Java Language Modularity With Superpackages

Alex Buckley

JSR 294 Co-spec lead
Sun Microsystems

Andreas Sterbenz

JSR 294 Co-spec lead
Sun Microsystems

TS-2401

Goal

In the Next 60 Minutes...

Discover how superpackages will ease information hiding and promote encapsulation in your Java™ applications

Agenda

Modularity

Information Hiding

Superpackages

Package Interfaces

Java Specification Request (JSR) 294

Q&A

Agenda

Modularity

Information Hiding

Superpackages

Package Interfaces

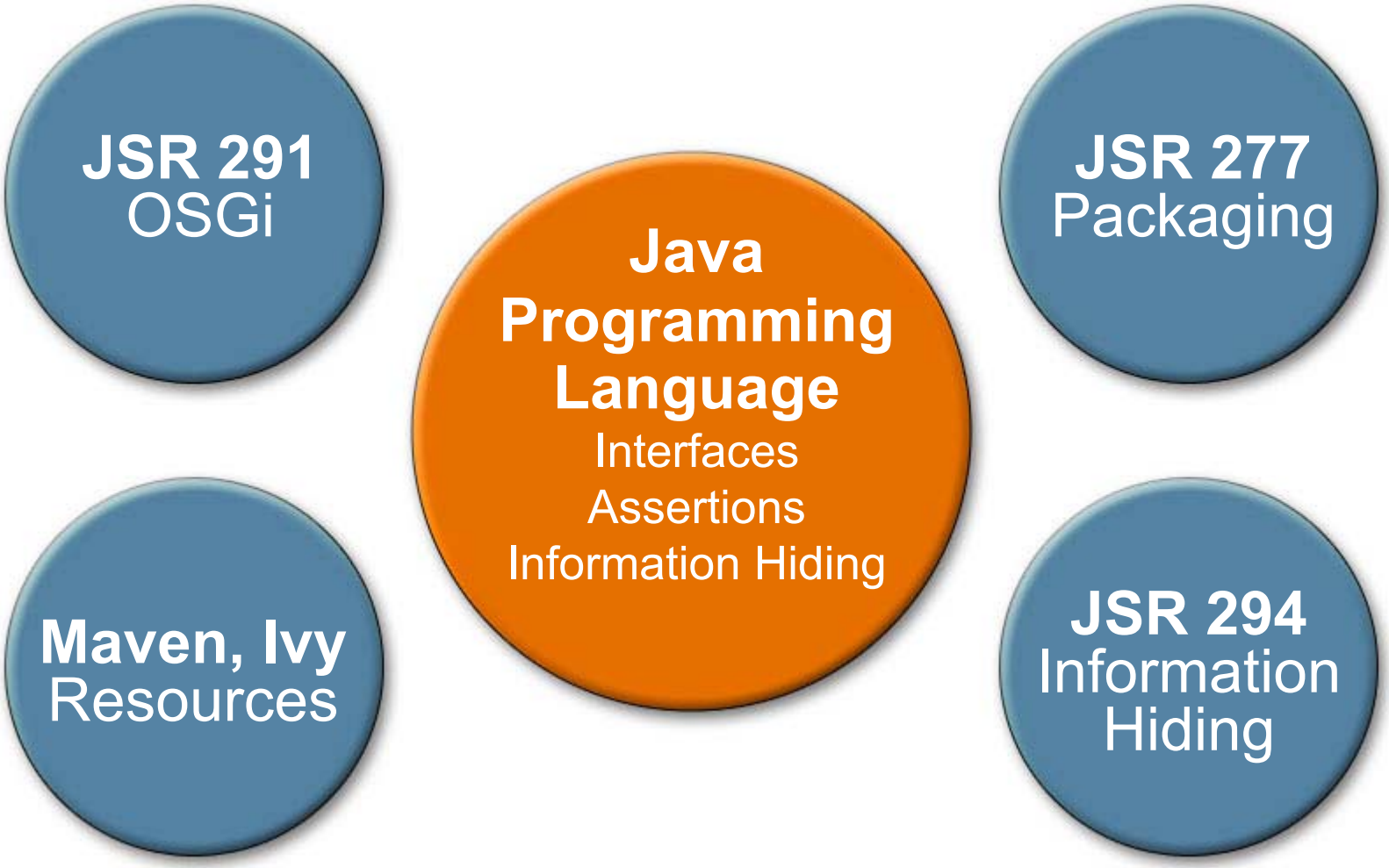
Java Specification Request (JSR) 294

Q&A

What Makes a Program Modular?

