Advanced Programming

Class Loading
Reflection API
Annotations

Where Do Classes Come From?

- The Java compiler and JVM must “know” where to look for the classes needed by an application

```
class  MyClass {}

public class Main {
  public static void main(String args[]) {
    MyClass object = new MyClass();    //→ right here...
    Integer number = new Integer(0);   //→ java.lang
    Random random = new Random();      //→ java.util
    PDDocument doc = new PDDocument();//→ org.apache.pdfbox.pdmodel
  }                                                  in pdfbox-app.jar
}
```

- Classes are introduced into the Java environment when they are referenced by name in a class that is already running.

Packages

- Classes are grouped together in *packages*

```
package com.example.model;
public class Student { … }
```

```
package com.example.model;
public class Project { … }
```

- A package is implemented as a directory (folder) and provides a *namespace* to a class.

```
..somewhere into the file system
    \com
        \example
            \model
                \Student.java
                \Project.java
```

PROJECT-HOME\build\classes

```
→ Student.class
→ Project.class
```

- Standard classes are also organized in packages

```
    \java
        \util
            \Random.class
            \ArrayList.class
```

JAVA-HOME\lib\rt.jar

sources are in JAVA-HOME*src.zip*

Imports

- The *import* statement provides a way to identify classes that you want to reference in your class

```
Instead of: java.util.Random random = new java.util.Random();
```

The long name of the class

```
import java.util.Random;
...
Random random = new Random(); // ← the simple (short) name
```

- Importing an entire package: `import java.util.*;`
- No need to import *java.lang* or classes from the same package as the current class.
- The *import* statement only "tells" where to look for a class; it does not perform any kind of code inclusion.
- *Static imports* makes the static members of a class available under their simple name.

```
import static java.lang.Math.random;
double d = random();
```

CLASSPATH

- The *import* statement only tells the *sufix* of the file containing the source or the bytecode.

```
import com.example.model.Student;
    some place  …\com\example\model\Student.java
    other place …\com\example\model\Student.class
```

- In order to identify the exact location of the file, we need the *prefix* as well.

```
d:\java\MyProject\src\com\example\model\Student.java
d:\java\MyProject\build\classes\com\example\model\Student.class
```

- The CLASSPATH environment variable specifies the locations where to look for classes, above the location of the package hierarchy.

javac, java

- javac

  -**classpath** or -**cp** <path to other classes>

  -**sourcepath** <path to search for class definitions>

  -**d** <the destination directory for class files>

```
javac -sourcepath src -classpath classes;lib\pdfbox.jar
      src\com\example\model\Student.java -d classes
```

- java

  -**classpath** or -**cp** <path to other classes>

```
java -cp d:\java\MyProject\classes;d:\java\lib\pdfbox.jar
     com.example.app.MainClass
```

Classpath defines a list of **local** or **remote** directories or JAR archives.
This is where the classes are searched for.

The Lifecycle of a Class

- **Loading** - the process of locating the binary representation of a class and bringing it into the JVM.
  - load the definition of the bytecode from a **.class** file
  - create an instance of class *java.lang.Class*
  - *ClassNotFoundException* may occur
- **Linking** - the process incorporating a class (type) into the runtime state of the JVM.
- **Initialization** - the process of executing the *static initializers* of a type.
- **Unloading** - When a type becomes unreachable (i.e. the running application has no references to the type), it becomes eligible for garbage collection.

Class Loaders

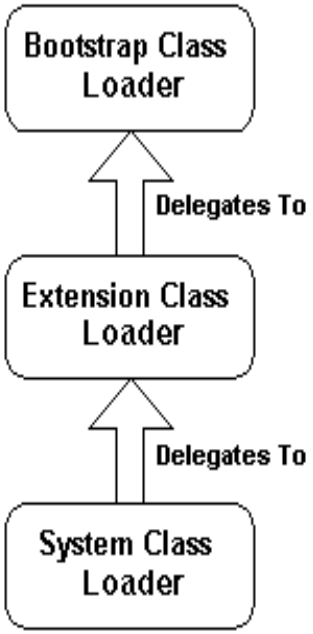
Objects responsible for loading classes.

