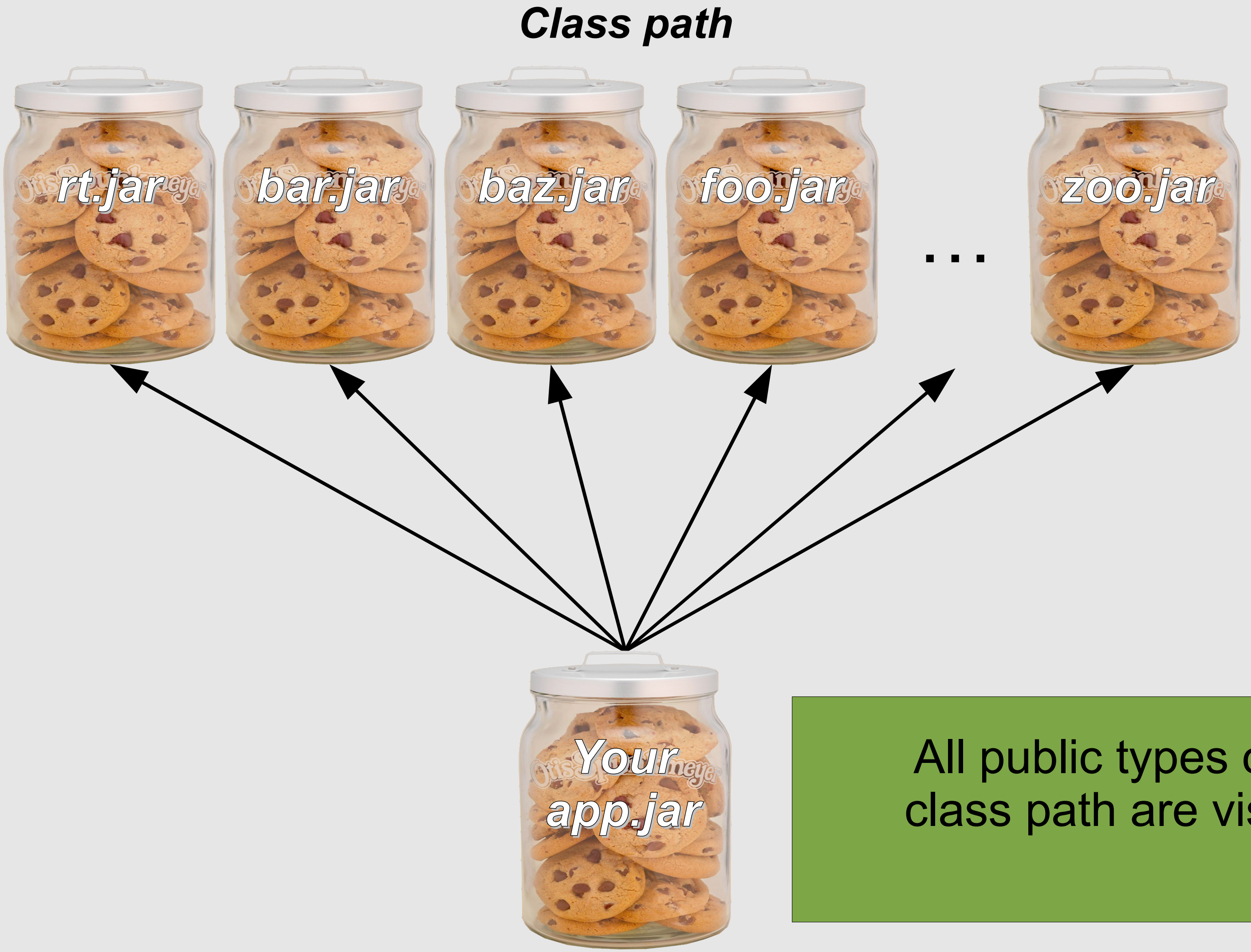
Say, “What?”

Class path

rt.jar
bar.jar
baz.jar
foo.jar
…
zoo.jar

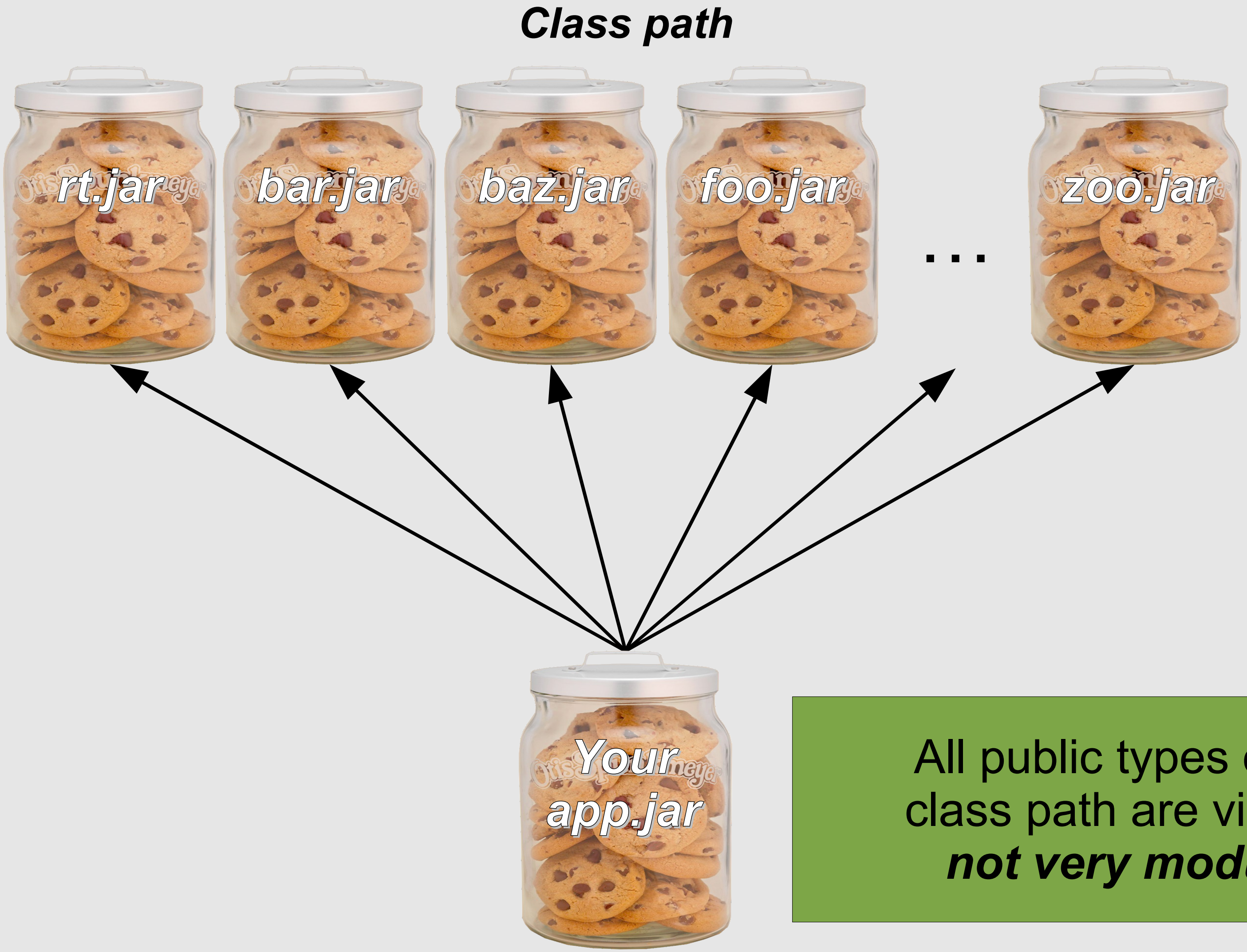
Besides its own types, which types are visible to your application?

Say, “What?”



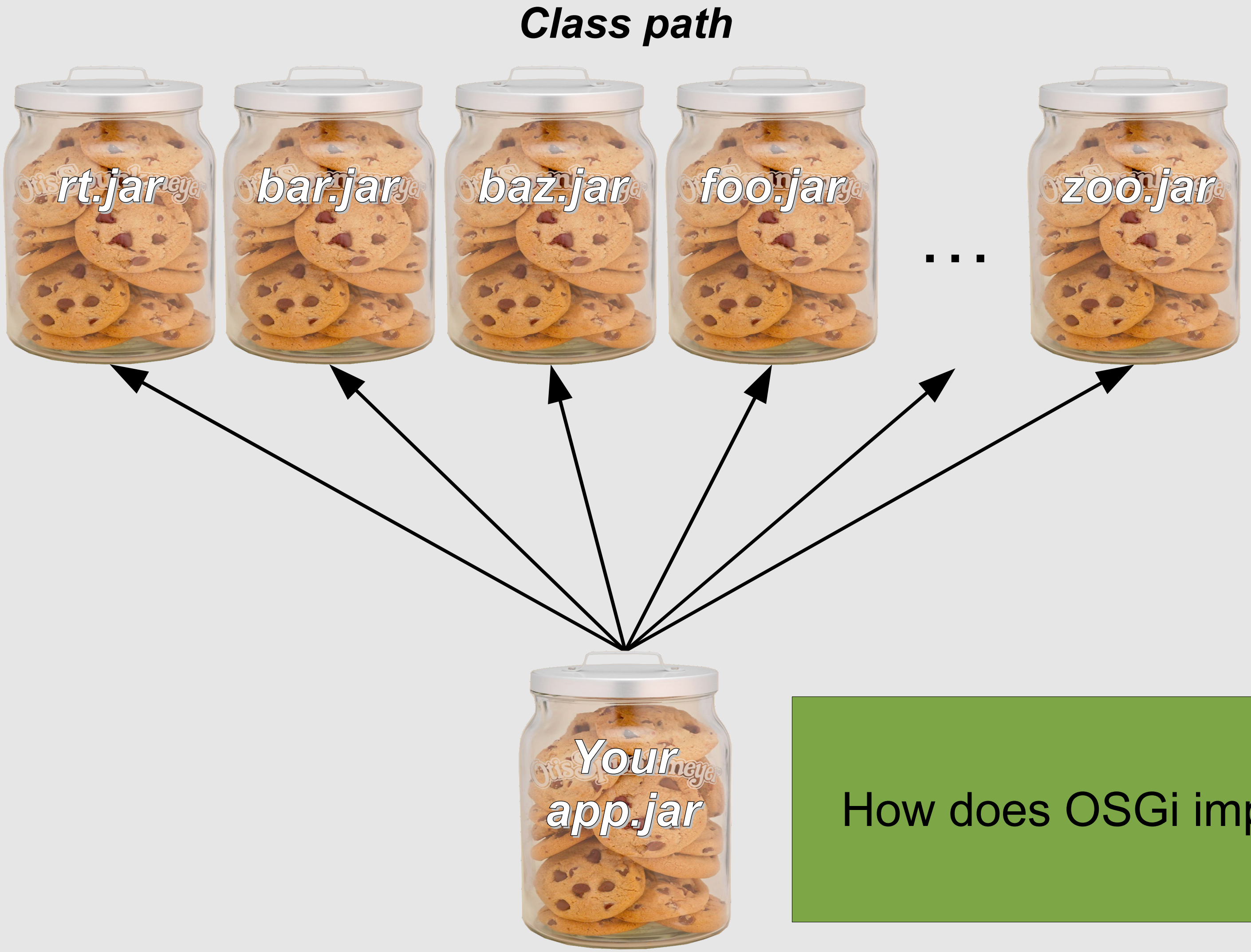
All public types on the class path are visible...

