



Classloading and Type Visibility in OSGi

Martin Lippert
akquinet it-agile GmbH
martin.lippert@akquinet.de



Overview

- Introduction to classloading
 - ◆ What is classloading?
 - ◆ How does classloading work?
 - ◆ What does classloading mean for daily development?
- Classloading in OSGi
 - ◆ What is different?
 - ◆ Dependency and Visibility
 - ◆ Advanced Classloading in OSGi
- Conclusions



What is Classloading?

- Classloaders are Java objects
- They are responsible for loading classes into the VM
 - ◆ Every class is loaded by a classloader into the VM
 - ◆ There is no way around
- Every class has a reference to its classloader object
 - ◆ `myObject.getClass().getClassLoader()`



ClassLoader API

```
public abstract class ClassLoader {  
  
    public Class<?> loadClass(String name)  
  
    public URL getResource(String name)  
    public Enumeration<URL> getResources(String name)  
    public InputStream getResourceAsStream(String name)  
  
    public final ClassLoader getParent()  
  
    public static URL getSystemResource(String name)  
    public static Enumeration<URL> getSystemResources(String name)  
    public static InputStream getSystemResourceAsStream(String name)  
    public static ClassLoader getSystemClassLoader()  
  
    ...  
}
```



Implicit class loading

```
public class A {  
    public void foo () {  
        B b = new B ();  
        b.sayHello ();  
    }  
}
```



causes the VM to load class B using the classloader of A



Explicit class loading

```
public void foo() {  
    ClassLoader cl =  
        this.getClass().getClassLoader();  
    Class<?> clazz = cl.loadClass("A");  
    Object obj = clazz.newInstance();  
  
    ...  
}
```

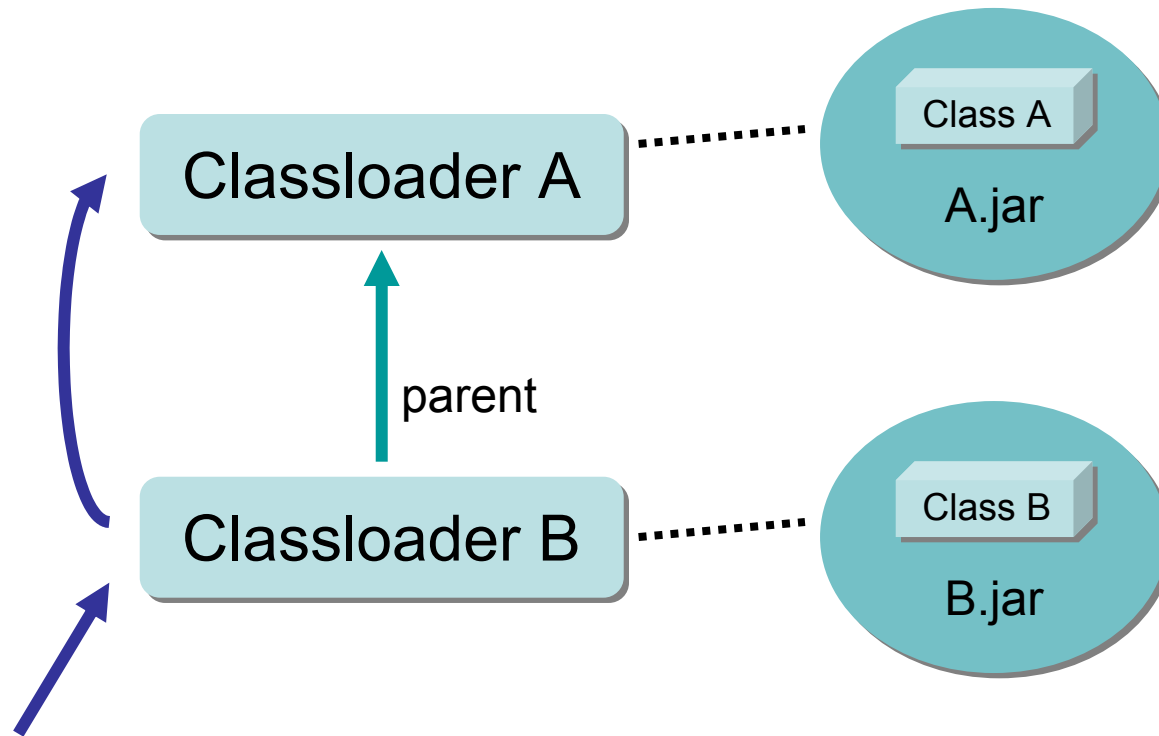


Hierarchical classloaders

- Classloaders typically have a parent classloader
 - ◆ Chained classloading
- If a classloader is invoked to load a class, it first calls the parent classloader
 - ◆ Parent first strategy
 - ◆ This helps to prevent loading the same class multiple times