- The JVM typically provides:
  - a *bootstrap* class loader, integral part of the JVM, responsible for loading the core Java classes (e.g. java.*, javax.*, etc.)
  - *user-defined* class loaders, subclasses of the abstrct class *java.lang.ClassLoader*
    - the *extension* class loader responsible for loading classes from the JRE's extension directories
    - the *system (or application)* class loader responsible for loading classes from the system class path.

      → *java.net.URLClassLoader*
- Every *Class* object contains a reference to the *ClassLoader* that defined it.

Delegation Model

- Class loaders are arranged hierarchically in a tree, with the bootstrap class loader as the root of the tree.
- The *ClassLoader* class uses a **delegation model** to search for classes and resources



When requested to find a class or resource, a ClassLoader instance will delegate the search for the class or resource to its parent class loader before attempting to find the class or resource itself.

If no class loader is successful in loading the type → ClassNotFoundException

Namespaces

A loaded class in a JVM is identified by its fully qualified name and its defining class loader - this is sometimes referred to as **the runtime identity of the class**.

Consequently, each class loader in the JVM can be said to define its own namespace. Within a namespace, all fully-qualified names are unique. Two different name spaces, however, can contain identical fully-qualified names.

Because the defining class loader is part of the runtime identity of the class, the classes associated with these names are considered distinct (e.g. class x.y.z.Foo defined by class loader A is NOT considered by the JVM to be the same class as class x.y.z.Foo defined by class loader B).

Dynamic Loading of a Class

- **Class.forName(String className)**

  Attempts to *locate, load, and link* the class or interface.

  Returns the *Class* object associated with the class or interface with the given string name, using the given class loader.

- Assume that the name of the class is known only at runtime (read from a properties file):

```
String driverName = "org.postgresql.Driver";
Class clazz = Class.forName(driverName);
// same as
ClassLoader loader = this.getClass().getClassLoader();
loader.loadClass(driverName);
```

Dynamic Instantiation of an Object

- **classObject.newInstance()**
- Creates a new instance of the class represented by this *Class* object. The class is instantiated as if by a new expression with an empty argument list.

```
Class clazz = Class.forName("java.awt.Button");

Button b = (Button) clazz.newInstance();
```

Example: File Shell Commands

```
public interface Command {
  void execute(String ... params) throws IOException;
}


public class Open implements Command {
  @Override
  public void execute(String... params) throws IOException {
    for (String param : params) {
      Desktop.getDesktop().open(new File(param));
    }
  }
}

public class Copy implements Command {
  @Override
  public void execute(String... params) throws IOException {
    String source = params[0];
    String target = params[1];
    Files.copy(Paths.get(source),Paths.get(target),REPLACE_EXISTING);
  }
}
```

Example: Testing the Commands

```
public class TestCommands {

  public static void main(String args[]) throws IOException {
    Scanner scanner = new Scanner(System.in);
    while (true) {
      String commandName = scanner.next();
      if (commandName.equalsIgnoreCase("exit")) {
          break;
      }
      String[] params = scanner.nextLine().trim().split("\\s+");
      try {
          // The command name is actually the class name
          Class clazz = Class.forName(commandName);
          Command command = (Command) clazz.newInstance();
          command.execute(params);
      } catch (ClassNotFoundException | InstantiationException |
               IllegalAccessException | IOException e) {
          System.err.println(e);
      }
    }
  }
}
```

URLClassLoader

- This class loader is used to load classes and resources from a search path of URLs referring to both JAR files and directories.
- We can dynamically configure it:

```
// Get the current class loader
URLClassLoader urlLoader =
    (URLClassLoader) this.getClass().getClassLoader();
// Add a new directory to the classpath
urlLoader.addURL(new File("c:\\commands").toURL());
// Load a class
urlLoader.loadClass("ro.info.uaic.shell.Open");
```

Reloading Classes

```
public class MyClassLoader extends URLClassLoader{
  public MyClassLoader(URL[] urls){
    super(urls);
  }
}
// Find the default URLs in the CLASSPATH
URLClassLoader systemLoader =
  (URLClassLoader) this.getClass().getClassLoader();
URL[] urls = systemLoader.getURLs();

// Load dynmically a class
MyClassLoader myLoader1 = new MyClassLoader(urls);
myLoader1.loadClass("ro.info.uaic.shell.Open");
…

// Modify, recompile and reload the class
myLoader1.loadClass("ro.info.uaic.shell.Open"); // → doesn't work!

// Create a new class loader
MyClassLoader myLoader2 = new MyClassLoader(urls);
myLoader2.loadClass("ro.info.uaic.shell.Open"); // → OK
```

The default class loaders always check to see if the class requested has already been loaded.