Say, “What?”



All public types on the class path are visible... ***not very modular***.

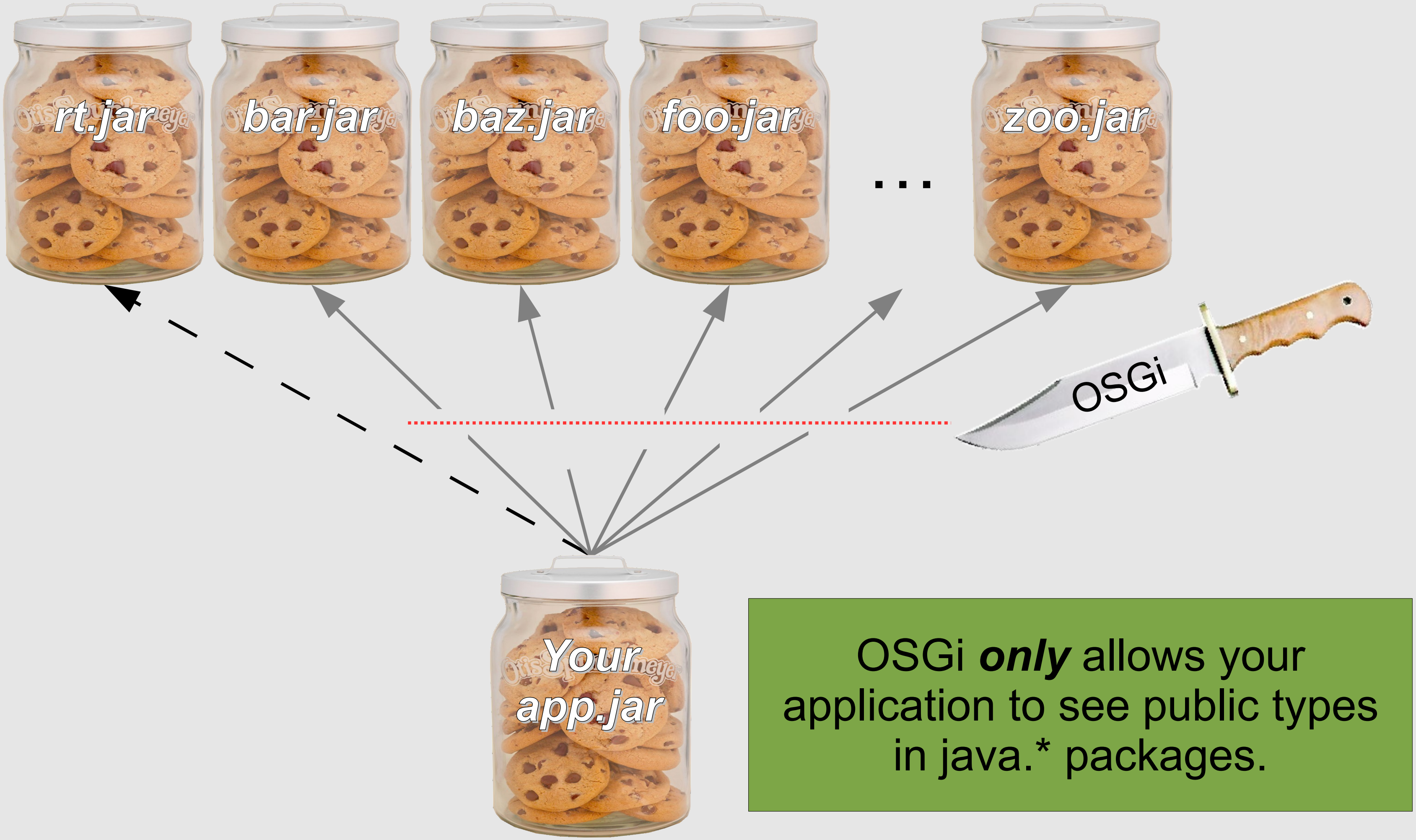
Say, “What?”



How does OSGi impact this?

Say, “What?”

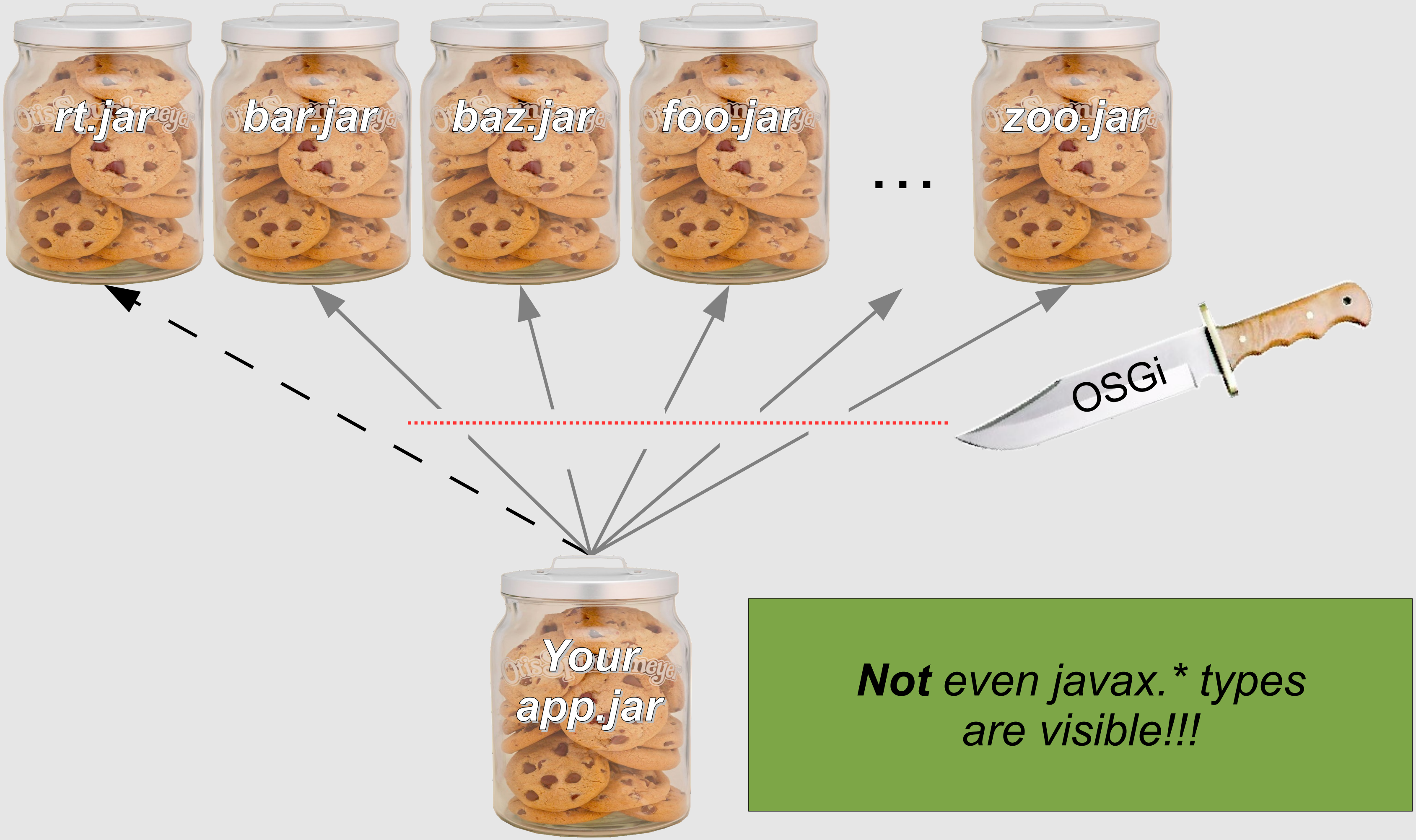
Class path



OSGi ***only*** allows your application to see public types in java.* packages.

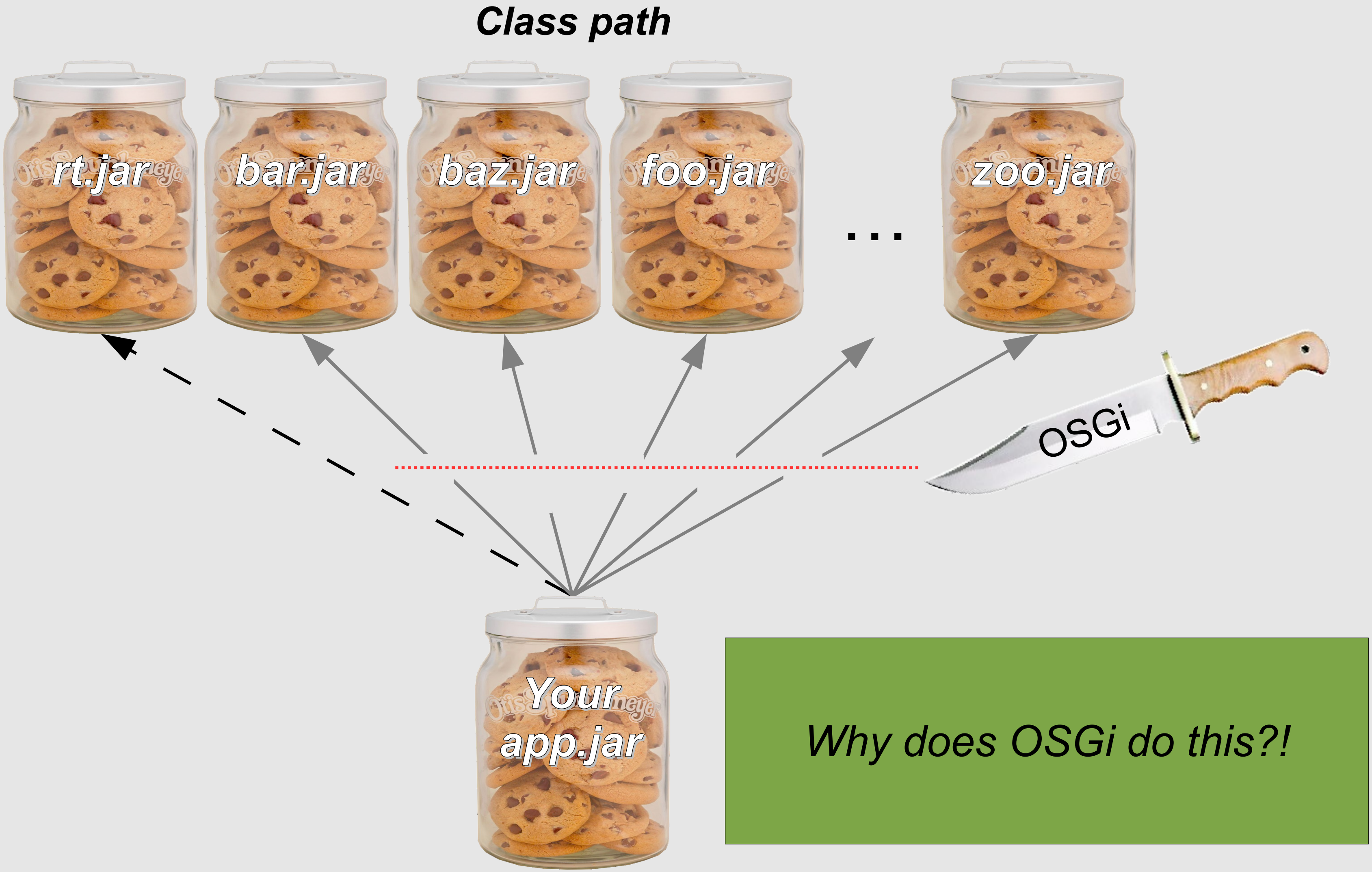
Say, “What?”

