JAVA

Classes and classloader

Overview

- classes are loaded to VM dynamically
  - it can be changed
    - from where they are loaded
    - how they are loaded
- java.lang.ClassLoader
  - VM uses classloaders to load classes
- each class is loaded by a classloader
  - Class.getClassLoader()
- exception
  - classes for arrays
    - created automatically when they are necessary
    - Class.getClassLoader() returns the same class as for elements of the array

Steps in class loading into VM

1. class loading
   - classloader
   - can cause exceptions (extending LinkageError)
     - ClassCircularityError
     - ClassFormatError
     - NoClassDefFoundError
   - can cause also OutOfMemoryError
2. "linking"

Steps in class loading into VM

2. "linking"
   - verification
     - a test, whether the class corresponds with the Java Virtual Machine Specification
     - exceptions (extending LinkageError)
       - VerifyError
   - preparation
     - creation of static fields
       - no initialization yet
     - OutOfMemoryError
   - resolution
     - symbolic references to other classes
     - exceptions – IncompatibleClassChangeError a potomci
       - IllegalAccessError, InstantiationError, NoSuchFieldError, NoSuchMethodError, UnsatisfiedLinkError

Steps in class loading into VM

3. initialization
4. creation of a new instance

Class and classloader

- a class in VM is identified by its name AND by its classloader
  - the same class loaded by different classloaders => two different classes from the view of VM

- each classloader has a parent (not in the sense of inheritance)
  - exception
    - bootstrap classloader
      - has no parent
  - if not specified => the system classloader
- during loading, the classloader first delegates class loading to its parent and only if the parent have not loaded the class, then it loads the class

Class and classloader

- other classes, necessary during a class loading, are loaded by the **same** classloader

Own classloader

- extends java.lang.ClassLoader
  - overrides the method findClass()

```
class MyClassLoader extends ClassLoader {
   public Class<?> findClass(String name) {
     byte[] b = loadClassData(name);
     return defineClass(name, b, 0, b.length);
   }

   private byte[] loadClassData(String name) {
     . . .
   }
}
```

- usage

```
Class.forName("clazz", true, new MyClassLoader());
```

or

```
new MyClassLoader().loadClass("clazz");
```

Own classloader

- it is possible to override loadClass()
  - not recommended
  - method does
    - findLoadedClass()
      - test, whether the class has been already loaded
    - delegates loading to the parent classloader
    - findClass()
    - resolveClass()