- Interfaces
  - Don’t rely on implementations
- Protocols
  - Enforce good idioms
- Information hiding
  - If you can’t see it, you can’t use it
- Contracts
  - If you use it, use it right
- Versions, resource declarations, centralized exception handling, etc.

Standards for Modularity



Agenda

Modularity

Information Hiding

Superpackages

Package Interfaces

JSR 294

Q&A

Information Hiding Circa 1972

“ Each module is characterized by its knowledge of a design decision which it hides from all others. Its interface is chosen to reveal as little as possible about its inner workings. ”

D.L.Parnas
http://www.acm.org/classics/may96/

Information Hiding Circa 2007

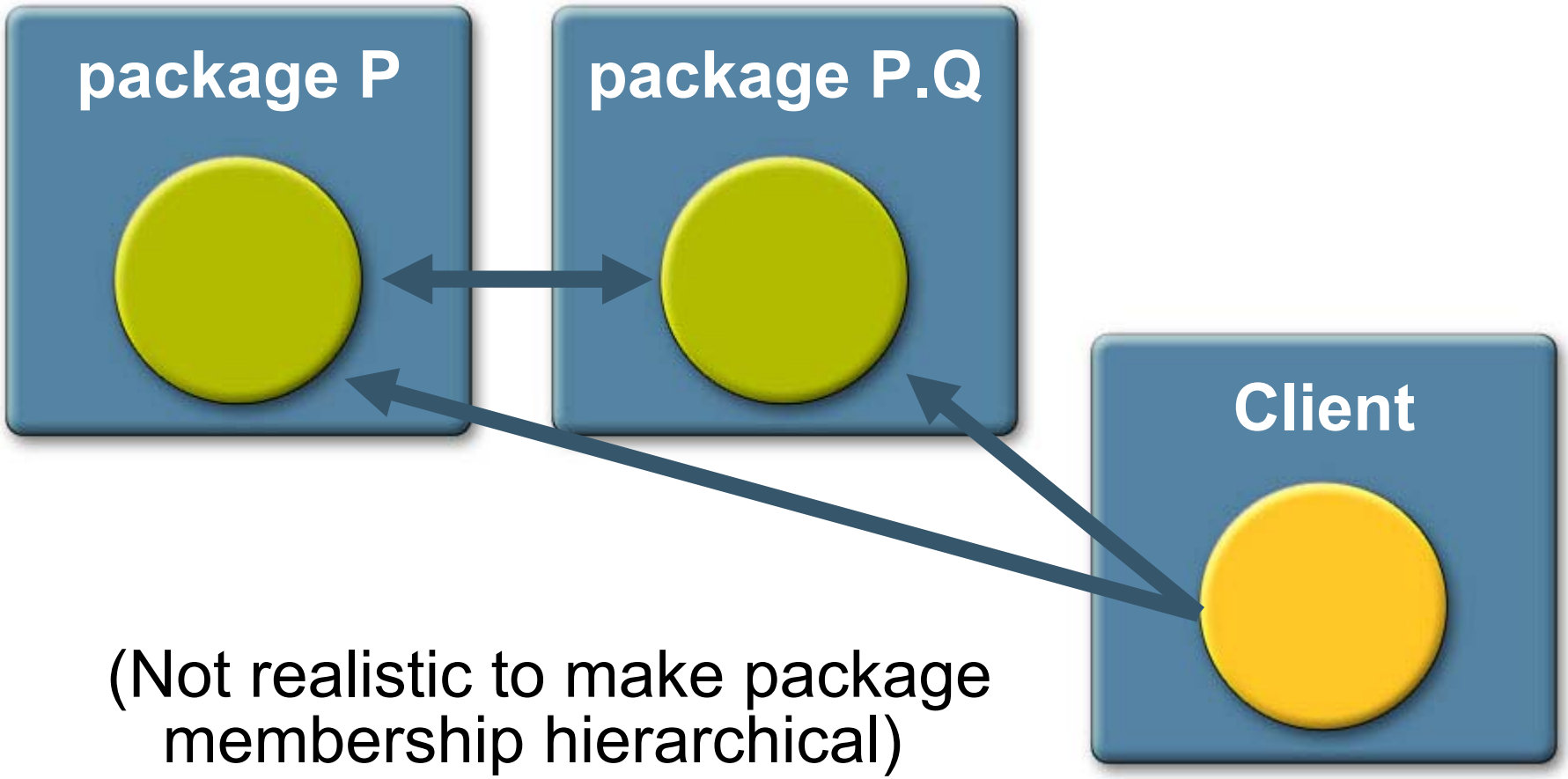
- Information hiding supports encapsulation
- Encapsulation supports reliable software