Class path



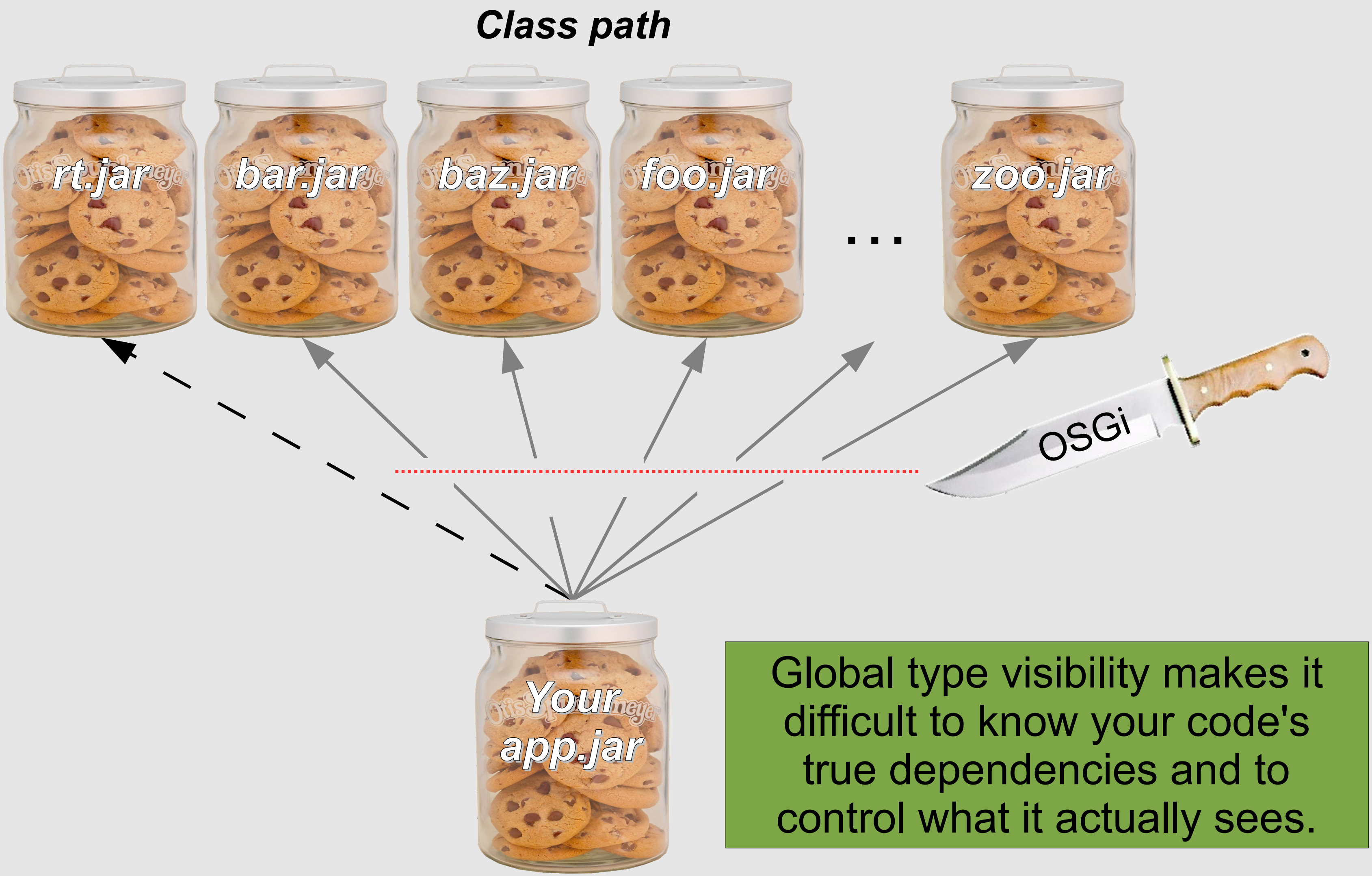
***Not** even javax.* types are visible!!!*

Say, “What?”



Why does OSGi do this?!

Say, “What?”



Global type visibility makes it difficult to know your code's true dependencies and to control what it actually sees.

What about legacy code?

What about legacy code?

What about legacy code?

Ok, not really, but sort of... there is ***no magic*** *OSGi pixie dust!*

What about legacy code?

- The MuleSoft fallacy
  - http://blogs.mulesoft.org/osgi-no-thanks/
  - To paraphrase (not a quote):
    - "OSGi provides little value and is too complex as demonstrated by our failed attempt to make modularity invisible when porting our huge legacy system to it with over 150 third-party JARs."
- There is no free lunch
  - Modularity has to be considered at all levels and ***will be visible***
- Porting huge legacy systems to another platform is complex. Period.

What about legacy code?

- Legacy code is written under a different mental model that no longer works in OSGi
  - **`@deprecated`** ***global public type visibility***
- Legacy code must be examined on a case-by-case basis
  - Does the code just provide types?
  - Does it make assumptions about type visibility? (i.e., use class loaders or `Class.forName()`)
    - If so, it likely won't work

That's not all!

- Currently, we've only discussed which types your application can see
- What about the flip side – which types from your application can be seen by other code?

That's not all!

If your public classes are these cookies

That's not all!

If your public classes are these cookies then everyone can see all your cookies in a standard JAR file...

That's not all!

If your public classes are these cookies then everyone can see all your cookies in a standard JAR file... again, ***not very modular***.

That's not all!

If your public classes are these cookies then everyone can see all your cookies in a standard JAR file... again, ***not very modular***.

How does OSGi impact this?

That's not all!

If your public classes are these cookies then everyone can see all your cookies in a standard JAR file... again, ***not very modular***.

In OSGi, no one sees any of your cookies.
Nothing!

That's not all!

If your public classes are these cookies then everyone can see all your cookies in a standard JAR file... again, ***not very modular***.

Why does OSGi do this?!

That's not all!

If your public classes are these cookies then everyone can see all your cookies in a standard JAR file... again, ***not very modular***.

Because it is ***impossible*** to protect your code's implementation details if you always expose everything.

JAR file comparison summary

	Standard JAR file	**OSGi JAR file**
Class path type visibility for internal code	*All public types*	*Only public java.* types*
Application type visibility for external code	*All public types*	*Nothing*

JAR file comparison summary

	Standard JAR file	**OSGi JAR file**
Class path type visibility for internal code	*All public types*	*Only public java.* types*
Application type visibility for external code	*All public types*	*Nothing*

Do these differences seem minor enough to ignore?

Wait a minute!

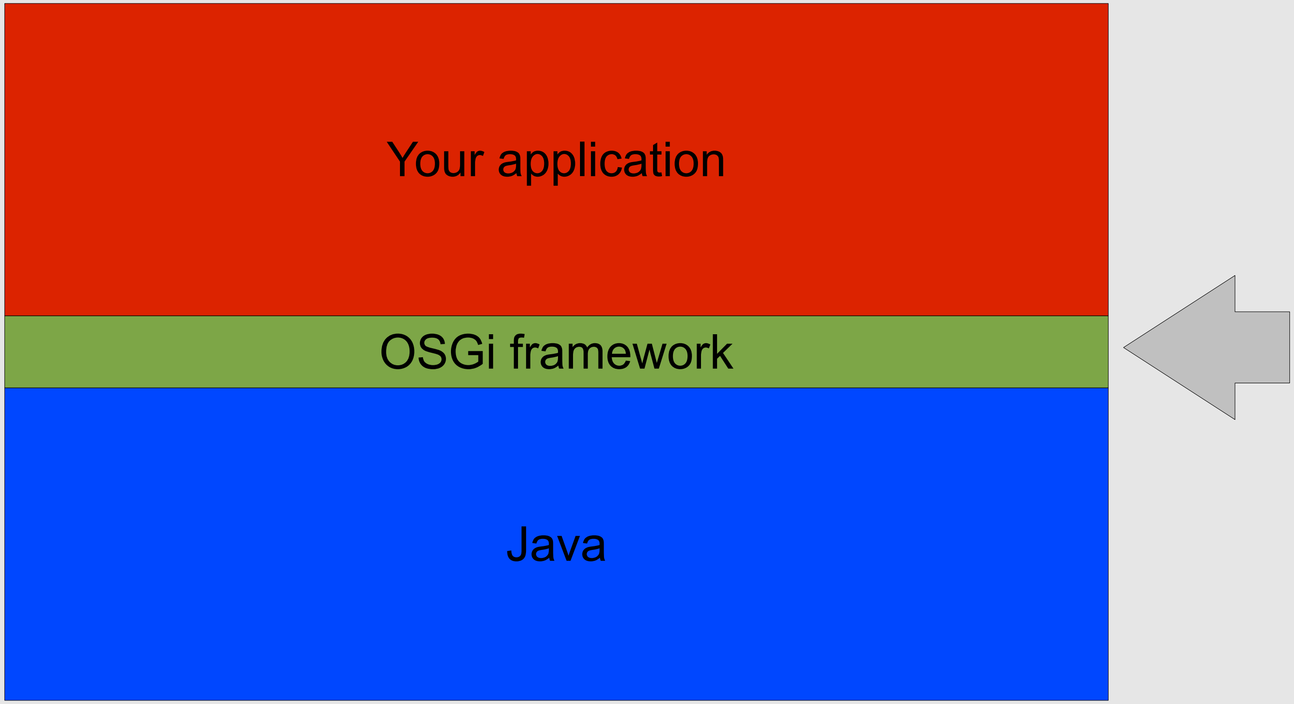
- You must be thinking
  - “*What a gyp!*”
  - “*I thought we were going to learn how to ignore OSGi?*”

Wait a minute!