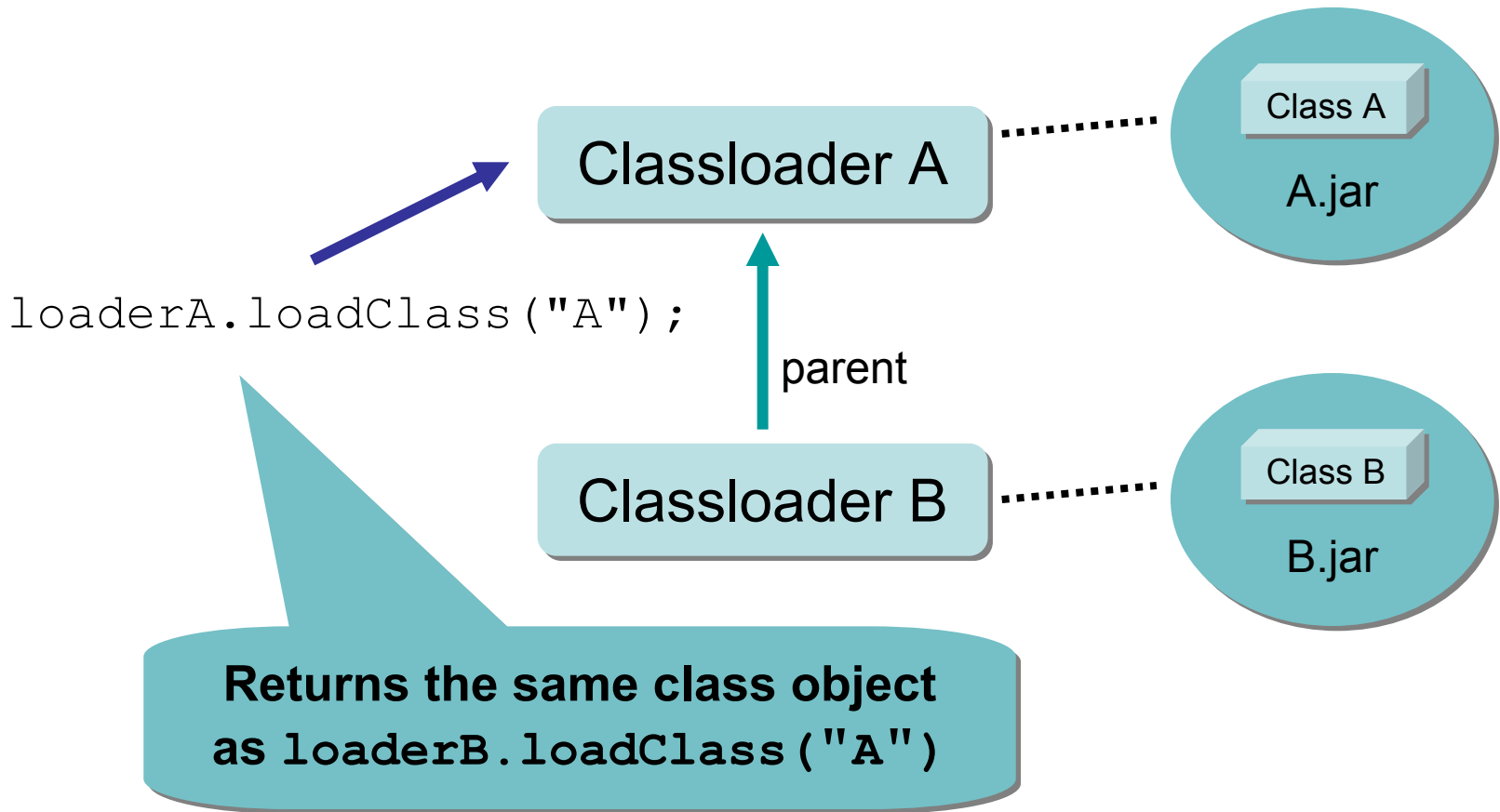


ClassLoader hierarchy

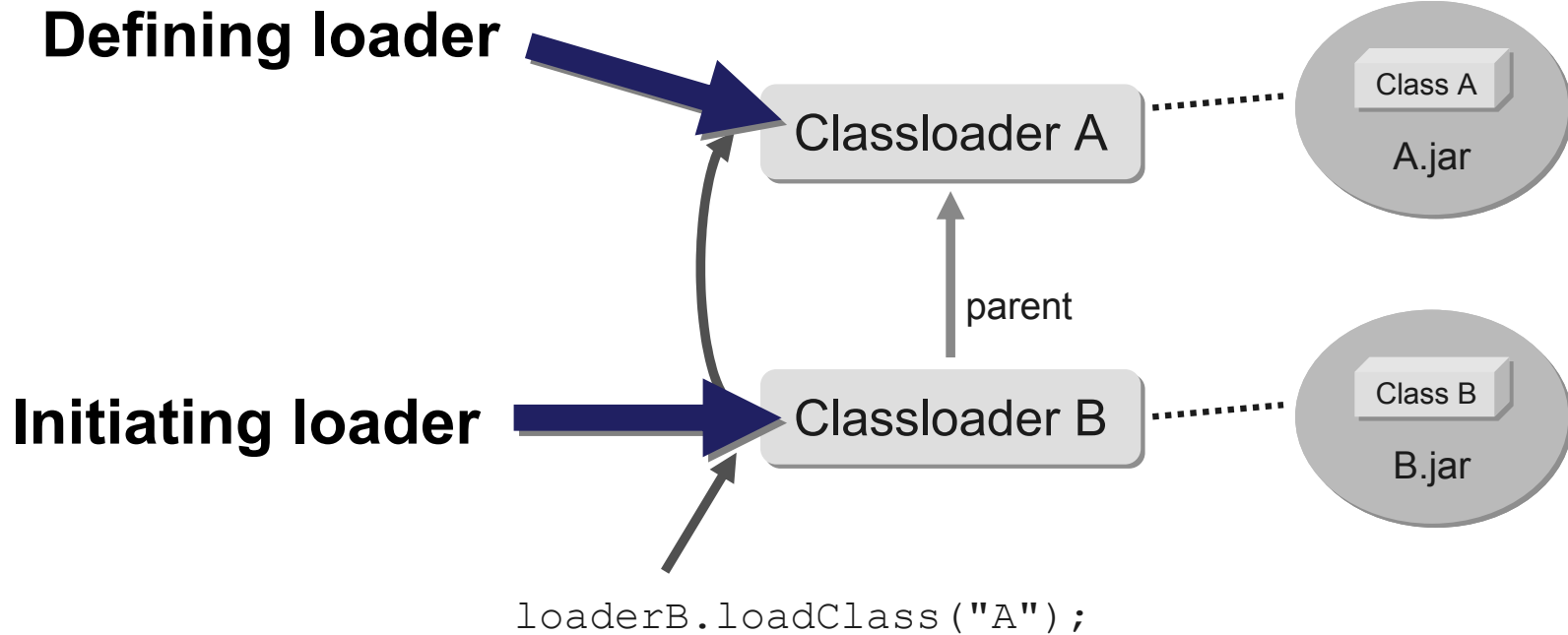


```
loaderB.loadClass("A");
```

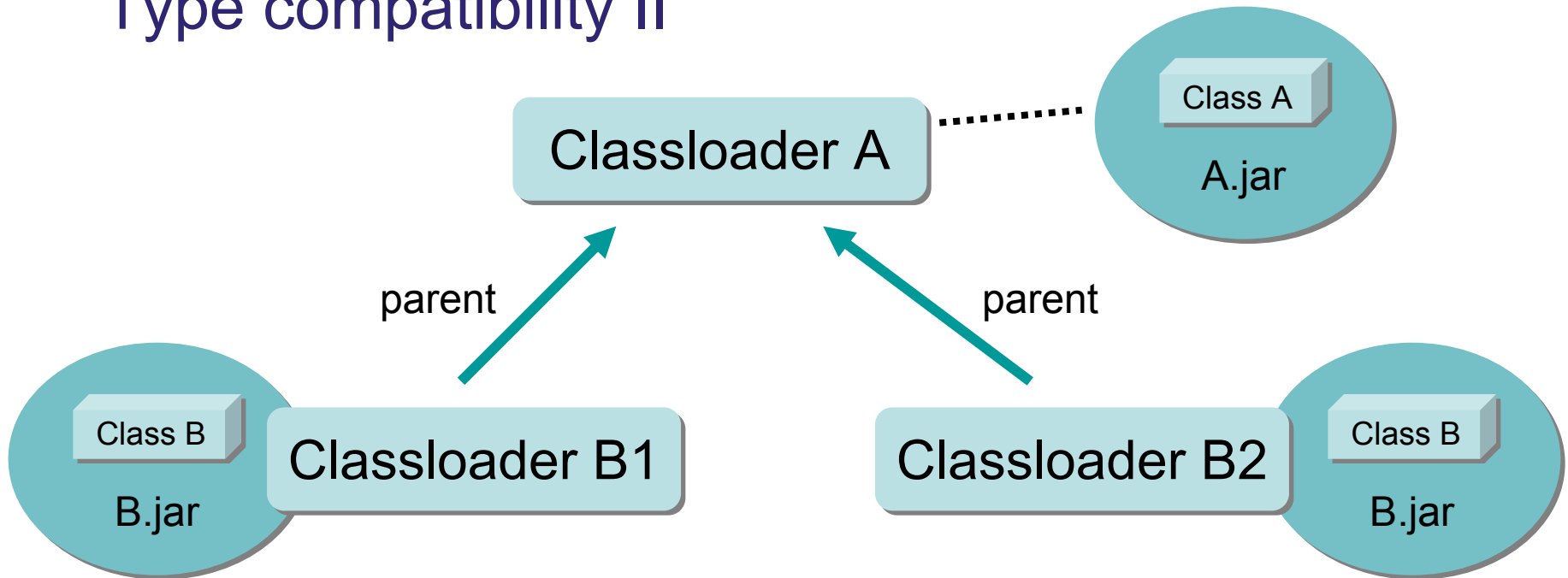

Type compatibility