Reflection API

- Most of the types used by an application are statically specified during design time and their usage is verified by the compiler:

```
java.awt.Button button = new java.awt.Button("Hello");
button.setBackground(Color.YELLOW);
```

- There are situations when the name of a class, method, field etc. is not known at compile time.

```
String componentName = <read the name of an AWT component>;
Class clazz = Class.forName(componentName);
// How do we create the component and set its properties?
```

- *Reflection* - the ability to **inspect** or **use** types at runtime without knowing their names at design-time, instantiate new objects, invoke methods and get/set field values.

java.lang.reflect

- For every type, the JVM instantiates an immutable instance of *java.lang.Class* which provides methods to examine its runtime properties.
  - Class Name
  - Modifiers (public, final etc.)
  - Package
  - Superclass
  - Implemented Interfaces
  - Constructors
  - Methods
  - Fields
  - Annotations
- Each type property from the above list has a **corresponding class**: *Constructor, Method, Field*, etc.

Inspecting Classes

- Object.getClass()

```
Class c1 = "Hello World".getClass();
Class c2 = System.out.getClass();
```

- The .class and .TYPE Syntax

```
String.class; java.io.PrintStream.class;
double.class; Double.class; Double.TYPE;
```

- Class.isInterface(), isEnum(), isPrimitive(), isArray(),..

```
int buffer[] = new int[10];
System.out.println(buffer.getClass().isArray());
```

- Class.getSuperclass()

```
java.awt.Button.class.getSuperclass();
```

- Class.getInterfaces()

```
Arrays.toString(java.util.HashSet.class.getInterfaces())
```

- Class.getPackage() `Object.class.getPackage();`

...

Inspecting Members of a Class

- Constructors

  `getConstructors(), getDeclaredConstructors()`
  → `returns an array:` **`Constructor[]`**
- Methods

  `getMethods(), getDeclaredMethods()`
  → `returns an array:` **`Method[]`**
- Fields

  `getFields(), getDeclaredFields()`
  → `returns an array:` **`Field[]`**
- Inner Classes

  `getClasses(), getDeclaredClasses()`
  → `returns an array:` **`Class[]`**

The Declaring Class

`getDeclaringClasses(), getEnclosingClass()`

Inspecting Modifiers

- Classes, Methods, Fields etc. may have modifiers, like "public", "private", "static" etc.
- getModifiers() returns an integer encoding

```
int modifiers = java.io.String.getModifiers();

Modifier.isAbstract(int modifiers)
Modifier.isFinal(int modifiers)
Modifier.isInterface(int modifiers)
Modifier.isNative(int modifiers)
Modifier.isPrivate(int modifiers)
Modifier.isProtected(int modifiers)
Modifier.isPublic(int modifiers)
Modifier.isStatic(int modifiers)
Modifier.isStrict(int modifiers)
Modifier.isSynchronized(int modifiers)
Modifier.isTransient(int modifiers)
Modifier.isVolatile(int modifiers)
```

Instantiating Objects

- Class.newInstance()

```
Class clazz = Class.forName("java.awt.Point");
Point point = (Point) clazz.newInstance();
```

- Constructor.newInstance(Object … args)

```
Class clazz = java.awt.Point.class;
// Identify a specific constructor
Class[] signature = new Class[] {int.class, int.class};
Constructor ctor = clazz.getConstructor(signature);

// Prepare the arguments – they must match the signature
Integer x = new Integer(10);
Integer y = new Integer(20);

// Create the object
Point point = (Point) ctor.newInstance(x, y);
```