- You must be thinking
  - "*What a gyp!*"
  - "*I thought we were going to learn how to ignore OSGi?*"

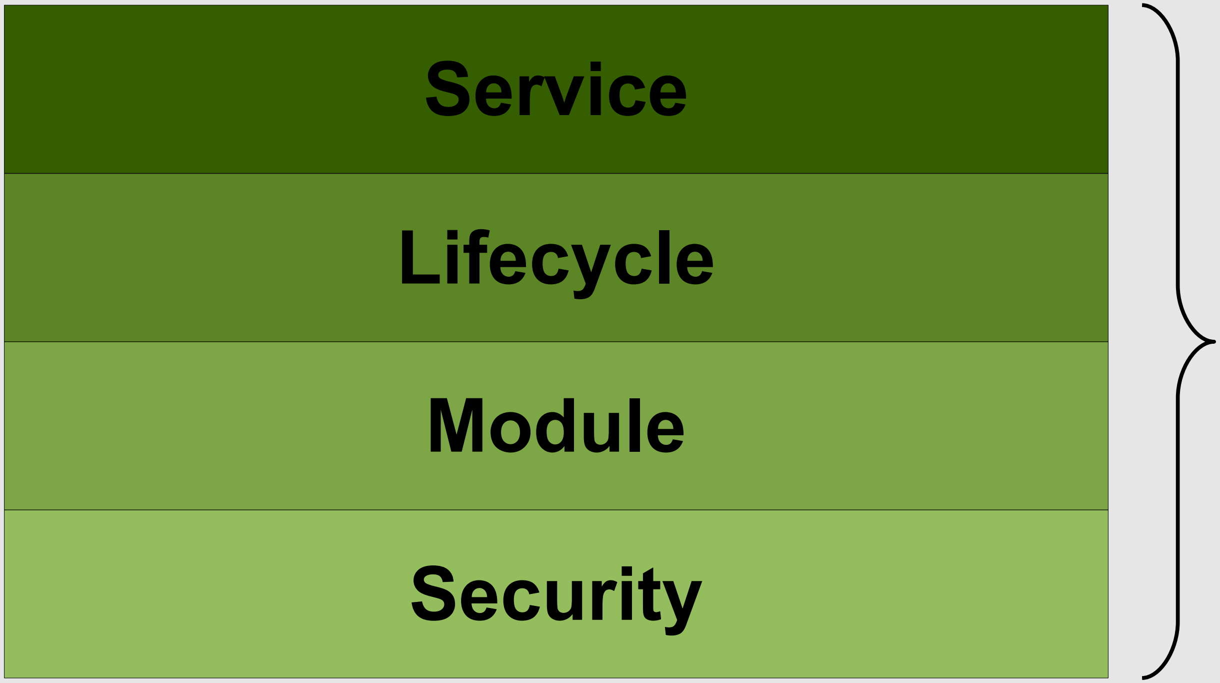
- ***There's a lot you can ignore, but type visibility isn't one of them...***
  - However, if you change your mental model to operate under these new rules, you'll no longer have to think about them
    - And your JAR files will ***still*** work as standard JAR files