Defining vs. Initiating classloader

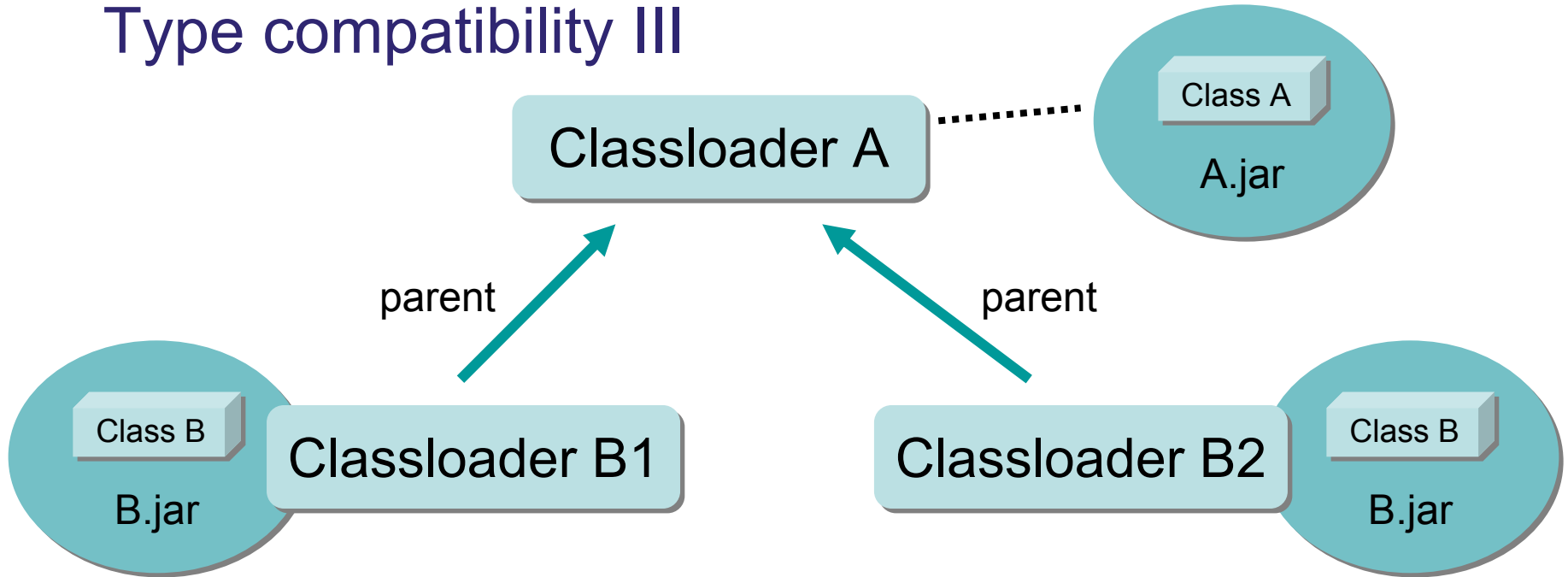


Type compatibility II



```
loaderB1.loadClass("A") == loaderB2.loadClass("A")  
loaderB1.loadClass("B") != loaderB2.loadClass("B")
```

