JAVA

Modules
Modules

• a module
  − explicitly defines what is provided but also what is \textit{required}

• why?
  − the \textit{classpath} concept is “fragile”
  − no encapsulation
Modules

- a module explicitly defines what is provided but also what is required

- why?
  - the classpath concept is "fragile"
  - no encapsulation

**GRAPH THEORY FOR GEEKS**
Modular apps – motivation

• why
  – applications get more complex
  – assembled from pieces
  – developed by distributed teams
  – complex dependencies
  – good architecture
    • know your dependencies
    • manage your dependencies
Modular apps – motivation

• Version 1.0 is cleanly designed...
Modular apps – motivation

- Version 1.1...a few expedient hacks...we'll clean those up in 2.0
Modular apps – motivation

- Version 2.0...oops...but...it works!
Modular apps – motivation

- Version 3.0...Help! Whenever I fix one bug, I create two more!
Modular apps – motivation

- Version 4.0 is cleanly designed. It's a complete rewrite. It was a year late, but it works...
Modular apps – motivation

- Version 4.1...does this look familiar?....
Module declaration

• module-info.java
  
  module com.foo.bar {
    requires com.foo.baz;
    exports com.foo.bar.alpha;
    exports com.foo.bar.beta;
  }

• modular artifact
  - modular JAR – JAR with module-info.class
  - a new format JMOD
    • a ZIP with classes, native code, configuration,...
Modules and JDK

- JDK std library modularized too
  - java.base – always „required“

```java
module java.base {
    exports java.io;
    exports java.lang;
    exports java.lang.annotation;
    exports java.lang.invoke;
    exports java.lang.module;
    exports java.lang.ref;
    exports java.lang.reflect;
    exports java.math;
    exports java.net;
    ...
```
Module readability & module path

• When one module depends directly upon another

Module *reads* another module (or, equivalently, second module is *readable* by first)

• *Module path* – equivalent to classpath
  – but for modules
    • -p, --module-path

  – running application
    `java -p <module_path> name_of_module/name_of_class`
module com.foo.app {
    requires com.foo.bar;
    requires java.sql;
}

![Module graph diagram]
Accessibility

- If two types S and T are defined in different modules, and T is public, then code in S can access T if:
  - S’s module reads T’s module, and
  - T’s module exports T’s package
Implied readability

- Readability is not transitive
  - example:

```java
java.sql.Driver {
    java.util.Logger getParrentLogger();
    ...
}
```

```java
module java.sql {
    requires public java.logging;
    requires public java.xml;
    exports java.sql;
    exports javax.sql;
    exports javax.transaction.xa;
}
```
module com.mysql.jdbc {
    requires java.sql;
    requires org.slf4j;
    exports com.mysql.jdbc;
    provides java.sql.Driver with com.mysql.jdbc.Driver;
}

module java.sql {
    requires public java.logging;
    requires public java.xml;
    exports java.sql;
    exports javax.sql;
    exports javax.transaction.xa;
    uses java.sql.Driver;
}
Qualified exports

• module java.base {

    ...

    exports sun.reflect to
    java.corba,
    java.logging,
    java.sql,
    java.sql.rowset,
    jdk.scripting.nashorn;

}

• not intended for common usage
requires static

- required at compile time, but is optional at runtime

```java
module com.foo.bar {
    requires static com.foo.baz;
}
```

- WARNING
  - code that uses required static package has to be prepared for unavailability
• before Java 9, anything can be accessed via reflection
  – even private members
• in Java 9+, reflection follows rules of modules
• but – packages can be opened

```java
module com.foo.bar {
    opens com.foo.bar.alpha;
}
```

• types in opened package are accessible at runtime

```java
open module com.foo.bar {
}
```

– opens all its packages
• **opens** package **to** list-of-modules

  - opens to code in the listed modules only
package java.lang.reflect;

public final class Module {
    public String getName();
    public ModuleDescriptor getDescriptor();
    public ClassLoader getClassLoader();
    public boolean canRead(Module source);
    public boolean isExported(String packageName);
    ...
}

Layer

- layer – instantiation of module graph at runtime
- maps each module in the graph to the unique class loader

- layers can be stacked
  - a new layer can be built on top of another one
    - a layer’s module graph can be considered to include, by reference, the module graphs of every layer below it

- boot layer
  - created by VM at startup

- layers intended for app. servers, IDEs,...
Compatibility with “old” Java

• Classpath still supported
  – in fact – modules are “optional”

• Unnamed module
  – artefacts outside any module
    • “old” code
  – reads every other module
  – exports all of its packages to every other module
Automatic module

- a named module that is defined implicitly
  - it does not have a module declaration

- “regular” JAR placed on the module path rather than the class path
  - JAR without module-info.java
JAVA

Scripting API
Overview

• support of scripting languages directly from Java
  – integrating scripts to a Java program
  – calling scripts
  – using Java objects from a script
    • and vice-versa
  – ...