A lot left to ignore



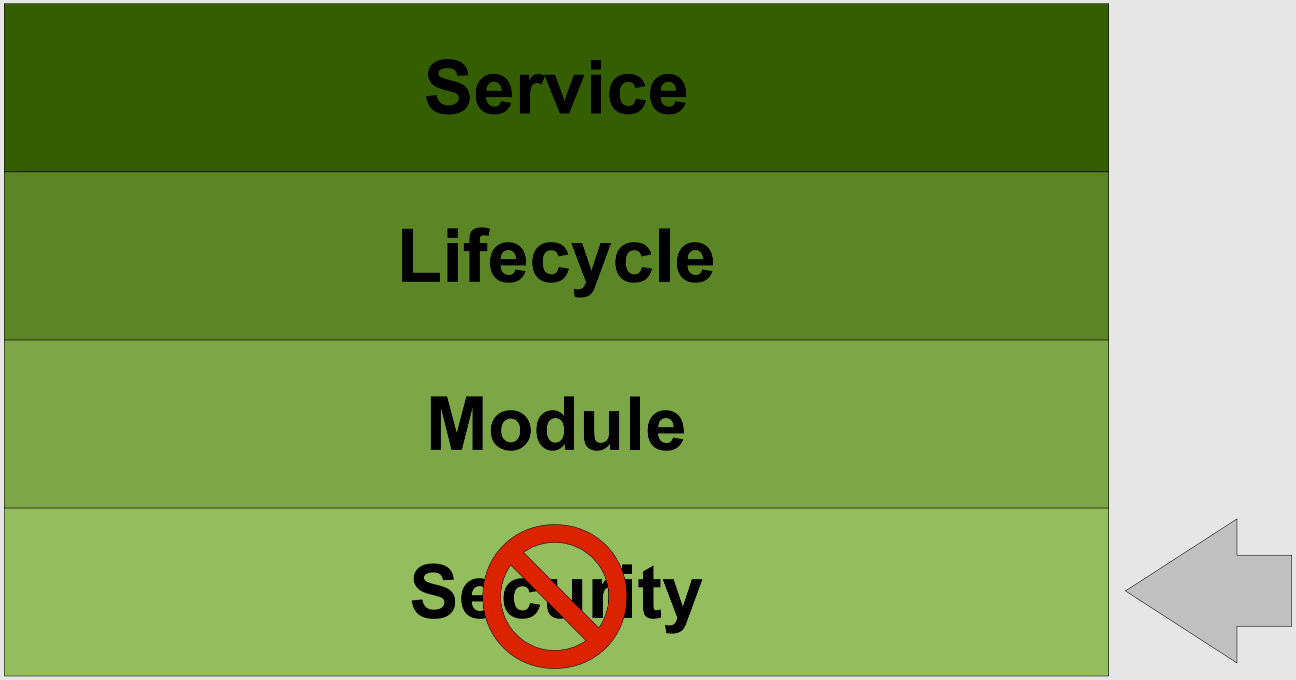
Expand focus

A lot left to ignore



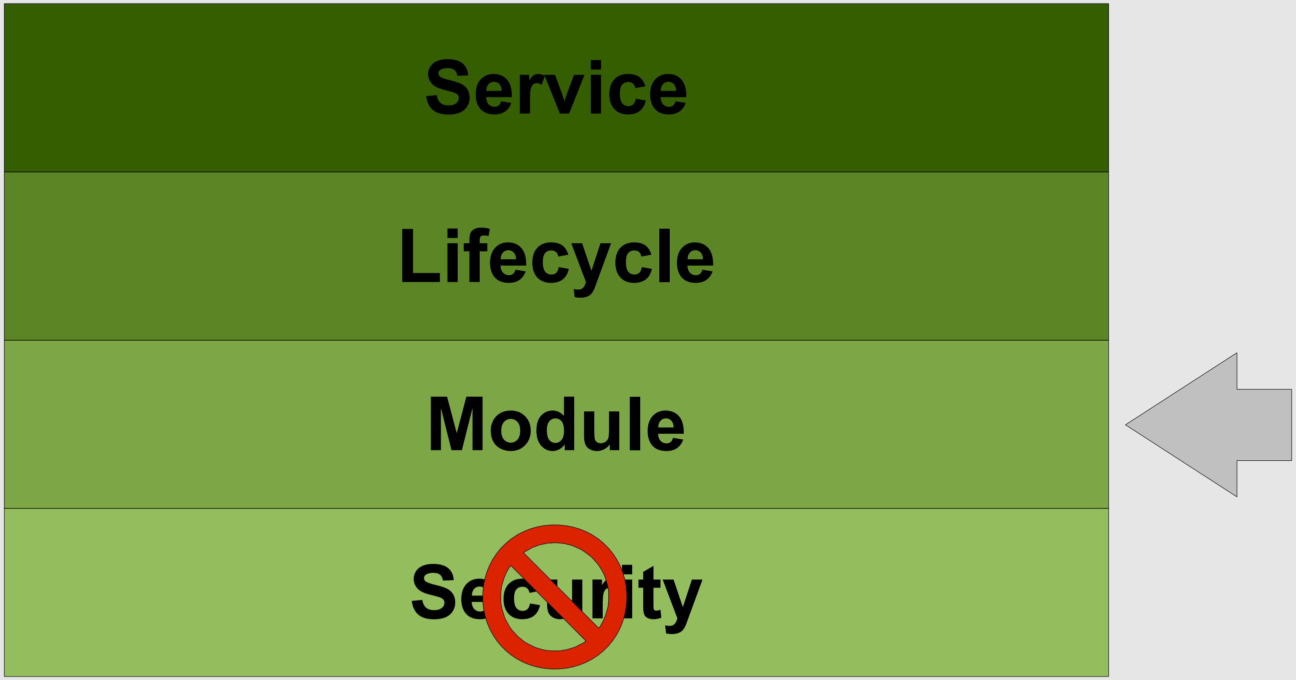
OSGi framework is conceptually layered

A lot left to ignore



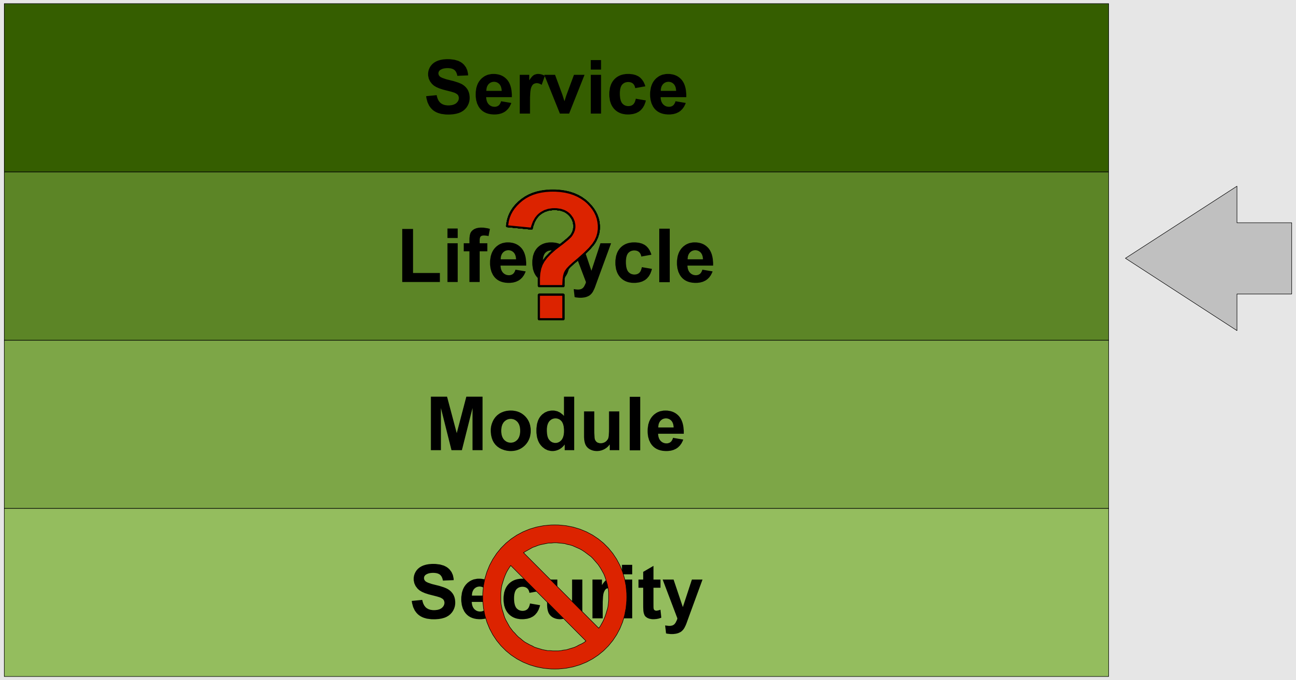
You can ignore security...we always do

A lot left to ignore



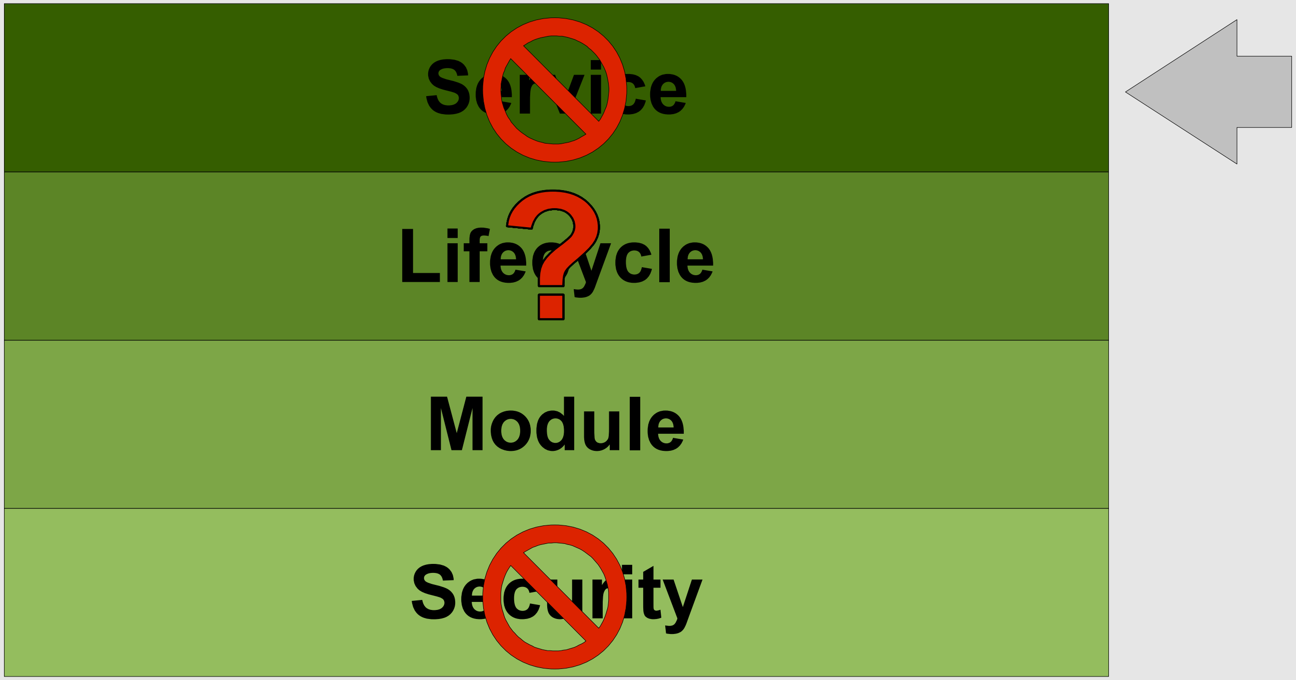
Handles type visibility, so you can't ignore this

A lot left to ignore



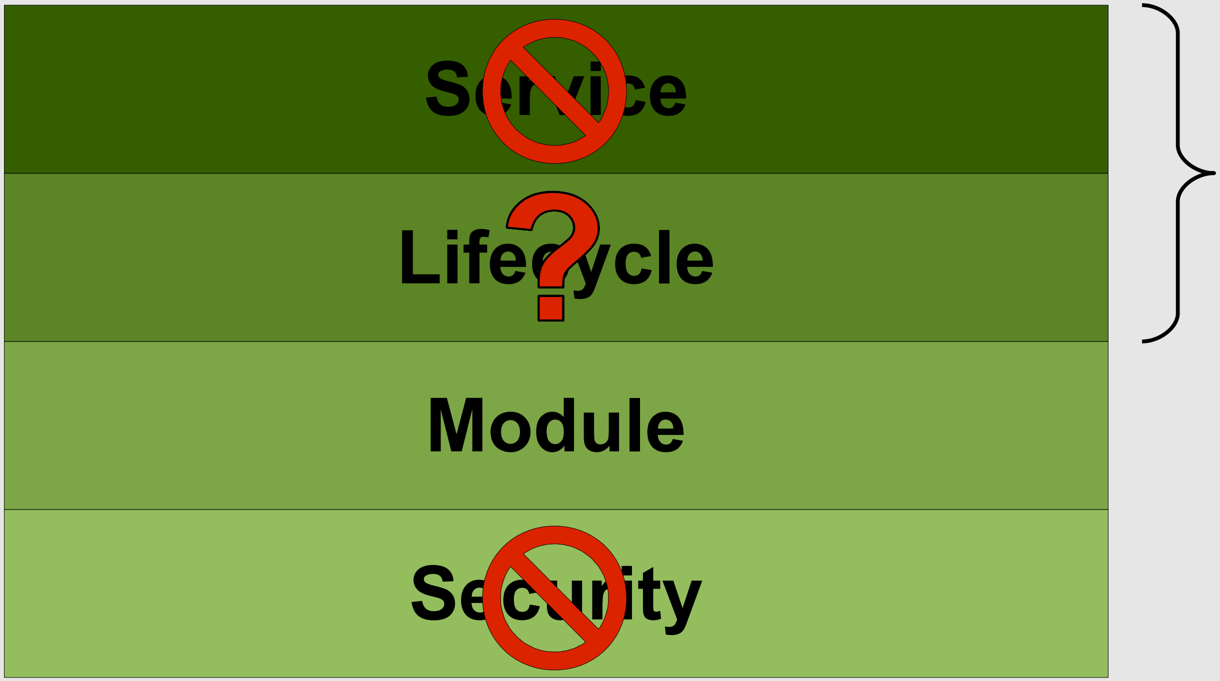
Maybe you can ignore this

A lot left to ignore



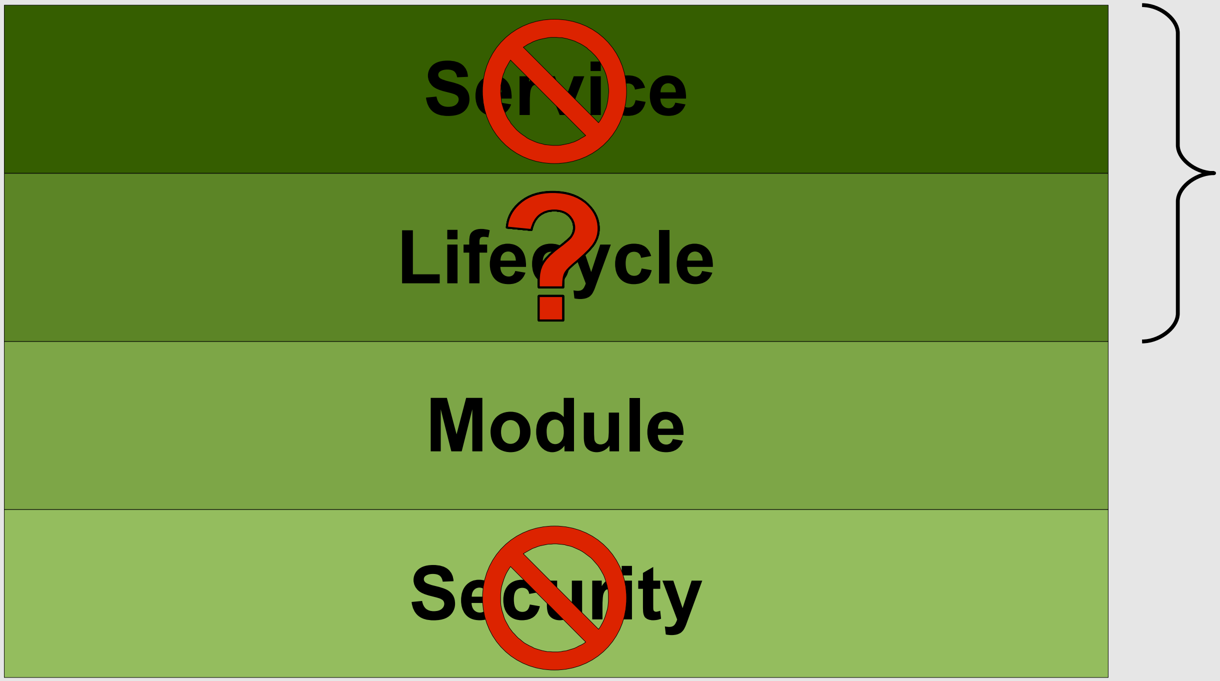
You can ignore this, although you lose a decoupling mechanism

A lot left to ignore



Basically, all OSGi API is from these two layers, so we can pretty much ignore it all

A lot left to ignore



Service

Lifecycle

Module

Security

Ironically, these are the thinnest layers

What about lifecycle?

What about lifecycle?

Another JAR file difference...

	Standard JAR file	**OSGi JAR file**
Class path type visibility for internal code	*All public types*	*Only public java.* types*
Application type visibility for external code	*All public types*	*Nothing*
Lifetime of JAR file	*Same as JVM*	*Can come and go*

What about lifecycle?

- You can ignore lifecycle if your code doesn't do anything that may live on after it
  - i.e., have things that need to be cleaned up
    - Such as active threads, open files, open ports, etc.

What about lifecycle?

- You can ignore lifecycle if your code doesn't do anything that may live on after it
  - i.e., have things that need to be cleaned up
    - Such as active threads, open files, open ports, etc.
- If you do have such issues, then...
  - Your code has explicit lifecycle requirements and must implement a “bundle activator”
    - i.e., provide “start” and “stop” callbacks
  - Your code must not create or use long-lived resources unless it has been started and not after it has been stopped

What about lifecycle?

- If you're concerned about not being able to ignore the OSGi lifecycle API...
- Dirty little secret...
    - You don't need to use the OSGi lifecycle API
    - It's possible to create your own lifecycle layer
        - And ultimately your own service-like layer
    - However, the same sort of rules ultimately still apply, they'll just be enforced by you
        - And even then, your lifecycle layer will still need to be implemented using OSGi API

Revisiting legacy code

- For legacy code, the two biggest obstacles when moving to OSGi are assumptions about
  - Global type visibility
  - Static lifecycle

End of story?

End of story?

No!
A JAR file that can't see anything and no one can see into isn't very useful!

Sharing cookies

If you have some types you want to share with other code, you need some way to expose them...

Sharing cookies

...OSGi allows you to ***export*** all public types in a Java package.

Sharing cookies

This gives ***you control*** over your code's implementation details, since you only expose what you want to external code.

Sharing cookies

- You must explicitly list all packages you wish to share in your JAR manifest
  - `Export-Package: org.foo.p1, org.foo.p2`

```
Export-Package:
 org.foo.p1,
 org.foo.p2
```

Sharing cookies

- You must explicitly list all packages you wish to share in your JAR manifest

  -
    ```
    Export-Package:
     org.foo.p1; version=1.0,
     org.foo.p2; version=1.1
    ```
  - You should actually specify package versions

Sharing cookies

- You must explicitly list all packages you wish to share in your JAR manifest

  -
    ```
    Export-Package:
     org.foo.p1; version=1.0,
     org.foo.p2; version=1.1
    ```
  - You should actually specify package versions
- Only the types in these listed packages are shared
  - You should keep this list short
  - Unlisted packages are hidden implementation details

Sharing cookies

- You must explicitly list all packages you wish to share in your JAR manifest

  -
    ```
    Export-Package:
     org.foo.p1; version=1.0,
     org.foo.p2; version=1.1
    ```
  - You should actually specify package versions

Since tools can help generate this syntax, you can potentially ignore it... but it is probably better to understand it for debugging purposes.

-
  - You should keep this list short
  - Unlisted packages are hidden implementation details

Gimme your cookies

By default, your code only sees types in java.* packages, so you'll almost certainly need some way to ask for more...

Gimme your cookies

...OSGi allows you to ***import*** required types in other Java packages not contained in your JAR file.

Gimme your cookies

This gives ***you control*** over what external types your code sees at execution time.

Gimme your cookies

- You must explicitly list all required external packages (except java.* packages) in your JAR manifest

  -
    ```
    Import-Package:
     org.foo.p1,
     org.foo.p2
    ```

Gimme your cookies

- You must explicitly list all required external packages (except java.* packages) in your JAR manifest

  -
    ```
    Import-Package:
     org.foo.p1; version=”[1.0,2.0)”,
     org.foo.p2; version=”[1.1,2.0)”
    ```
  - With meaningful version ranges

Gimme your cookies

- You must explicitly list all required external packages (except java.* packages) in your JAR manifest
  - `Import-Package: org.foo.p1; version=”[1.0,2.0)”, org.foo.p2; version=”[1.1,2.0)”`
  - With meaningful version ranges
- Only the external types in these listed packages are visible internally, in addition to internal and java.* types

Gimme your cookies

- You must explicitly list all required external packages (except java.* packages) in your JAR manifest

  -
    ```
    Import-Package:
     org.foo.p1; version="[1.0,2.0)",
     org.foo.p2; version="[1.1,2.0)"
    ```
  - With meaningful version ranges
- On
  list
  int d
  java.* types

Tools can again help here and generate much of this using byte-code analysis, but you'll still need to review it.