Type compatibility III

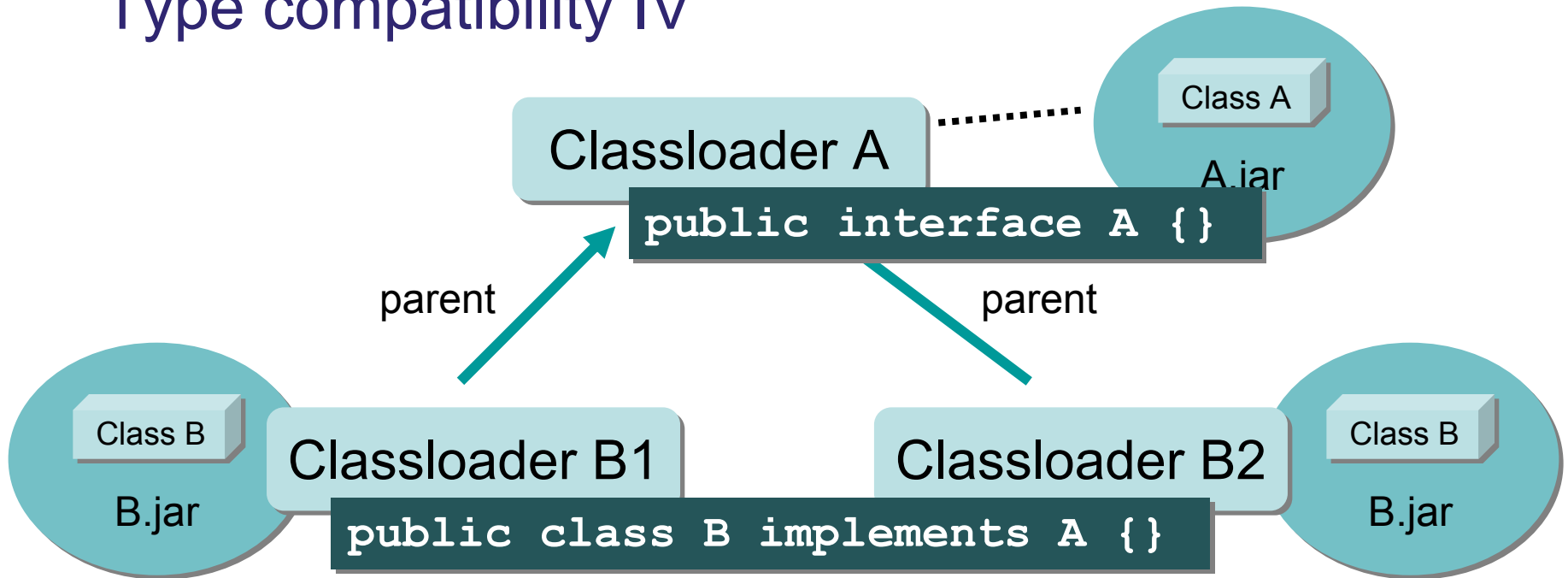


```
Object b1 = loaderB1.loadClass("B").newInstance();
```

```
b1 ! instanceof loaderB2.loadClass("B")
```

Remember: A class is identified by its name (including the package name) AND its defining class loader !!!

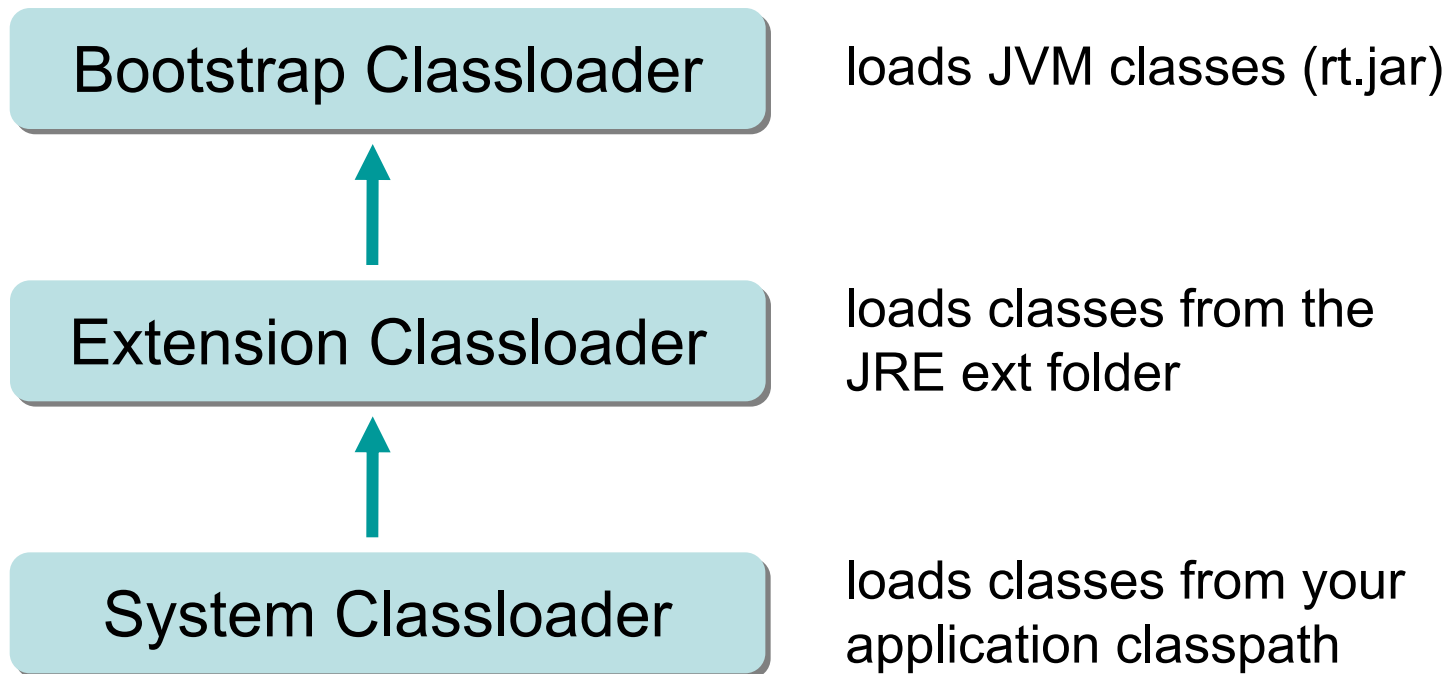
Type compatibility IV



```
A anA = loaderB1.loadClass("B").newInstance();  
A anotherA = loaderB2.loadClass("B").newInstance();  
anA = anotherA; (Assignment)
```



The default setting





Threads context classloader

```
Thread.currentThread().getContextClassLoader()
```

```
Thread.currentThread().setContextClassLoader(...)
```

- Typically used in libraries to access the context in which the library is called



ClassLoader.loadClass vs. Class.forName

- `ClassLoader.loadClass()` caches the loaded class object and returns always the same class object
 - ◆ **This is done by the defining classloader**
 - ◆ This ensures that each classloader loads the same class only once
- `Class.forName()` calls the normal classloader hierarchy to load the class (same happens as above)
 - ◆ **But caches the class object within the initiating classloader**
 - ◆ In standard cases no problem but can be tricky in dynamic environments



Classloading is dynamic

- You can create classloaders at runtime
- You can trigger them to load a specific class
- For example:
 - ◆ What app/web servers do for hot deployment
- Some people say the classloading mechanism is the only real innovation in the Java programming language