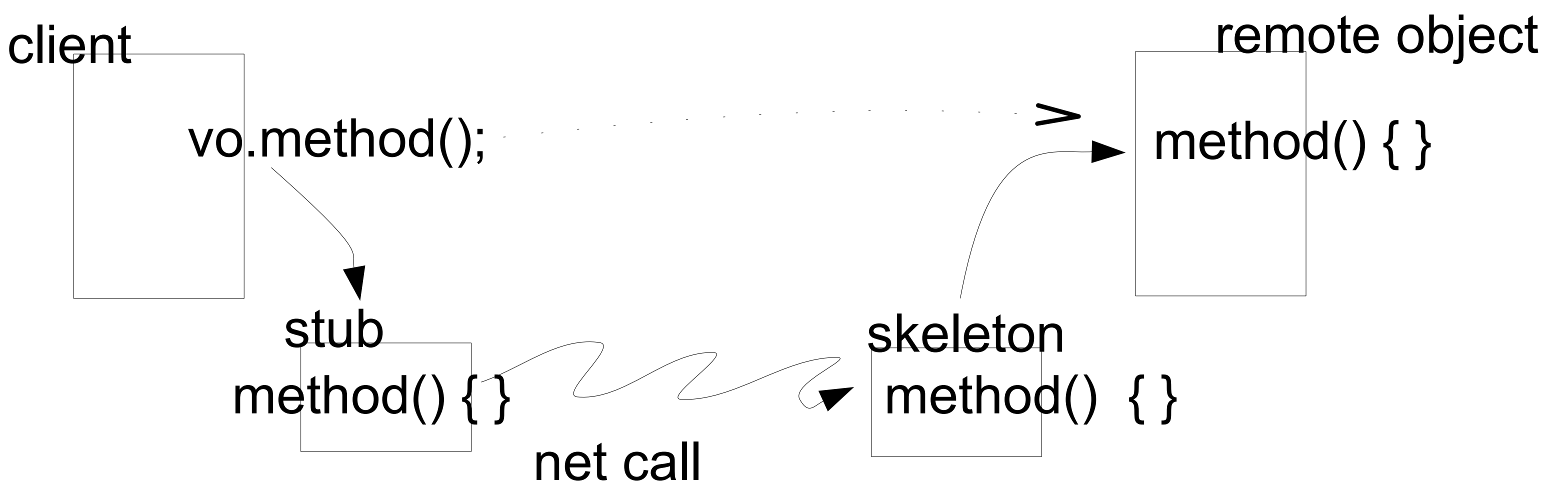
Examples of usage

- class loading from different sources – e.g. net

```
private byte[] loadClassData(String name) throws
  ClassNotFoundException {
  URL url = new URL("........");
  URLConnection con = defURL.openConnection();
  InputStream is = con.getInputStream();
  ByteArrayOutputStream bo = new ByteArrayOutputStream();
  int a = is.read();
  while (a != -1) {
    bo.write(a);
    a = is.read();
  }
  is.close();
  byte[] ar = bo.toByteArray();
  bo.close();
  return ar;
}
```

Examples of usage

- RMI
  - automatic loading of stubs over net



Examples of usage

- separation of name spaces
  - *(since Java 9 – better solution to use modules)*
  - e.g. application server
    - a single VM
    - "applications" are loaded by own classloaders
      - can use different version of libraries
        - (different classes with the same names)
  - problems if the “applications” want to communicate directly
    - solution – for communication, interfaces and classes loaded by a common parent classloader are used
    - non-ideal solution – via the reflection API

Loading other resources

- a classloader can load “anything”
- the same rules as for loading classes

- methods

```
URL getResource(String name)

InputStream getResourceAsStream(String name)

Enumeration<URL> getResources(String name)
```

Java

Service loader & provides

Overview

- let us have an interface for a “service”
  - e.g. javax.xml.parsers.DocumentBuilderFactory
- different “providers” of the service implementations
  - if we would like to use a particular implementation in a program, it would be necessary to use its name directly in source code
    - change of the implementation => change of code
  - better => use ServiceLoader<?>

Usage

- pack the implementation to JAR
  - add a file META-INF/services/service.interface.name e.g: `META-INF/services/javax.xml.parsers.DocumentBuilderFactory`
  - in the file, the name of a class implementing the interface per line
- in code
  - sl = ServiceLoader.load(interface.class, classloader);
  - sl.iterator()

- ServiceLoader is available since JDK 6
  - META-INF/services is used since JDK 1.3
    - it was necessary to create own ServiceLoader

- explicitly supported by modules (since Java 9)
  - definition in module-info.java

JAVA

Bytecode

Bytecode

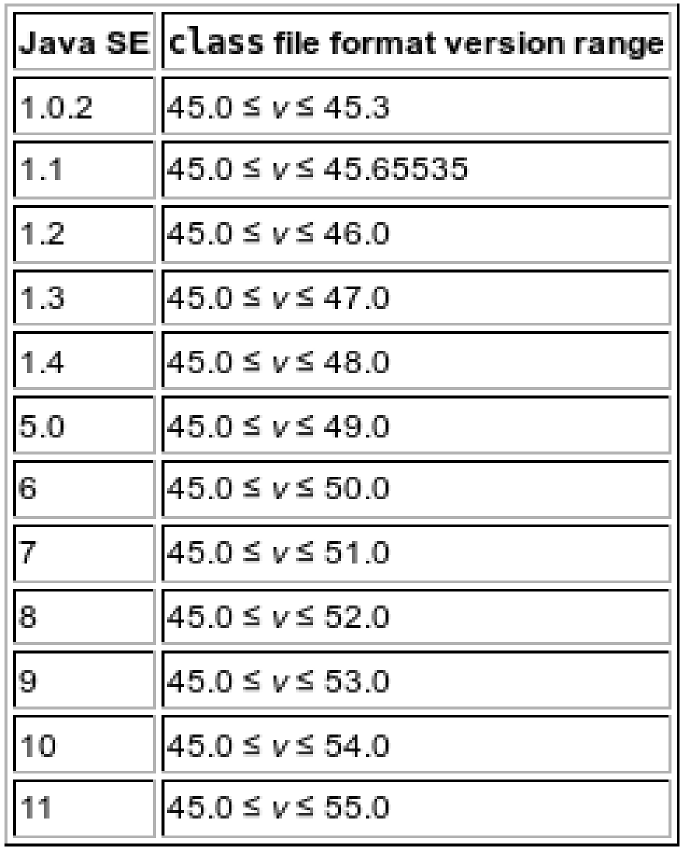
- The Java Virtual Machine Specification
- https://docs.oracle.com/javase/specs/