JAR + metadata != module

- Once you've added export and import metadata to your JAR files, you basically have a ***module***
  - Albeit, maybe not a very meaningful one

JAR + metadata != module

- Once you've added export and import metadata to your JAR files, you basically have a ***module***
  - Albeit, maybe not a very meaningful one
- Modules are not stalagmites, they don't just form, they are a ***design primitive***
  - Just like classes
  - You need to think hard about
    - What you put into a module
    - What you expose from a module
    - What you expose to a module

JAR + metadata != module

- Once you've added export and import metadata to your JAR files, you basically have a ***module***
  - Albeit, maybe not a very meaningful one
- Modules are not stalagmites, they don't just form, they are a ***design primitive***
  - Just like classes
  - You need to think hard about
    - What you put into a module
    - What you expose from a module
    - What you expose to a module

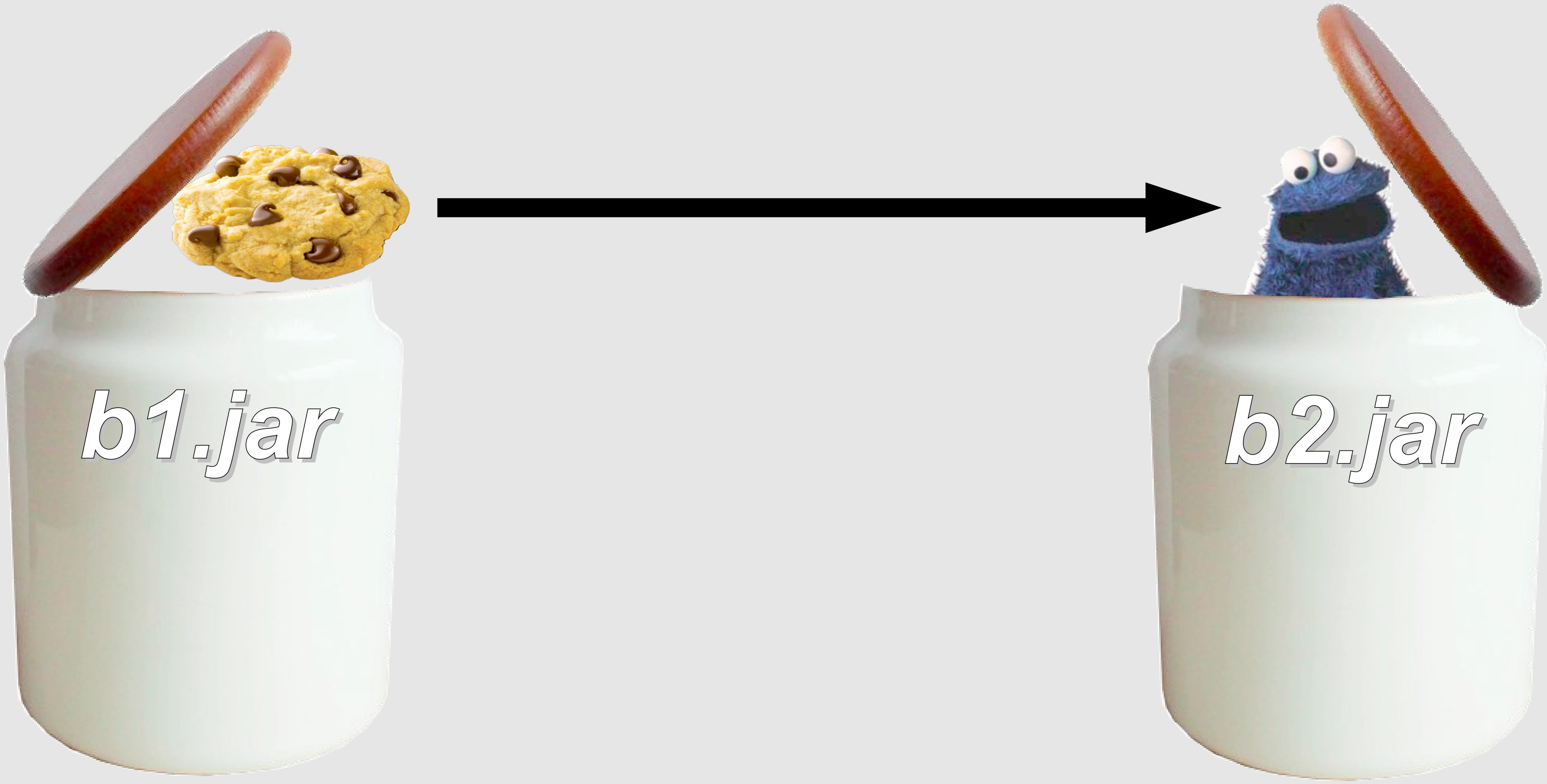
Maximize cohesion, minimize coupling!

OSGi at execution time

- Even if we ignore everything else, once we have some modules, they still need to run in an OSGi framework
  - This is easily accomplished with most OSGi frameworks
  - But what is actually happening?

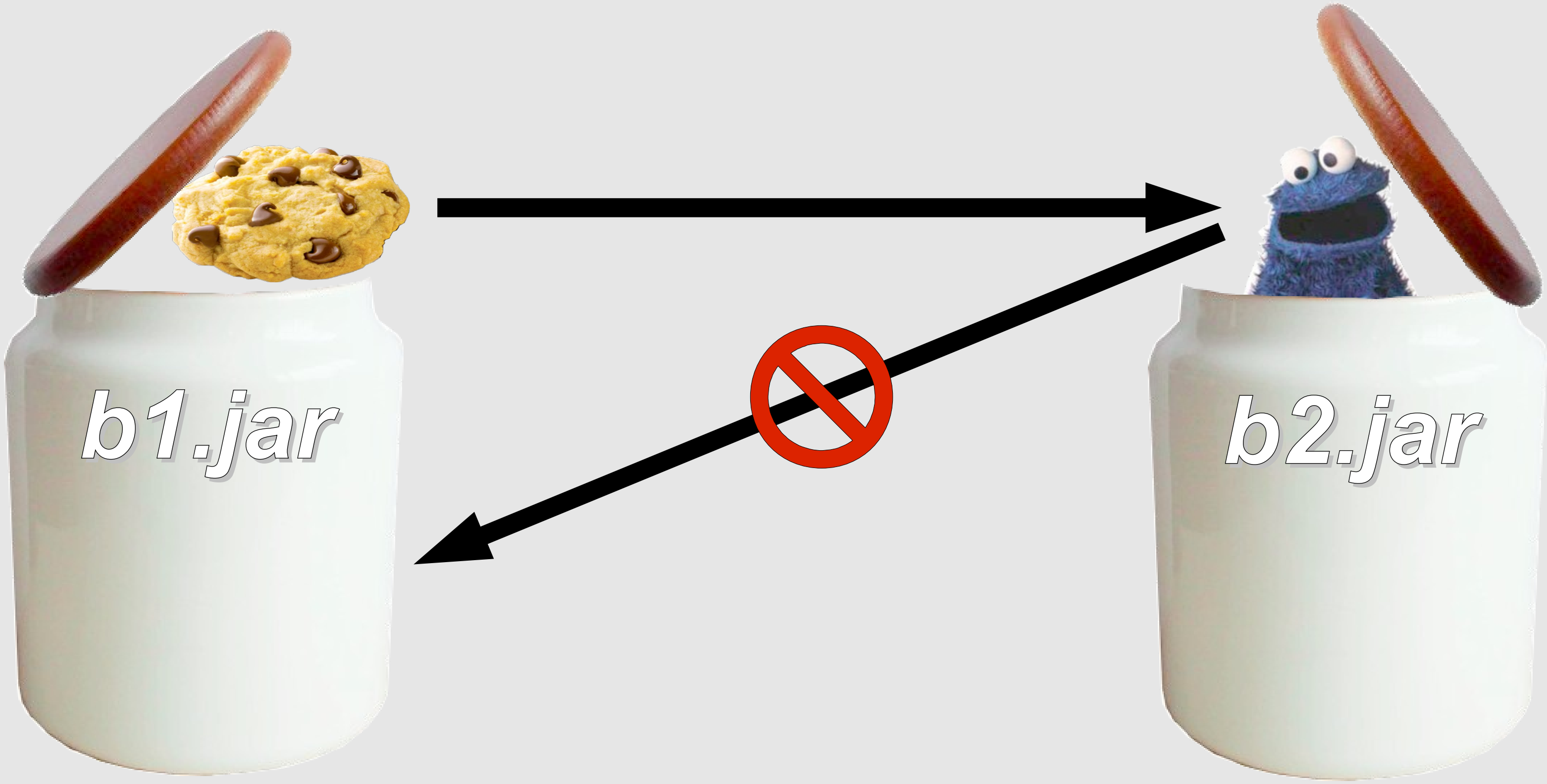
OSGi at execution time

- The framework ***resolves*** module dependencies
  - Resolving dependencies involves matching exported packages to imported packages to ensure type consistency
  - A module can't be used if its dependencies aren't satisfied