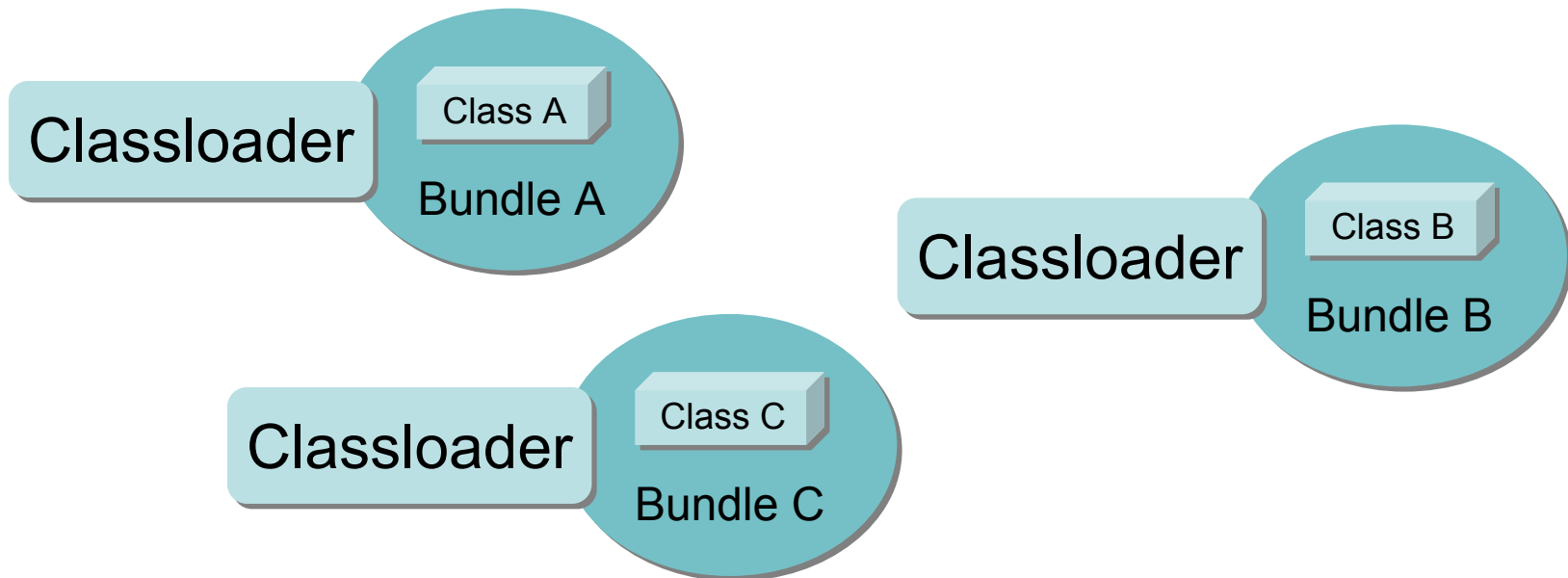


Classloading in OSGi

- “OSGi is a classloading framework”
- Remember:
 - ◆ Dependencies between bundles
 - Import- and Export-Package, Require-Bundle
 - ◆ Dynamic Bundle Lifecycle
 - Install, Update, Uninstall bundles
- Realized via specialized classloading

ClassLoader per bundle

- One classloader per bundle
 - ◆ Controls what is visible from the bundle
 - ◆ Controls what is visible from other bundles