- Information hiding is an issue of *program design*
- Should be enabled by the language itself
- Accessibility modifiers strike a good balance
  - Simple
  - No overlap (Witness removal of `private protected`)

Packages for Information Hiding

- Packages help to prevent name conflicts
  - But are not namespaces
- Packages support information hiding
  - But do not follow its central tenet: Provide an interface
- Package names are hierarchical
  - But package membership is not hierarchical
  - Member of package P.Q is not member of package P
- Projects are often larger than a single package
  - But packages cannot be composed into a larger entity
  - "`public` is too public"

"`public` is too public"



(Not realistic to make package membership hierarchical)

Existing Approaches

- Lack of documentation
- Class loaders
- Static classes

Information Hiding via Lack of Documentation

- Idea: Do not document “internal” packages
- Hope nobody will find them
- Problems are obvious

Information Hiding via Class Loaders

- Idea: two class loaders per “component”
  - Internal class loader—Resolves all classes
  - Public class loader—Resolves only exported classes
- Problems
  - Does not address compile time
  - Does not prevent access via reflection
  - Breaks down if internal class loader object is “leaked”
  - Sometimes unclear which class loader should be used
    - What should the context class loader be set to?
- Not what class loaders were designed for

Information Hiding via Static Classes

- Instead of creating multiple packages, put all classes into one package
  - Each package becomes a top-level public class
  - Each class becomes a static nested class
    - Public, package private, or private as desired
- Problems
  - Non-intuitive
  - At the VM level, there are no private nested classes
    - They are realized as package private classes
  - Converting existing code requires renaming classes
  - Many classes in each source file

Goal: A language Construct for Information Hiding

- An entity bigger than a package
  - Accessibility within the entity is wider than package-private but narrower than public
- Hierarchical names guide accessibility
  - Entity P.Q can be a [hidden] member of entity P
- Run-time access control
  - Universally understood
- Interfaces for packages and these new entities
- A basis for deployment modules

What We Don't Want to Do

- Introduce `friend`
  - Appropriate in C++ (operator overloading, no packages)
  - Offers no higher-level entity for composing types
  - Offers no basis for deployment modules
- Embed deployment modules in the language
- Burden programmers who don't use the new entity
- Have compile-time and run-time behavior differ
  - No leaky abstractions!

Agenda

Modularity

Information Hiding

Superpackages

Package Interfaces

JSR 294

Q&A

Definition

- A superpackage is a named collection of one or more packages or superpackages and their types
- Public types can be declared as exported to make them accessible outside the superpackage
- Public types that are not exported are accessible only to other types in the superpackage
- Declared in a Java source file (`super-package.java`) and compiled by the Java compiler