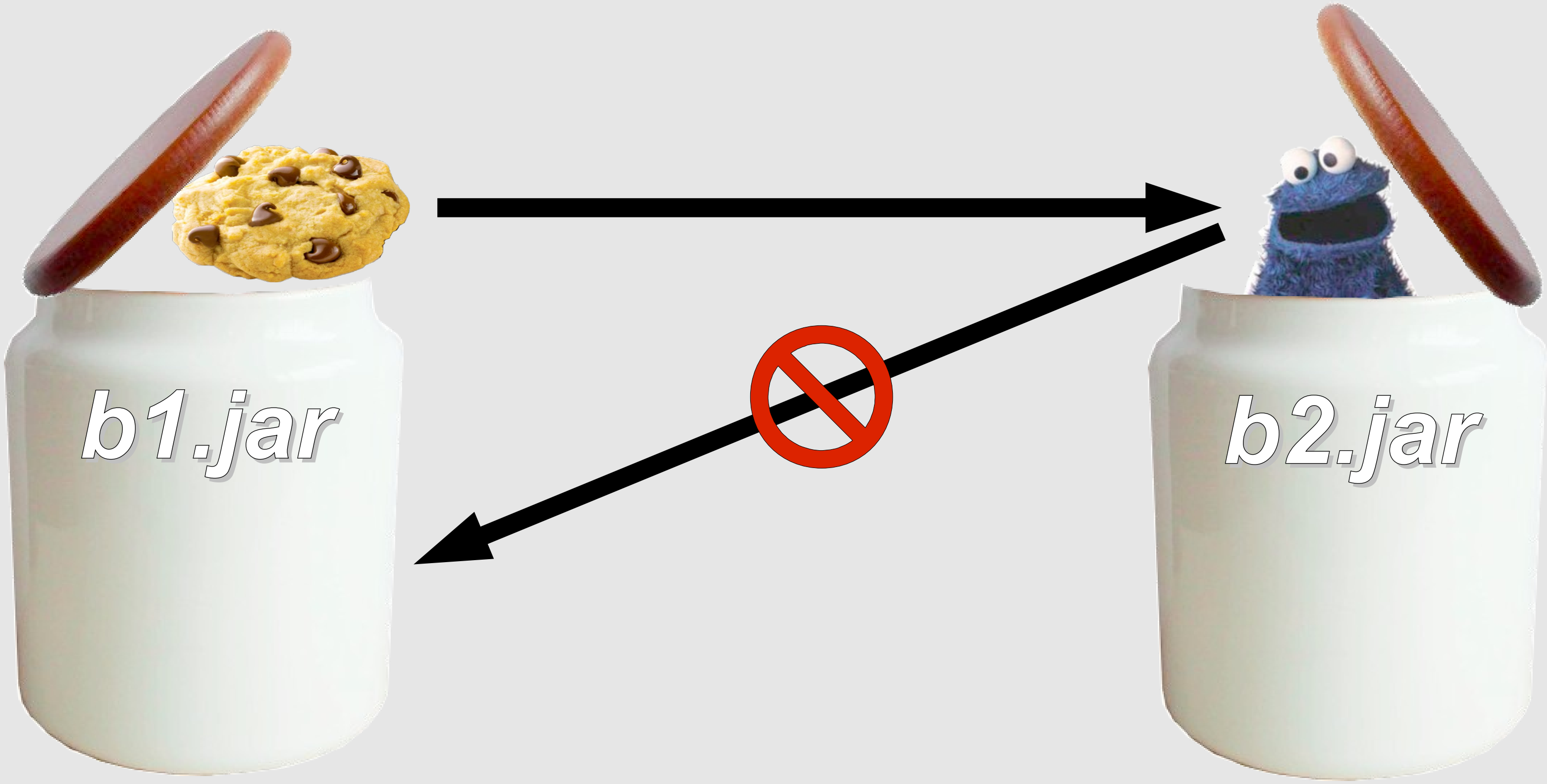
OSGi at execution time

- The framework ***enforces*** module boundaries
  - Ensuring that only exported packages are exposed and only imported packages are visible
  - Each module gets a class loader to enforce isolation



OSGi at execution time

- After dependency resolution, OSGi gets out of the way
  - It's just class loader delegation and application code execution after that



Understanding search order

- OSGi class loading search order is strict and consistent, at a high level it is as follows

Understanding search order

- OSGi class loading search order is strict and consistent, at a high level it is as follows
  - Boot delegate java.* packages, ***fail if not found***

Understanding search order

- OSGi class loading search order is strict and consistent, at a high level it is as follows
  - Boot delegate java.* packages, ***fail if not found***
  - Delegate imported packages to exporter class loaders, ***fail if not found***

Understanding search order

- OSGi class loading search order is strict and consistent, at a high level it is as follows
  - Boot delegate java.* packages, ***fail if not found***
  - Delegate imported packages to exporter class loaders, ***fail if not found***
  - Search internal content, ***fail if not found***

Understanding search order

- Taking into account all OSGi features, it's a little more complicated, but still strict and consistent

Understanding search order

- Taking into account all OSGi features, it's a little more complicated, but still strict and consistent
  - Boot delegate java.* packages, ***fail if not found***

Understanding search order

- Taking into account all OSGi features, it's a little more complicated, but still strict and consistent
  - Boot delegate java.* packages, ***fail if not found***
  - Delegate imported packages to exporter class loaders, ***fail if not found***

Understanding search order

- Taking into account all OSGi features, it's a little more complicated, but still strict and consistent
  - Boot delegate java.* packages, ***fail if not found***
  - Delegate imported packages to exporter class loaders, ***fail if not found***
  - Delegate to required bundle class loaders, ***do not fail if not found***

Understanding search order

- Taking into account all OSGi features, it's a little more complicated, but still strict and consistent
  - Boot delegate java.* packages, ***fail if not found***
  - Delegate imported packages to exporter class loaders, ***fail if not found***
  - Delegate to required bundle class loaders, ***do not fail if not found***
  - Search internal content, ***do not fail if not found***

Understanding search order

- Taking into account all OSGi features, it's a little more complicated, but still strict and consistent
  - Boot delegate java.* packages, ***fail if not found***
  - Delegate imported packages to exporter class loaders, ***fail if not found***
  - Delegate to required bundle class loaders, ***do not fail if not found***
  - Search internal content, ***do not fail if not found***
  - Attempt to dynamically import if package is not required or exported, ***if successful***
    - Delegate to exporter class loader
    - Treat as a normal import for subsequent load requests

Understanding search order

- Taking into account all OSGi features, it's a little more complicated, but still strict and consistent
  - Boot delegate java.* packages, ***fail if not found***
  - Delegate imported packages to exporter class loaders, ***fail if not found***
  - Delegate to required bundle class loaders, ***do not fail if not found***
  - Search internal content, ***do not fail if not found***
  - Attempt to dynamically import if package is not required or exported, ***if successful***
    - Delegate to exporter class loader
    - Treat as a normal import for subsequent load requests
  - ***Fail***

When things go wrong...

Unresolved constraints

- In Felix you might see something like this:

  -
    ```
    org.osgi.framework.BundleException:
    Unresolved constraint in bundle importer
    [5]: Unable to resolve 5.0: missing
    requirement [5.0] package;
    (&(package=exporter)
    (version>=1.0.0)(!(version>=2.0.0)))
    ```

When things go wrong...

Unresolved constraints

- In Felix you might see something like this:

```
– org.osgi.framework.BundleException:
  Unresolved constraint in bundle importer
  [5]: Unable to resolve 5.0: missing
  requirement [5.0] package;
  (&(package=exporter)
  (version>=1.0.0)(!(version>=2.0.0)))
```

Questions to ask yourself:
Is there a provider of the missing package?

When things go wrong...

Unresolved constraints

- In Felix you might see something like this:

  -
    ```
    org.osgi.framework.BundleException:
    Unresolved constraint in bundle importer
    [5]: Unable to resolve 5.0: missing
    requirement [5.0] package;
    (&(package=exporter)
    (version>=1.0.0)(!(version>=2.0.0)))
    ```

Questions to ask yourself:
Do import attributes match the exported
package's attributes?

When things go wrong...

Unresolved constraints

- It could also be a transitive dependency

  -
    ```
    org.osgi.framework.BundleException:
    Unresolved constraint in bundle importer
    [5]: Unable to resolve 5.0: missing
    requirement [5.0] package;
    (&(package=exporter)(version>=1.0.0)(!
    (version>=2.0.0))) [caused by: Unable to
    resolve 6.0: missing requirement [6.0]
    package; (&(package=transitive)
    (version>=1.0.0))]
    ```

It complains about not being able to resolve `exporter` package...

When things go wrong...

Unresolved constraints

- It could also be a transitive dependency
  - org.osgi.framework.BundleException: Unresolved constraint in bundle importer [5]: Unable to resolve 5.0: missing requirement [5.0] package; (&(package=exporter)(version>=1.0.0)(!(version>=2.0.0))) [**caused by: Unable to resolve 6.0: missing requirement [6.0] package; (&(package=transitive)(version>=1.0.0))**]

But actually, `exporter` was found,
but its provider has a dependency on
`transitive` that couldn't be satisfied.

When things go wrong...

Constraint violations

- In Felix you might see something like this:

  -
    ```
    org.osgi.framework.BundleException:
    Constraint violation for package 'bar'
    when resolving module 7.0 between existing
    import 6.0.bar BLAMED ON [[7.0] package;
    (&(package=bar)(version>=1.0.0)(!
    (version>=2.0.0)))] and uses constraint
    5.0.bar BLAMED ON [[7.0] package;
    (&(package=exporter1.foo)(version>=1.0.0)
    (!(version>=2.0.0)))]
    ```

When things go wrong...

Constraint violations

- In Felix you might see something like this:

  -
    ```
    org.osgi.framework.BundleException:
    Constraint violation for package 'bar'
    when resolving module 7.0 between existing
    import 6.0.bar BLAMED ON [[7.0] package;
    (&(package=bar)(version>=1.0.0)(!
    (version>=2.0.0)))] and uses constraint
    5.0.bar BLAMED ON [[7.0] package;
    (&(package=exporter1.foo)(version>=1.0.0)
    (!(version>=2.0.0)))]
    ```

Here, module 7.0 (aka bundle 7) is exposed to two versions of package `bar` from modules 5.0 and 6.0 (aka bundles 5 and 6).

When things go wrong...

Constraint violations

- In Felix you might see something like this:

  -
    ```
    org.osgi.framework.BundleException:
    Constraint violation for package 'bar'
    when resolving module 7.0 between existing
    import 6.0.bar BLAMED ON [[7.0] package;
    (&(package=bar)(version>=1.0.0)
    (!(version>=2.0.0)))] and uses constraint
    5.0.bar BLAMED ON [[7.0] package;
    (&(package=exporter1.foo)(version>=1.0.0)
    (!(version>=2.0.0)))]
    ```

Questions to ask yourself:
Are the involved bundles' import constraints accurate/specific enough?

When things go wrong...

Constraint violations

- In Felix you might see something like this:

  -
    ```
    org.osgi.framework.BundleException:
    Constraint violation for package 'bar'
    when resolving module 7.0 between existing
    import 6.0.bar BLAMED ON [[7.0] package;
    (&(package=bar)(version>=1.0.0)
    (!(version>=2.0.0)))] and uses constraint
    5.0.bar BLAMED ON [[7.0] package;
    (&(package=exporter1.foo)(version>=1.0.0)
    (!(version>=2.0.0)))]
    ```

Questions to ask yourself:
Have you deployed unnecessary providers
of the conflicting package?

When things go wrong...

Constraint violations

- In Felix you might see something like this:

  -
    ```
    org.osgi.framework.BundleException:
    Constraint violation for package 'bar'
    when resolving module 7.0 between existing
    import 6.0.bar BLAMED ON [[7.0] package;
    (&(package=bar)(version>=1.0.0)
    (!(version>=2.0.0)))] and uses constraint
    5.0.bar BLAMED ON [[7.0] package;
    (&(package=exporter1.foo)(version>=1.0.0)
    (!(version>=2.0.0)))]
    ```

Questions to ask yourself:
Were dependencies resolved incrementally
(i.e., incremental bundle deployment)?

When things go wrong...