• since Java 6 directly part of JDK
  – JavaScript engine is also part of JDK
    – Java 6-7 – Mozilla Rhino engine
    – Java 8 – Nashorn engine
      • an implementation of JavaScript language in Java
    – since Java 11 – Nashorn deprecated
      – will be removed without replacement
        • but the Scripting API remains
  – there are many implementations for other languages
    • used via the ServiceLoader
Why

- a unified interface for all scripting languages
  - previously, every implementation has its own interface
- easy usage of scripting languages
  - variable “without” types
  - automatic conversions
  - ...
  - no need to compile programs
    - a “shell” can be used
- usage
  - complex configuration files
  - an interface for the application admin
  - extending an application (plugins)
  - scripting in an applications
    - as JS in a browser, VBScript in Office,...
Usage

• package javax.scripting
• ScriptEngineManager
  – a core class
  – obtaining an instance of a script engine
• basic usage
  – an instance of ScriptEngineManager
  – obtaining a particular engine
  – running a script using the eval() method
public class Hello {
    public static void main(String[] args) {
        ScriptEngineManager manager =
            new ScriptEngineManager();
        ScriptEngine engine =
            manager.getEngineByName("JavaScript");
        //ScriptEngine engine =
        //    manager.getEngineByExtension("js");
        //ScriptEngine engine =
        //    manager.getEngineByMimeType("application/javascript");
        try {
            engine.eval("println("Hello World!")");
            System.out.println(
                engine.eval("'Hello World again!'"));
        } catch(ScriptException e) { ... }
    }
}
Overview

- **script**
  - a String or char stream (a reader)
  - evaluation via ScriptEngine.eval()

- **interface Compilable**
  - its implementation is optional
    - has to be tested – instanceof Compilable
  - a compilation of a script into byte-code

- **interface Invocable**
  - its implementation is optional
    - has to be tested – instanceof Invocable
  - calling methods and functions of a script

- **Bindings, ScriptContext**
  - environment for script execution
    - mapping variables shared between Java and a script
Obtaining an engine

- `ScriptEngineManager.getEngineFactories()`
  - a list of all `ScriptEngineFactory`

```java
for (ScriptEngineFactory factory : engineManager.getEngineFactories()) {
    System.out.println("Engine name: "+ factory.getEngineName());
    System.out.println("Engine version: "+ factory.getEngineVersion());
    System.out.println("Language name: "+ factory.getLanguageName());
    System.out.println("Language version: "+ factory.getLanguageVersion());
    System.out.println("Engine names:");
    for (String name : factory.getNames()) {
        System.out.println("   " + name);
    }
    System.out.println("Engine MIME-types:");
    for (String mime : factory.getMimeTypes()) {
        System.out.println("   " + mime);
    }
}
```
Obtaining an engine

- ScriptEngineFactory.getEngine()

- or directly

- ScriptEngineManager.getEngineByName()
- ScriptEngineManager.getEngineByExtension()
- ScriptEngineManager.getEngineByMimeType()
Scripts

• evaluating a script
  - Object ScriptEngine.eval( String s, ...)
  - Object ScriptEngine.eval( Reader r, ...)

• passing variables (a basic variant)
  - void ScriptEngine.put(String name, Object value)
  - Object ScriptEngine.get(String name)
  - WARNING: be aware of type conversions
Passing variables

- interface Bindings
  - extends Map<String, Object>
  - a basic implementation – SimpleBindings
- interface ScriptContext
  - an environment, in which scripts run
  - a basic implementation – SimpleScriptContext
  - contains scopes
    - scope = Binding
  - special scopes
    - ENGINE_SCOPE – local for ScriptEngine
    - GLOBAL_SCOPE – global for EngineManager
  - getAttribute(..) / setAttribute(..) corresponds to getBindings(..).get / put
  - std Reader and Writer (input/output) for a script can be set
Passing variables

Calling functions/methods

- interface Invocable
  - optional, has to be tested (instanceof)
  - offers
    - calling script functions from Java code
    - calling script objects' methods from Java code (in a case of object oriented script)
    - implementing a Java interface by functions (methods) of a script

```java
ScriptEngine engine = manager.getEngineByName("javascript");
Invocable inv = (Invocable) engine;

engine.eval("function run() { println('function run'); };"forgee);
Runnable r = inv.getInterface(Runnable.class);
(new Thread(r)).start();

engine.eval("var runobj = { run: function()
    { println('method run'); } } };"forgee);
o = engine.get("runobj");
r = inv.getInterface(o, Runnable.class);
(new Thread(r)).start();
```
JavaScript engine in JDK

- some functions removed (or substituted)
  - mostly from security reasons
- integrated functions for import of Java packages
  - importPackage(), importClass()
    - packages accessible via Packages.PackageName, shortcuts (variables) defined for the most used packages: java (equivalent to Packages.java), org, com,...
    - java.lang is not imported automatically (possible conflicts of objects Object, Math,...)
    - since Java 8 it is necessary to first use
      `load("nashorn:mozilla_compat.js");`
- `JavaImporter` object
  - for “hiding” imported elements to variables (to avoid conflicts)
    ```javascript
    var imp = new JavaImporter( java.lang, java.io);
    ```
JavaScript engine in JDK

- Java objects in js
  - creating as in Java
  - var obj = new Clazz( ...)

- Java arrays in js
  - created via Java reflection
  - var arr = java.lang.reflect.Array.newInstance( ..)
  - then used commonly: arr[i], arr.length,...
    ```java
    var a = java.lang.reflect.Array.newInstance( java.lang.String, 5);
    a[0] = "Hello"
    ```

- anonymous classes
  - anonymous implementation of a Java interface
    ```java
    var r = new java.lang.Runnable() {
      run: function()
      {
        println( "running...");
      }
    };
    var th = null;
    th = new java.lang.Thread( r);
    th.start();
    ```
• anonymous classes (cont.)
  – auto-conversion of a function to an interface with a single method

```java
function func() {
    print("I am func!");
}

th = new java.lang.Thread( func);
th.start();
```
overloaded Java methods
  - reminder
    overloading “resolved” at compile time (javac)
  - when JavaScript variables passed to Java methods, the script engine selects the right variant
  - selection can be influenced
    - object[“method_name(parameter_types)”](parameters)
    - warning! string without spaces!
Other engines

- many existing engines
  - awk, Haskell, Python, Scheme, XPath, XSLT, PHP,...

- creating own engine
  - implementing API
    - at least necessary to implement
      - ScriptEngineFactory
      - ScriptEngine
    - declaring implementation of the javax.script.ScriptEngineFactory
      - for the ServiceLoader