What a Superpackage Is Not

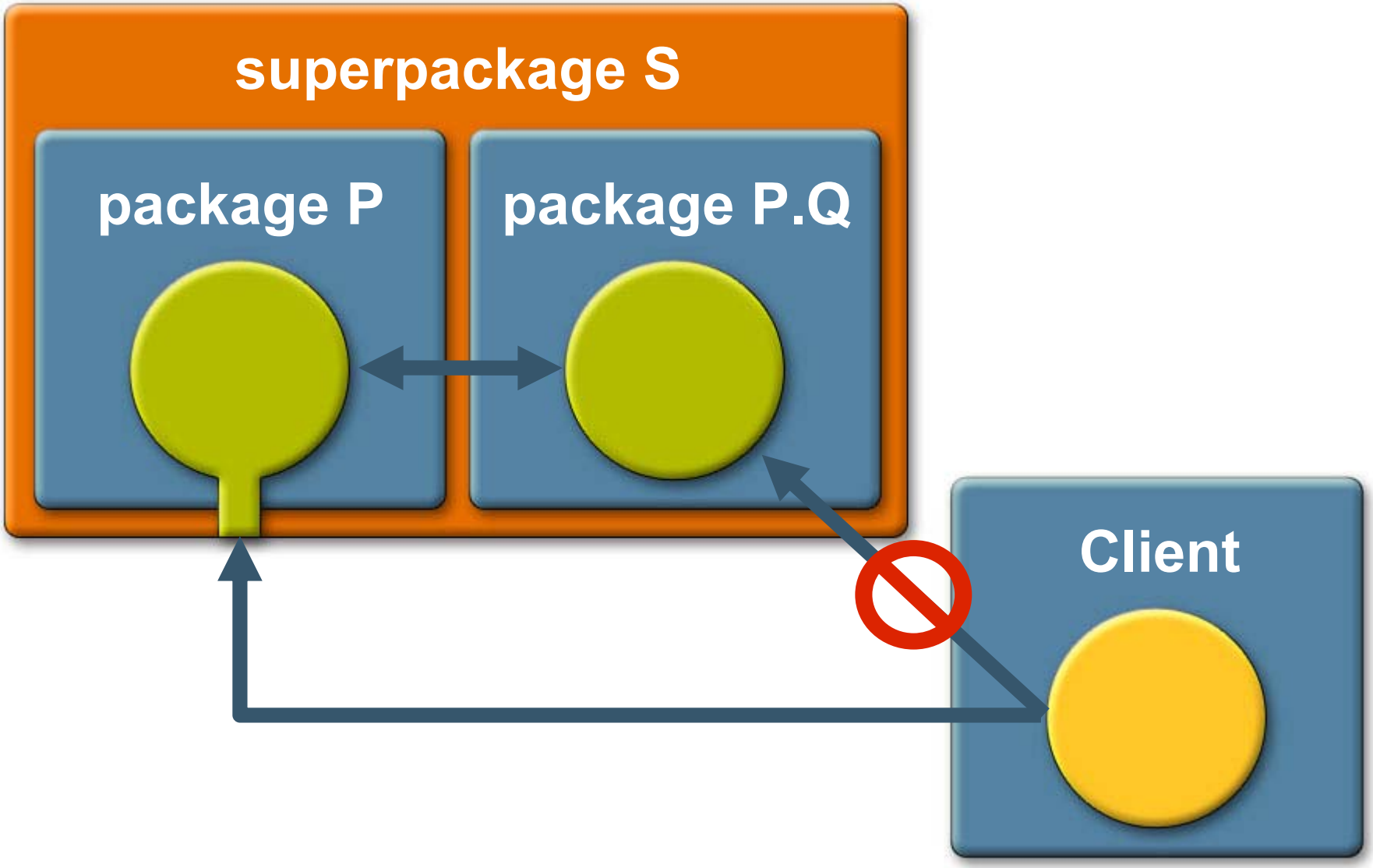
- A package(!)
- A namespace
- A type

Example

```
superpackage jdk {

  member package java.util;
  member package java.io;
  member package sun.io;    // Impl detail

  export java.util.*;       // Public API
  export java.io.*;
}
```


superpackage S
package P
package P.Q
Client

Superpackages at Compile-Time

- ***super-package.java* declares which types belong to a superpackage**
- **.java files do not** declare which superpackage they belong to

/* Run javac here */

- ***.spkg* file declares which types belong to a superpackage**
- **.class files declare which superpackage they belong to**

Superpackages at Run-Time

- **Within a superpackage, accessibility is as today**
- **Outside a superpackage, the Java Virtual Machine consults `.spkg` file to determine accessibility**
- **If member is public and exported, then accessible**
- **Access control checks are orthogonal to integrity checks performed by a module system**
  - **Can circumvent access control by hacking `.class`**

The terms “Java Virtual Machine” and “JVM” mean a Virtual Machine for the Java™ platform.