Constraint violations

- It could also be a transitive constraint
  -
    ```
    org.osgi.Framework.BundleException:
    Constraint violation for package 'bar'
    when resolving module 8.0 between existing
    import 5.0.bar BLAMED ON [[8.0] package;
    (&(package=bar)(version>=1.0.0)(!
    (version>=2.0.0)))] and uses constraint
    7.0.bar BLAMED ON [[8.0] package;
    (&(package=exporter2.woz)(version>=1.0.0)
    (!(version>=2.0.0))), [6.0] package;
    (&(package=exporter3.boz)(version>=1.0.0)
    (!(version>=2.0.0)))]
    ```

When things go wrong...

Constraint violations

- It could also be a transitive constraint

  -
    ```
    org.osgi.Framework.BundleException:
    Constraint violation for package 'bar'
    when resolving module 8.0 between existing
    import 5.0.bar BLAMED ON [[8.0] package;
    (&(package=bar)(version>=1.0.0)(!
    (version>=2.0.0)))] and uses constraint
    7.0.bar BLAMED ON [[8.0] package;
    (&(package=exporter2.woz)(version>=1.0.0)
    (!(version>=2.0.0))), [6.0] package;
    (&(package=exporter3.boz)(version>=1.0.0)
    (!(version>=2.0.0)))]
    ```

Then you need to investigate the most deeply nested blamed requirement.

When things go wrong...

Constraint violations

- It could also be a transitive constraint

  -
    ```
    org.osgi.Framework.BundleException:
    Constraint violation for package 'bar'
    when resolving module 8.0 between existing
    import 5.0.bar BLAMED ON [[8.0] package;
    (&(package=bar)(version>=1.0.0)(!
    (version>=2.0.0)))] and uses constraint
    7.0.bar BLAMED ON [[8.0] package;
    (&(package=exporter2.woz)(version>=1.0.0)
    (!(version>=2.0.0))), [6.0] package;
    (&(package=exporter3.boz)(version>=1.0.0)
    (!(version>=2.0.0)))]
    ```

To clarify, this is the chain of imports that led to the constraint violation.

When things go wrong...

Constraint violations

- It could also be a transitive constraint
  - `org.osgi.Framework.BundleException: Constraint violation for package 'bar' when resolving` **`module 8.0`** `between existing import 5.0.bar BLAMED ON [[8.0] package; (&(package=bar)(version>=1.0.0)(!(version>=2.0.0)))] and uses constraint 7.0.bar BLAMED ON [`**`[8.0] package; (&(package=exporter2.woz)(version>=1.0.0)(!(version>=2.0.0)))`**`, [6.0] package; (&(package=exporter3.boz)(version>=1.0.0)(!(version>=2.0.0)))]`

So, here module 8.0 imports `exporter2.woz` from module 6.0...

When things go wrong...

Constraint violations

- It could also be a transitive constraint

  -
    ```
    org.osgi.Framework.BundleException:
    Constraint violation for package 'bar'
    when resolving module 8.0 between existing
    import 5.0.bar BLAMED ON [[8.0] package;
    (&(package=bar)(version>=1.0.0)(!
    (version>=2.0.0)))] and uses constraint
    7.0.bar BLAMED ON [[8.0] package;
    (&(package=exporter2.woz)(version>=1.0.0)
    (!(version>=2.0.0))), [6.0] package;
    (&(package=exporter3.boz)(version>=1.0.0)
    (!(version>=2.0.0)))]
    ```

Who imports `exporter3.boz` from module 7.0, which apparently has a “uses” contraint on `bar`.

When things go wrong...

ClassNotFoundException

- In Felix you might see something like this:

  -
    ```
    java.lang.ClassNotFoundException:
    exporter.Exporter not found by importer [5]
    at org.apache.felix.framework.
    ModuleImpl.findClassOrResourceByDelegation(
    ModuleImpl.java:787)
    at org.apache.felix.framework.
    ModuleImpl.access$400(ModuleImpl.java:71)
    at org.apache.felix.framework.
    ModuleImpl$ModuleClassLoader.loadClass(Modu
    leImpl.java:1768)
    ... 36 more
    ```

When things go wrong...

ClassNotFoundException

- In Felix you might see something like this:

```
- java.lang.ClassNotFoundException:
  exporter.Exporter not found by importer [5]
  at org.apache.felix.framework.
  ModuleImpl.findClassOrResourceByDelegation(
  ModuleImpl.java:787)
  at org.apache.felix.framework.
  ModuleImpl.access$400(ModuleImpl.java:71)
  at org.apache.felix.framework.
  ModuleImpl$ModuleClassLoader.loadClass(Modu
  leIr
  ...
```

Questions to ask yourself:
Is the class in question supposed to be in the bundle or imported?

When things go wrong...

ClassNotFoundException

- In Felix you might see something like this:

```
– java.lang.ClassNotFoundException:
  exporter.Exporter not found by importer [5]
  at org.apache.felix.framework.
  ModuleImpl.findClassOrResourceByDelegation(
  ModuleImpl.java:787)
  at org.apache.felix.framework.
  ModuleImpl.access$400(ModuleImpl.java:71)
  at org.apache.felix.framework.
  ModuleImpl$ModuleClassLoader.loadClass(Modu
  leI
  ...
```

Questions to ask yourself:
If it's a bundle class, does the bundle actually contain the class?

When things go wrong...

ClassNotFoundException

- In Felix you might see something like this:

  -
    ```
    java.lang.ClassNotFoundException:
    exporter.Exporter not found by importer [5]
    at org.apache.felix.framework.
    ModuleImpl.findClassOrResourceByDelegation(
    ModuleImpl.java:787)
    at org.apache.felix.framework.
    ModuleImpl.access$400(ModuleImpl.java:71)
    at org.apache.felix.framework.
    ModuleImpl$ModuleClassLoader.loadClass(Modu
    leIr
    ...
    ```

Questions to ask yourself:
If it's an imported class, does the bundle actually import the package?

When things go wrong...

ClassNotFoundException

- In Felix you might see something like this:

  -
    ```
    java.lang.ClassNotFoundException:
    exporter.Exporter not found by importer [5]
    at org.apache.felix.framework.
    ModuleImpl.findClassOrResourceByDelegation(
    ModuleImpl.java:787)
    at org.apache.felix.framework.
    ModuleImpl.access$400(ModuleImpl.java:71)
    at org.apache.felix.framework.
    ModuleImpl$ModuleClassLoader.loadClass(Modu
    leI
    ...
    ```

Questions to ask yourself:
If it does import the package, does the exporting bundle actually contain the class?

When things go wrong...

NoClassDefError

- In Felix you might see something like this:

  -
    ```
    java.lang.NoClassDefFoundError:
    exporter/Other
    at exporter.Exporter.<init>(Exporter.java:7)
    at importer.Importer.start(Importer.java:10)
    at org.apache.felix.framework.util.
    SecureAction.startActivator
    (SecureAction.java:629)
    at org.apache.felix.framework.Felix.
    activateBundle(Felix.java:1827)
    ... 32 more
    ```

When things go wrong...

NoClassDefError

- In Felix you might see something like this:

  -
    ```
    java.lang.NoClassDefFoundError:
    exporter/Other
    at exporter.Exporter.<init>(Exporter.java:7)
    at importer.Importer.start(Importer.java:10)
    at org.apache.felix.framework.util.
    SecureAction.startActivator
    (SecureAction.java:629)
    at org.apache.felix.framework.Felix.
    activateBundle(Felix.java:1827)
    ...
    ```

Questions to ask yourself:
The same types of questions as with
class not found exceptions...

When things go wrong...

NoClassDefError

- In Felix you might see something like this:

  -
    ```
    java.lang.NoClassDefFoundError:
    exporter/Other
    at exporter.Exporter.<init>(Exporter.java:7)
    at importer.Importer.start(Importer.java:10)
    at org.apache.felix.framework.util.
    SecureAction.startActivator
    (SecureAction.java:629)
    at org.apache.felix.framework.Felix.
    activateBundle(Felix.java:1827)
    ...
    ```

The tricky part is that the class in question is not directly relevant to you...

When things go wrong...

NoClassDefError

- In Felix you might see something like this:

```
– java.lang.NoClassDefFoundError:
  exporter/Other
  at exporter.Exporter.<init>(Exporter.java:7)
  at importer.Importer.start(Importer.java:10)
  at org.apache.felix.framework.util.
  SecureAction.startActivator
  (SecureAction.java:629)
  at org.apache.felix.framework.Felix.
  activateBundle(Felix.java:1827)
  ...
```

Here, the `Importer` was creating `Exporter`, but the failure is for `Other`, which `Importer` might know nothing about...

When things go wrong...

NoClassDefError

- In Felix you might see something like this:

  -
    ```
    java.lang.NoClassDefFoundError:
    exporter/Other
    at exporter.Exporter.<init>(Exporter.java:7)
    at importer.Importer.start(Importer.java:10)
    at org.apache.felix.framework.util.
    SecureAction.startActivator
    (SecureAction.java:629)
    at org.apache.felix.framework.Felix.
    activateBundle(Felix.java:1827)
    ...
    ```

This means means the issue is likely in the bundle containing `Exporter`, not the bundle containing `Importer`.

Poking around

- Use the Gogo shell to see what's going on
  - `lb` – to list installed bundles
  - `headers` – to view a bundle's manifest main headers
  - `inspect p[ackage] c[apability]` – to view a bundle's exported packages with wiring
  - `inspect p[ackage] r[equirement]` – to view a bundle's imported packages with wiring
  - `which` – to try to load a class from a bundle and see from where it comes

How to load classes?

- Generally speaking
  - Your modules should ***not*** need to explicitly load classes
  - Normal, on-demand, implicit class loading as your code executes ***should be sufficient***

How to load classes?

- Generally speaking
  - Your modules should ***not*** need to explicitly load classes
  - Normal, on-demand, implicit class loading as your code executes ***should be sufficient***
- But, what if this isn't sufficient?
  - What if your code needs to dynamically load a class?

How to load classes?

- First things first
  - Don't use `Class.forName()`
    - The resulting class is cached in the defining AND the initiating class loader
      - Subsequent requests from the initiating class loader will always return the same class, which is not usually what you want
      - Inhibits garbage collection
  - Yes, the JavaDocs tell you to use `Class.forName()`, but still don't
    - One of the main arguments for `Class.forName()` is that it handles array types, but OSGi class loaders should handle this too via `ClassLoader.loadClass()`

How to load classes?

- If you are loading a class on behalf of a client, some options are
  - If the client provides a client-loaded object, then use its class loader
  - Allow the client to provide the needed class loader as a parameter
  - Require that the client set/unset the Thread Context Class Loader before performing operation

How to load classes?

- If no client is involved, then some options are
  - If you know it will always be the same class at execution time, you just don't know which one, use dynamic imports
    - e.g., maybe the class is set via a configuration property
  - Search installed bundles and use `Bundle.loadClass()`
    - a la the extender pattern

How to load classes?

- Another alternative, use services and the service registry
  - Provides a loosely-coupled collaboration mechanism
  - Can eliminate the need to deal directly with class loaders
    - Rather than looking for classes to instantiate, look for instantiated service objects

Conclusions

Conclusions

- When using OSGi ***you must unlearn*** the global type visibility assumption
  - OSGi provides strict and explicit type visibility rules to ***give control back to you***

Conclusions

- When using OSGi ***you must unlearn*** the global type visibility assumption
  - OSGi provides strict and explicit type visibility rules to ***give control back to you***
- If you change your mindset, then your code will work well with (or without) OSGi...
  - ***...and then you can begin to ignore it***

If you want all the details...

Get this book - http://www.manning.com/hall/