Format of the class file

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

Format of the class file

- magic
  - 0xCAFEBABE
- version



Java SE	class file format version range
1.0.2	45.0 ≤ v ≤ 45.3
1.1	45.0 ≤ v ≤ 45.65535
1.2	45.0 ≤ v ≤ 46.0
1.3	45.0 ≤ v ≤ 47.0
1.4	45.0 ≤ v ≤ 48.0
5.0	45.0 ≤ v ≤ 49.0
6	45.0 ≤ v ≤ 50.0
7	45.0 ≤ v ≤ 51.0
8	45.0 ≤ v ≤ 52.0
9	45.0 ≤ v ≤ 53.0
10	45.0 ≤ v ≤ 54.0
11	45.0 ≤ v ≤ 55.0

Code

- a stack-oriented assembler

```
void spin() {
    int i;
    for (i = 0; i < 100; i++) { ; }
}
```

```
Method void spin()
 0    iconst_0     // Push int constant 0
 1    istore_1     // Store into local variable 1 (i=0)
 2    goto 8       // First time through don't increment
 5    iinc 1 1     // Increment local variable 1 by 1 (i++)
 8    iload_1      // Push local variable 1 (i)
 9    bipush 100   // Push int constant 100
11    if_icmplt 5  // Compare and loop if less than (i<100)
14    return       // Return void when done
```

Instructions

- opcodes
- size 1 byte
  - max 256 possible instructions
    - not all of them are used
- instruction categories
  - load and store (aload_0, istore,...)
  - arithmetic and logic (ladd, fcmpl,...)
  - type conversion (i2b, d2i,...)
  - object creation and manipulation (new, putfield)
  - operand stack management (swap, dup2,...)
  - control transfer (ifeq, goto,...)
  - method invocation and return (invokespecial, areturn,...)

Instructions

- invokedynamic
  - since Java 7
  - support for compiling dynamic languages into Java bytecode

  - since Java 8 used also for compiling lambda expressions

Bytecode

- tools for bytecode manipulation
  - ASM, BCEL, SERP, ...
- ASM
  - http://asm.ow2.org/
    - bytecode manipulation
    - new classes creation
    - updating existing ones

JNI
(Java Native Interface)

Overview

- integration of native code (in C, C++,...) to Java
- integration of Java code to native code

- common usage
  - platform-specific operations

Example

```
class HelloWorld {
    public native void displayHelloWorld();

    static {
        System.loadLibrary("hello");
    }

    public static void main(String[] args) {
        new HelloWorld().displayHelloWorld();
    }
}
```

Example

- javac -h <output_dir> HelloWorld.java
    - (old way – javah -jni HelloWorld)
  - > HelloWorld.h

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class HelloWorld */
#ifndef _Included_HelloWorld
#define _Included_HelloWorld
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     HelloWorld
 * Method:    displayHelloWorld
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_HelloWorld_displayHelloWorld
  (JNIEnv *, jobject);
#ifdef __cplusplus
}
#endif
#endif
```

Example

- implementation of the native method

```
#include <jni.h>
#include "HelloWorld.h"
#include <stdio.h>

JNIEXPORT void JNICALL
Java_HelloWorld_displayHelloWorld(JNIEnv *env,
                                       jobject obj) {
    printf("Hello world!\n");
}
```

Example

- compilation of native code (HelloWorld.c)
  - result – shared library (.so, .dll)
  - UNIX

    -
      ```
      gcc -shared -Wall -fPIC -o libhello.so
      -I/java/include -I/java/include/linux HelloWorld.c
      ```
  - Windows

    -
      ```
      cl -Ic:\java\include -Ic:\java\include\win32
                      -LD HelloWorldImp.c -Fehello.dll
      ```
- run the program
  - java HelloWorld

Type mapping

Java Type	Native Type	Size in bits
boolean	jboolean	8, unsigned
byte	jbyte	8
char	jchar	16, unsigned
short	jshort	16
int	jint	32
long	jlong	64
float	jfloat	32
double	jdouble	64
void	void	n/a

Type mapping

- "non-primitive" types
  - jobject
  - jstring
  - jclass
  - jthrowable
  - jarray
  - jobjectArray
  - jbooleanArray
  - jintArray
  - .....Array

Accessing Strings

- **jstring** cannot be used directly
  - convert ot char*