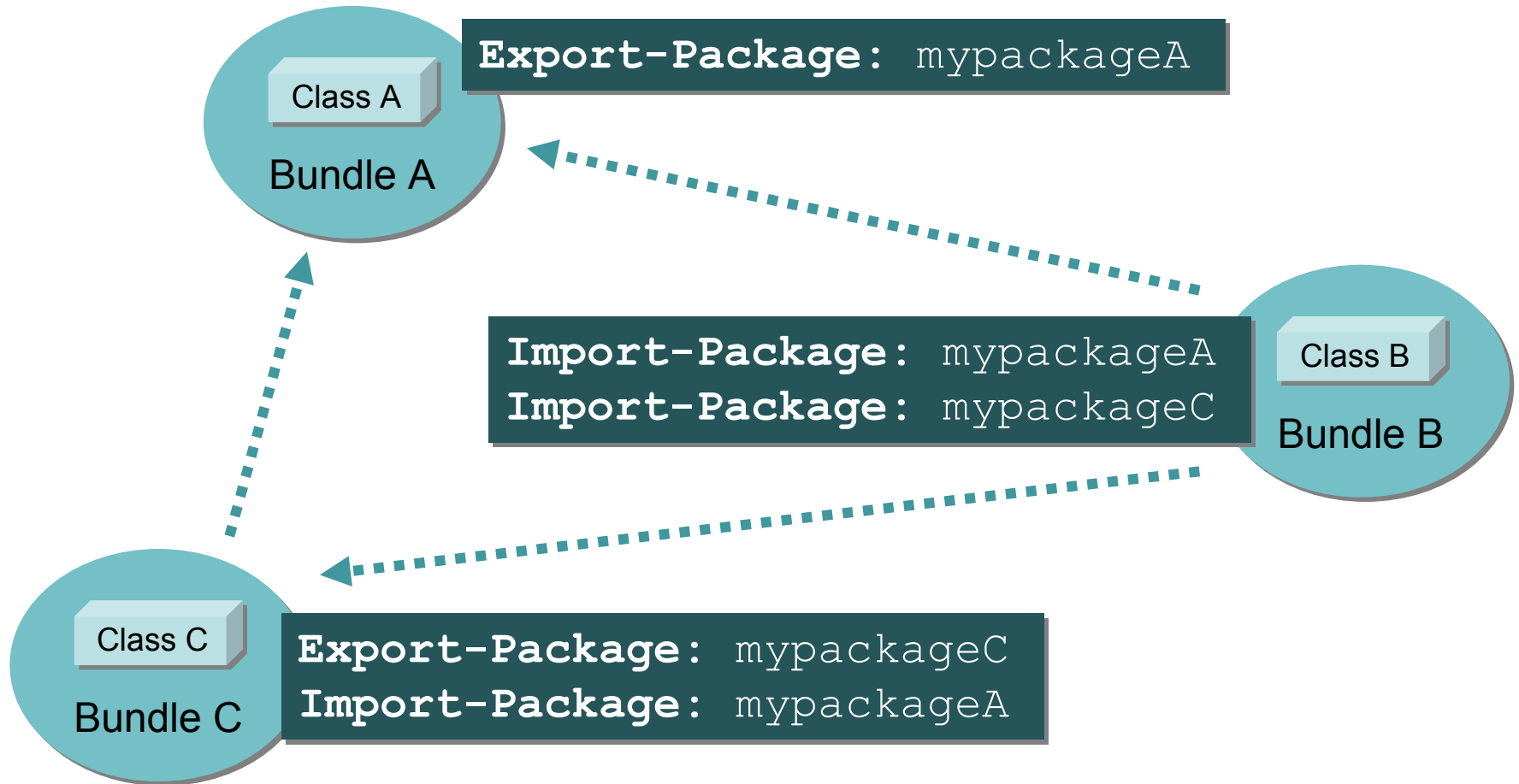




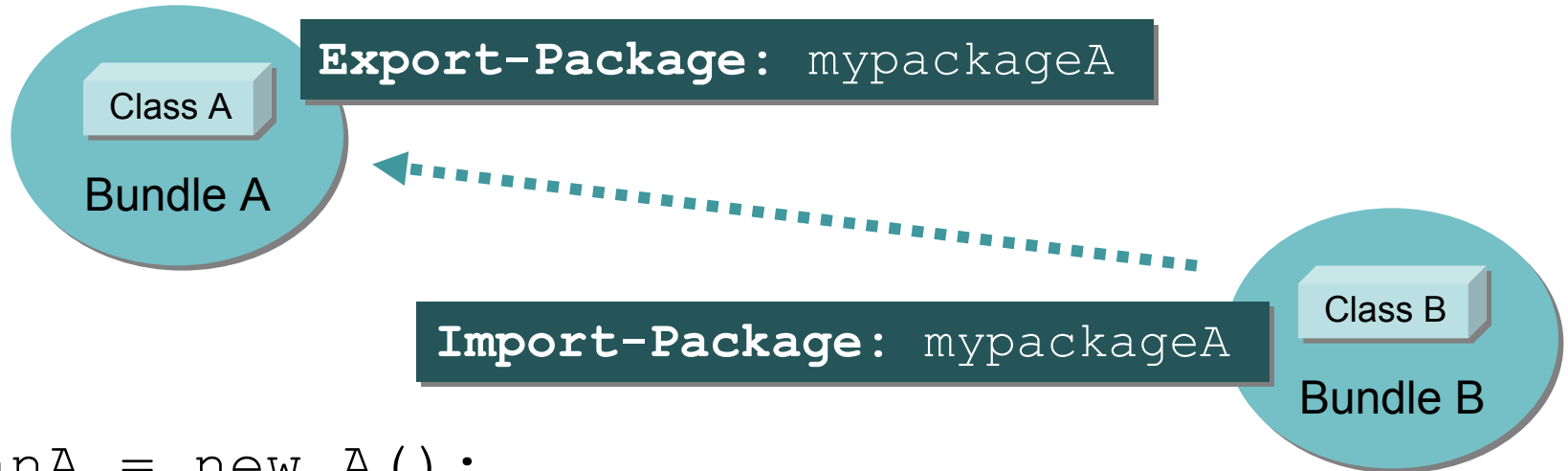
ClassLoader per bundle

- Effects
 - ◆ No linear class path for your application anymore
 - ◆ Instead class path per bundle
 - ◆ No real parent hierarchy anymore
- Classloader parent setting
 - ◆ Default: Bootstrap classloader
 - ◆ Can be parameterized via system property

Dependencies via delegation



Type Visibility I

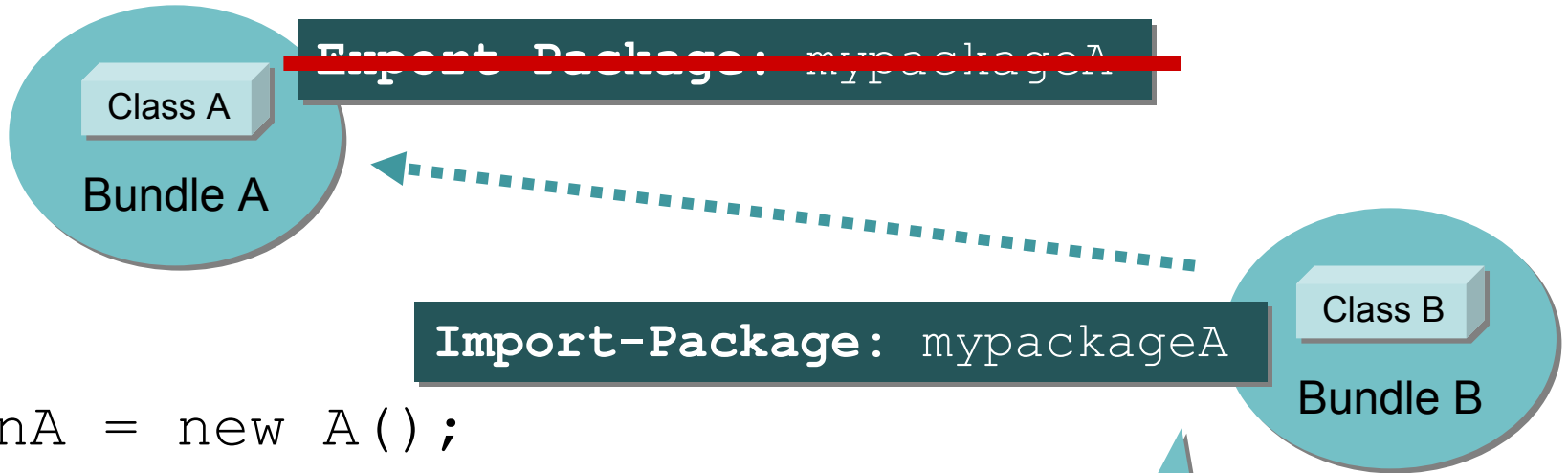


```
A anA = new A ();
```

```
A anA = new A ();
```

class A is loaded only once by bundle A (its classloader)