Nested Superpackages

- Superpackages can contain superpackages
- Useful in large projects
  - Information hiding between internal components
- A nested superpackage can be exported
  - Its exported types are accessible outside the enclosing superpackage
  - Types from non-exported superpackages are only accessible within the enclosing superpackage
- Multiple levels of nesting possible

Example

```
superpackage foo {
  member superpackage foo.xml, foo.net;
  export superpackage foo.xml;
}


superpackage foo.xml member foo {
  member package foo.xml.dom,
                 foo.xml.dom.utils;
  export foo.xml.dom.Factory;
}
```

Impact of Superpackages

- New classfile attribute
  - Must work with serialization
- Reification of superpackages in java.lang.reflect
  - Must perform access control checks
- javax.lang.model
  - JSR 269 Pluggable Annotation Processing
- Documentation and instrumentation
  - javadoc and javap
- **`com.sun.source.tree`**
  - Sun's compiler Tree API

Miscellenia

- Name/location of `super-package.java`
- Text or binary format of `.spkg` file
- Context-sensitive “keywords”
  - `member/export/superpackage`
- Annotations allowed in superpackage declaration
- A class belongs to at most one superpackage
- Superpackage namespace is distinct from (and is not obscured by) type and package namespaces

Agenda

Modularity

Information Hiding

Superpackages

Package Interfaces

JSR 294

Q&A

Package Interfaces

- A class can have an interface, but a package cannot
- Javadoc™ says which classes to use in a package
- Superpackages restrict public types, but types are still fundamentally organized as packages
- A package interface makes intentions clear

Separate Compilation Is Not Very Separate

- It is impossible to avoid referencing a class *somewhere in a Java platform program*
  - *To use* **`new`**
  - *To invoke a static factory method*
- *Compilation requires class definitions*
- *Class definitions may not always be available*
- *Dummy classes with empty methods are dull*
- *A package interface provides the type information needed for separate compilation*

Example

```
package interface P;
import Z.*;

class C implements I {
  public C(int i);
  protected Object f;
  String m()
    throws Exc;
}
```

```
package A;
import P.*;

class Client {
  C c = new C(5);
  ... c.f ...
  try { c.m(); }
  catch (Exc e) {..}
}
```

Agenda

Modularity

Information Hiding

Superpackages

Package Interfaces

JSR 294

Q&A

JSR 294

- “Improved Modularity Support in the Java Programming Language”
- Expert group discussion started in March 2007
  - General agreement on superpackage direction
  - Mailing list is publicly readable
- Membership ensures coherence with JSR 277 (Java Module System) and JSR 291 (OSGi)
- Scheduled for inclusion in Java Platform, Standard Edition 7 (Java SE 7)
  - Early access implementation later this year
  - Open source via OpenJDK

JSR 294 Expert Group Membership

Bryan Atsatt	Oracle (and 277, and 291)
Alex Buckley	Sun
Michal Cierniak	Google (and 277)
Matthew Flatt	University of Utah
Doug Lea	SUNY Oswego (and 277)
Glyn Normington	IBM (and 277, and 291)
Andreas Sterbenz	Sun
Eugene Vigdorchik	Jetbrains

Deployment Aspects

- Superpackages work just fine with existing deployment mechanisms
  - Java Archive (JAR) file, WAR, EAR files; Applets, JNLP/WebStart, etc.
- Even more interesting when combined with support for deployment modules
- Deployment modules defined by JSR 277
  - Superpackage forms basis of deployment module
  - Uses members, exports, metadata information
- Other deployment technologies also expected to take advantage of superpackages

Summary

- Superpackages are an effective mechanism for information hiding in the Java programming language
- Extension of the familiar access control model
- Superpackages support the Java Module System
- Scheduled to arrive in Java SE 7

For More Information

- TS-2318
  - “JSR 277: Java Module System”
- BOF-2400
  - “JSR 277 and JSR 294”
- JSR main page
  - http://jcp.org/en/jsr/detail?id=294
- Expert group mailing list archive and observer list
  - http://cs.oswego.edu/mailman/listinfo/jsr294-modularity-observer
- Blogs
  - http://blogs.sun.com/abuckley/
  - http://blogs.sun.com/andreas/

Q&A

Java Language Modularity With Superpackages

Alex Buckley

JSR 294 Co-spec lead
Sun Microsystems

Andreas Sterbenz

JSR 294 Co-spec lead
Sun Microsystems

TS-2401