```
Java_.....(JNIEnv *env, jobject obj, jstring prompt) {
  char *str = (*env)->GetStringUTFChars(env, prompt, 0);
  printf("%s", str);
  (*env)->ReleaseStringUTFChars(env, prompt, str);
  ...
```

  - new string

```
char buf[128];
...
jstring jstr = (*env)->NewStringUTF(env, buf);
```

Accessing arrays

```
Java_..._sumArray(JNIEnv *env, jobject obj,
                                     jintArray arr) {
  int i, sum = 0;
  jsize len = (*env)->GetArrayLength(env, arr);
  ...
  jint *body = (*env)->GetIntArrayElements(env,arr,0);
  for (i=0; i<len; i++) {
    sum += body[i];
  }
  ...
  (*env)->ReleaseIntArrayElements(env, arr, body, 0);
  return sum;
}
```

Accessing methods

```
JNIEXPORT void JNICALL
Java_..._nativeMethod(JNIEnv *env, jobject obj, jint i) {
  jclass cls = (*env)->GetObjectClass(env, obj);
  jmethodID mid = (*env)->GetMethodID(env, cls, "method", "(I)V");
  (*env)->CallVoidMethod(env, obj, mid, i);
}
```

- method signature
  - the **javap** tool
    - javap -s class

Signature	Java type
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double
L*class;*	class
[*type*	type []
(*parameters*)*return-type*	method

Accessing methods

- GetMethodID()
- GetStaticMethodID()

- CallVoidMethod()
- CallIntMethod()
- ....
- CallStaticVoidMethod()
- CallStaticIntMethod()
- ....

Accessing fields

- similar as methods

- fid = (*env)->GetStaticFieldID(env, cls, "si", "I");
- fid = (*env)->GetFieldID(env, cls, "s", "Ljava/lang/String;");

- si = (*env)->GetStaticIntField(env, cls, fid);
- jstr = (*env)->GetObjectField(env, obj, fid);

- SetObjectField
- SetStaticIntField
- ...

Exception handling

```
jmethodID mid = ....
...
(*env)->CallVoidMethod(env, obj, mid);
jthroable exc = (*env)->ExceptionOccurred(env);
if (exc) {
  (*env)->ExceptionDescribe(env);
  (*env)->ExceptionClear(env);
  ....
}
```

Supporting synchronization

```
...
(*env)->MonitorEnter(env, obj);
     synchronized block
(*env)->MonitorExit(env, obj);
...
```

- wait() and notify() are not directly supported
  - but can be called as any other method

C++

- C

  -
    ```
    jstring jstr = (*env)->GetObjectArrayElement
                                               (env, arr, i);
    ```
- C++

  -
    ```
    jstring jstr = (jstring) env->GetObjectArrayElement
                                                    (arr, i);
    ```

JNA
(Java Native Access)

Overview

- not a part of JDK
- https://github.com/java-native-access/jna

- automatic mapping between Java and native libraries
  - support for “all” platforms

- no need to manually create native code
  - JNI is used internally

JNA „Hello world“

```
import com.sun.jna.Library;
import com.sun.jna.Native;


public interface JNAApiInterface extends Library {
  JNAApiInterface INSTANCE = (JNAApiInterface)
Native.loadLibrary((Platform.isWindows() ? "msvcrt" :
"c"), JNAApiInterface.class);

  void printf(String format, Object... args);
}



public static void main(String args[]) {
  JNAApiInterface jnaLib = JNAApiInterface.INSTANCE;
  jnaLib.printf("Hello World");
}
```

Type mapping

Native Type	Size	Java Type
char	8-bit integer	byte
short	16-bit integer	short
w char_t	16/32-bit character	char
int	32-bit integer	int
int	boolean value	boolean
long	32/64-bit integer	NativeLong
long long	64-bit integer	long
float	32-bit FP	float
double	64-bit FP	double
char*	C string	String
void*	pointer	Pointer

Type mapping

- array (pointer) → array
- struct → … extends Structure
- by-reference arguments
  - char **bufp → PointerByReference bufp
  - int* lenp → IntByReference lenp
- …
- JNAerator
  - https://github.com/nativelibs4java/JNAerator
  - generates headers of Java methods