Type Visibility II



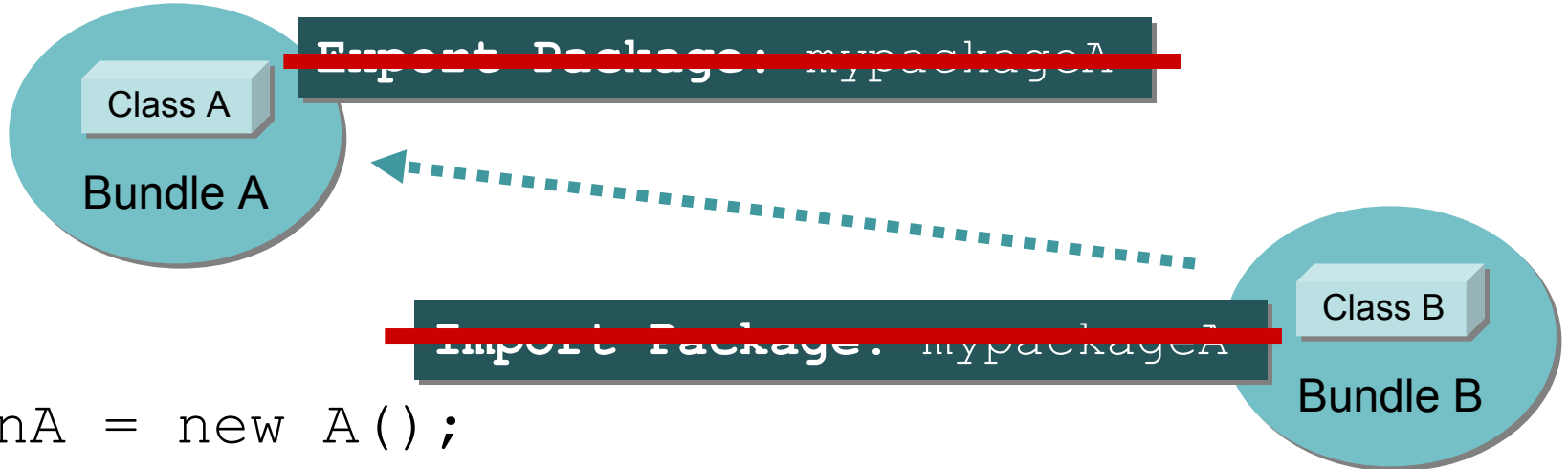
```
A anA = new A ();
```

class is loaded successfully

```
A anA = new A ();
```

bundle B remains in state "installed" (not resolved)

Type Visibility III



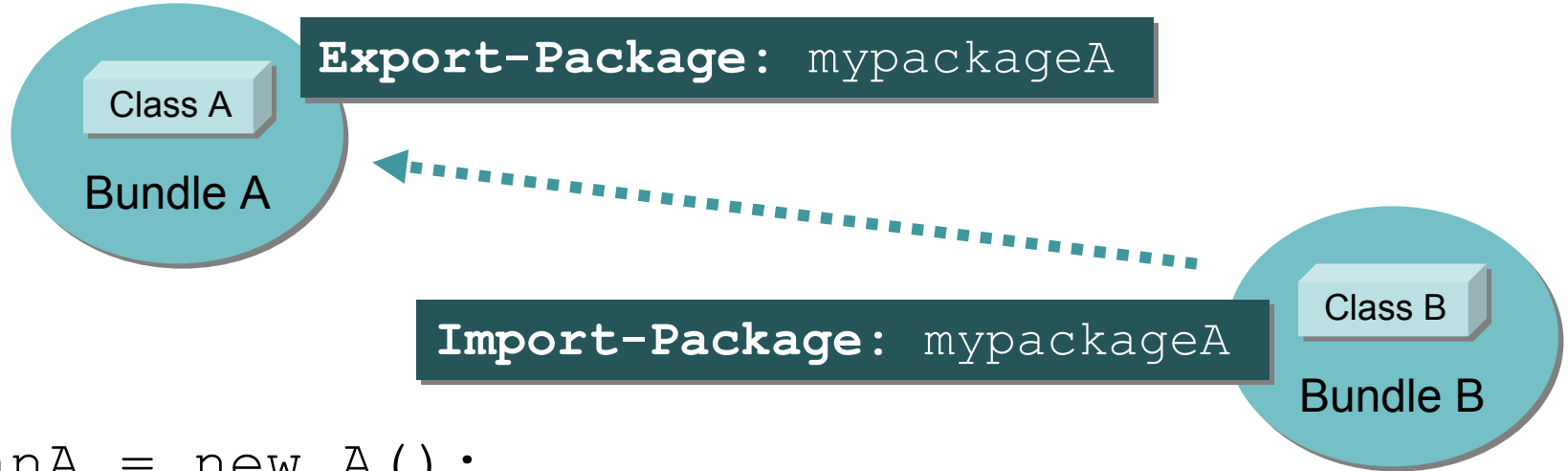
```
A anA = new A ();
```

class is loaded successfully

```
A anA = new A ();
```

ClassNotFoundException

Type Compatibility revisited I



```
A anA = new A ();
```

```
A anotherA = new A ();
```

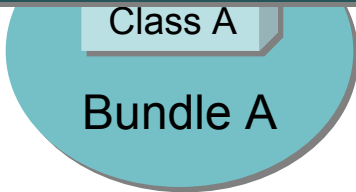
exactly the same type



Type Compatibility revisited II

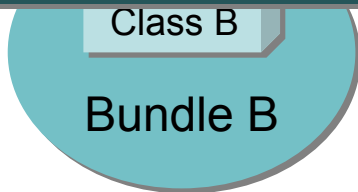
Export-Package:

`mypackageA;version="1.0.0"`



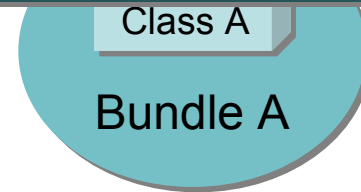
Import-Package:

`mypackageA;version="1.0.0"`



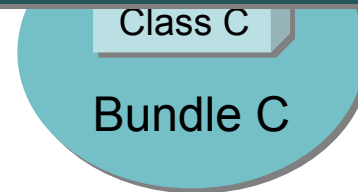
Export-Package:

`mypackageA;version="2.0.0"`



Import-Package:

`mypackageA;version="2.0.0"`

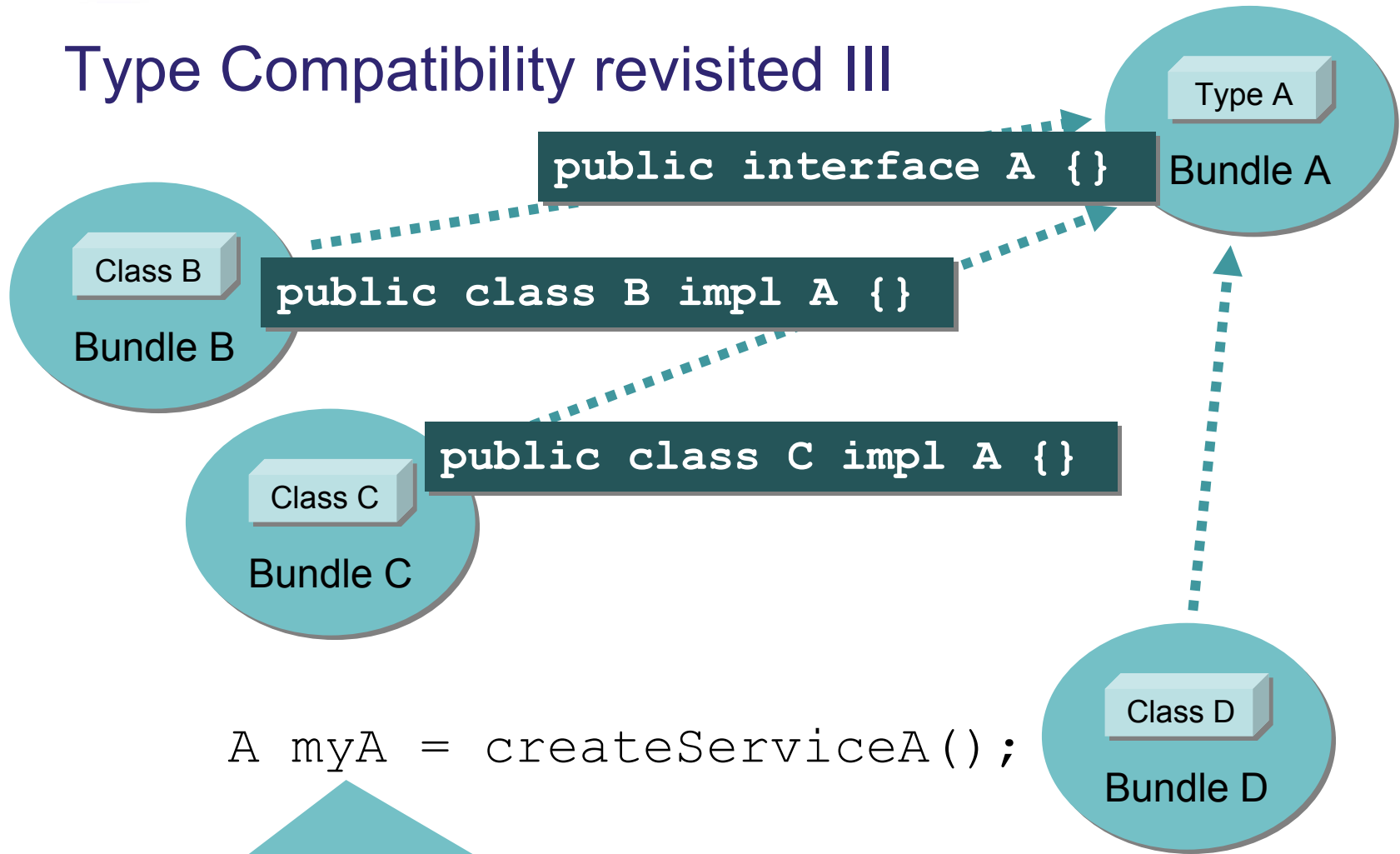


`A anA = new A();`

`A anA = new A();`

Completely different and incompatible types

Type Compatibility revisited III



```
A myA = createServiceA();
```

Static type of myA is A, dynamic type of myA could be B or C



ClassNotFoundException

- Typical reasons for a `ClassNotFoundException`:
 - ◆ Dependency to declaring bundle not defined
 - ◆ Type is not visible (not exported)
- Dynamically generated classes
 - ◆ Proxies
 - ◆ CGLib
 - ◆ ...



Buddy loading I

- What happens if a library needs to load classes from its clients?
 - ◆ e.g. persistence libraries?
- Cyclic dependencies are not allowed and maybe even not what you want
- Equinox provides so called “Buddy Loading”



Buddy loading II

Eclipse-BuddyPolicy: registered

org.hibernate
Bundle

Allows org.hibernate bundle to
execute successfully
`loadClass("A")`

**Eclipse-RegisterBuddy:
org.hibernate**

Class A

Bundle A

```
A anA = new A();
```



Buddy loading III

- Important difference:
 - ◆ Buddy loading can load **all** classes from a buddy bundle
 - ◆ not only exported types
- Its just a workaround for libraries and other existing code that does not behave correctly within the OSGi world



The loading sequence

1. Try the parent for “java.” packages
2. Try the parent for boot delegation packages
3. Try to find it from imported packages
4. Try to find it from required bundles
5. Try to find it from its own class path
6. Try to find it from dynamic import
7. Try to find it via buddy loading



“High Performance Classloading”

- Classloading consumes a remarkable amount of time at startup
- OSGi allows to highly optimize classloading
 - ◆ Finding the right class
 - ◆ Highly optimized implementations available



Classloading Hooks

- Equinox provides a hook mechanism
 - ◆ To enhance and modify the behavior of the runtime
- Examples
 - ◆ Modify bytecode at load-time
 - ◆ Intercept bundle data access
- Eat your own dog food
 - ◆ Some Equinox features are implemented using those hooks
 - ◆ e.g. Eclipse-LazyStart



Garbage Collection for Classloaders

- You could expect that the classloader of a bundle gets garbage collected if the bundle is stopped or uninstalled
- **This is not automatically the case!!!**
- You need to ensure that all objects from those classes loaded by this classloader are no longer referenced



What does this mean?

- Bundle A provides an interface I and a registry for objects of type I
- Bundle X has a class Y that implements I, creates objects of type Y and registers those objects at the registry
- If you uninstall X, you need to cut the references that still exists in the registry of A.



Conclusions

- Changing the viewpoint from the linear classpath to a per-bundle classpath
- Clearly defined dependencies and visibilities
 - ◆ Real modularity
 - ◆ Classloading only implementation detail
- **Use OSGi in a clean and correct way and you never need to think about classloading at all**



Thank you for your attention!

Q&A

Martin Lippert: martin.lippert@akquinet.de