Superpackages: Development Modules in Dolphin

Gilad Bracha

Computational Theologist
Sun Microsystems

TS-3885

Agenda

Modules: Development vs. Deployment
Information Hiding
Module Files
Separate Compilation
Conclusions

Agenda

Modules: Development vs. Deployment

Information Hiding

Module Files

Separate Compilation

Conclusions

Development Modules vs. Deployment Modules

What's the Distinction?

- Development modules
  - A language construct
  - Require direct VM support to enforce semantics (access control)
- Deployment modules
  - Unit of packaging and distribution
  - Require extensive tool and library support, but not necessarily language or VM support
- Concepts do interact, but the interface between them is relatively narrow

Deployment Modules

JSR 277: Java™ Module System

- Handled by JSR 277, whose concerns include:
  - Versioning
    - Version number schemes
    - How to run several versions of the same module side by side in the same VM
  - Distribution and packaging
    - JAR files and/or alternative formats
  - Module interconnect
    - Dynamic module connectivity
  - Repositories
    - System administration, security, loading/performance
  - More ...

Deployment Modules

JSR 277

- Hard problems
- In ideal world, a comprehensive solution at language and run time levels covers everything
- In reality, a longstanding open research issue
  - Development time issues handled with conservative, simple language constructs
  - Deployment issues handled by tools on top of reflective run time API

Agenda

Modules: Development vs. Deployment

Information Hiding

Module Files

Separate Compilation

Conclusions

Problem #1: Information Hiding

Today, If One Develops a System Made of Several Subsystems, One Has Two Choices:

- Put all the code in one package
  - Unwieldy
  - Exposes subsystem internals to each other
    - Sometimes necessary
    - Often harmful
- Put each subsystem in its own package
  - Cannot grant subsystems privileged access without excess publicity

Strawmen

- Don’t document
- Static classes
- A modest proposal

Strawmen

- **Don’t document**
- Static classes
- A modest proposal

Don’t Document

The Ostrich Solution

- Define subpackages as convenient
- Define all APIs needed outside of any package as public
- Don’t describe them with the Javadoc™ tool
- Pray and repeat
  - All the problems of access and dependence on APIs intended to be private
  - Does not protect from maliciousness or imbecility
  - Witness com.sun.*

Strawmen

- Don’t document
- **Static classes**
- A modest proposal

Static Classes

The Clever Ostrich Solution

- Define one package
- Define subsystems as top-level classes in said package
- Define top level classes of each subsystem as static nested classes of top level classes
- Sort of works, but
  - Only one level deep
  - Ugly, especially name mangling at the binary level
  - Very little VM level protection

Information Hiding with Static Classes

```
package superpackage;

    class Subsystem1 {
        public static class PublicClass1{...}
    // Public to the world
        private static class PrivateClass1{ ... }
    // Private to Subsystem1 at language level
    // - but at binary level it is
    // package private to superpackage
    }

    class Subsystem2 {
        public static class PublicClass2{...}
        private static class PrivateClass2{ ... }
    }
```

Information Hiding with Static Classes

```
package superpackage;

    class Subsystem1 {
        private static class PrivateClass1{
            int subsystem1Method1(){...}
        // Intended to be private to Subsystem1, but really
        // package private to superpackage
            private int subsystem1Method2() {...}
        // this works, until you want to inherit it
     }

    }

    class Subsystem2 {
        public static class PublicClass2{...}
        private static class PrivateClass2{ ... }
    }
```

Information Hiding with Static Classes

```
package superpackage;

    class Subsystem1 {
        public static class SubSubsystem11{...}
        private static class SubSubsystem12{
            private static class PrivateClass12{...}
        // Still accessible  to SubSubystem11
        // Still in the same compilation unit
         }
    }
```

Information Hiding with Static Classes

- Too complex
- Doesn't nest
- VM protection not exactly what you expect

Strawmen

- Don’t document
- Static classes
- **A modest proposal**

Defining a Superpackage

```
super package com.sun.myModule {

    export com.sun.myModule.myStuff.*;
    export com.sun.myModule.yourStuff.Interface;

    com.sun.myModule.myStuff;
    com.sun.myModule.yourStuff;
    com.sun.SomeOtherModule.theirStuff;
    org.someOpenSource.someCoolStuff;

}
```

Superpackages May Nest

```
super package mySystem {
    export mySubsubsystem11.PublicType111;

    mySubsystem1;
    mySubsystem2;
}

super package mySubsystem1 {
    export mySubsubsystem11.PublicType111,
           mySubsubsystem11.SemiPublicType112,
           mySubsubsystem12.SemiPublicType121;

    mySubsubsystem11;
    mySubsubsystem12;
}
```

Agenda

Modules: Development vs. Deployment

Information Hiding

Separate Compilation

Module Files

Conclusions

Module Files

Don't Take "File" Too Literally

- The authoritative binary definition of a module
  - Membership
  - Imports
  - Exports
  - Metadata
- Class files can claim membership in a module
- Claims must be cross checked with module file
- VM uses membership and export info to enforce access control

Module Files (cont.)

- Other information is useful for JSR 277
  - For example, import information can be used to validate configurations
- A module file corresponds to (part of) a JSR 277 module definition
  - Multiple module instances can coexist at runtime

Agenda

Modules: Development vs. Deployment

Information Hiding

Module files

Separate Compilation

Conclusions

Problem #2: Separate Compilation

Compilation Units Today Consist of Implementations

- Sometimes one doesn't have the implementation handy
  - Haven't built it yet
  - Another developer hasn't handed it to me yet
- Needed to be able to compile against the interface of another "module"
  - Workaround is ugly and tedious: declare phony implementation

Separate Compilation

```
package fully.qualified.packageName;

public class C implements fully.qualified.interface {
    public String someMethod(){ // fake body
              return nil; // fake return statement
    }
    public C(int i){}; // fake body
    protected Object aFieldName;
}
```

Separate Compilation: Definition

```
package interface fully.qualified.packageName;

// implicitly public types and members
class C implements fully.qualified.interface {
    String someMethod();
    C(int i);
    protected Object aFieldName;
}
```

Separate Compilation: Usage

```
package   another.packageName;

import fully.qualified.packageName;

// Code as usual – exactly as if imported package exists
}
```

Superpackages May Nest: Revised

```
package interface mySystem {
    public class mySubsubsystem11.PublicType111;
}

package interface mySubsystem1 {
    public class mySubsubsystem11.PublicType111;
    public class mySubsubsystem11.SemiPublicType112;
    public class mySubsubsystem12.SemiPublicType121;
}
```

Superpackages May Nest: Revised

```
super package mySystem {

    export mySystem.*;

    mySubsystem1;
    mySubsystem2;
}

super package mySubsystem1 {

    export mySubsystem.*;

    mySubsubsystem11;
    mySubsubsystem12;
}
```

Agenda

Modules: Development vs. Deployment

Information Hiding

Module Files

Separate Compilation

Conclusions

Summary

- Java Platform 7 will:
  - Provide flexible information hiding
  - Likely provide true separate compilation
- Language level module constructs (JSR 294)
- Deployment level module system (JSR 277)

For More Information

Useful Links

- gilad.bracha@sun.com
- http://jcp.org/en/jsr/detail?id=294
- http://jcp.org/en/jsr/detail?id=277
- http://blogs.sun.com/gbracha/

Q&A

Superpackages: Development Modules in Dolphin

Gilad Bracha

Computational Theologist
Sun Microsystems

TS-3885