Invoking Methods

- Method.invoke(Object obj, Object … args)

```
Class clazz = java.awt.Rectangle.class;
Rectangle rectangle = new Rectangle(0, 0, 100, 100);

// Identify the method by its name and arguments
Class[] signature = new Class[] {Point.class};
Method method = clazz.getMethod("contains", signature);

// Prepare the arguments
Point point = new Point(10, 20);

// Invoke the method
method.invoke(rectangle, point);
```

Setting and Getting Fields

- Field.set(), Field.get()

```
Class clazz = java.awt.Point.class;
Point point = new Point(0, 20);

// Get the fields
Field xField = clazz.getField("x");
Field yField = clazz.getField("y");

// Set x
xField.set(point, 10);

// Get y
Integer yValue = yField.get(point);
```

- How about *private* (inaccesible) fields?

```
Field privateField = clazz.getDeclaredField("x");
privateField.setAccessible(true);
```

Working Dynamically with Arrays

- Array.newInstance()

```
Object a = Array.newInstance(int.class, 10);
a.getClass(); // class [I
a.getClass().isArray(); // true
a.getClass().getComponentType(); //int
```

- Array.getLength(), set(), get()

```
for (int i=0; i < Array.getLength(a); i++) {
    Array.set(a, i, new Integer(i));
}
for (int i=0; i < Array.getLength(a); i++) {
    System.out.print(Array.get(a, i) + " ");
}
```

java.beans.Introspector

- *Introspection* is the automatic process of analyzing a bean's design patterns to reveal the bean's properties, events, and methods.
- The *Introspector* class provides a standard way for tools to learn about the properties, events, and methods supported by a target Java Bean.

```
Class beanClass = java.awt.Button.class;

BeanInfo info = Introspector.getBeanInfo(beanClass);

for (PropertyDescriptor pd : info.getPropertyDescriptors()) {

        System.out.println(pd.getName());

}

// background, enabled, focusable, font, foreground, label, name, visible, ...
```

Drawbacks of Reflection

- **Performance Overhead -** Because reflection involves types that are dynamically resolved, certain JVM optimizations can not be performed. Consequently, reflective operations have slower performance than their non-reflective counterparts.
- **Security Restrictions -** Reflection requires a runtime permission which may not be present when running under a security manager.
- **Exposure of Internals -** Since reflection allows code to perform operations that would be illegal in non-reflective code, such as accessing private fields and methods, the use of reflection can result in unexpected side-effects, which may render code dysfunctional and may destroy portability.

Annotations

@Override, @Inject, @Entity, @HelloWorld,...

- A form of *metadata*, similar to *comments.*
- Unlike comments, annotations are *strong-typed*
- Provide data about a program that is not part of the program itself. Annotations have no direct effect on the operation of the code they annotate.
- Annotations can be applied to declarations of *classes*, *fields*, *methods*, and other program elements.
- Annotations are used for:
  - Information for the compiler
  - Compile-time and deployment-time processing
  - Runtime processing

Built-in Java Annotations

- @Deprecated

```
@Deprecated
/**
  @deprecated use MyNewFancyClass instead.
*/
public class MyOldUglyClass { … }
```

- @Override

```
public class HelloWorld {
    @Override
    String toString() { return "Hello World!"; }
```

- @SuppressWarnings

```
@SuppressWarnings("deprecation")
public void methodWithWarning() {
    // Questionable, deprecated things happen here
}
```

- @SafeVarargs, @FunctionalInterface

Declaring and Using Annotations

- Declaring annotations: **@interface**

```
/**
* This is a annotation declaration
* It contains four members (parameters)
*/
public @interface RequestForEnhancement {
    String request();
    String solicitor();
    boolean urgent() default false;
    String date() default "[unimplemented]";
}
```

Compiles to → RequestForEnhancement.class

- Using annotations

```
@RequestForEnhancement(
    request = "Implement this method in O(log n)",
    solicitor = "The Boss",
    date = "yesterday"
)
public void helloWorld() {
    System.out.println("Hello World!");
}
```

Special Types of Annotations

- Markers

```
public @interface Preliminary { } // No members (parameters)
...

@Preliminary public class TimeTravel { ... }
```

- Single-Member Annotations

```
public @interface Note { // Only one parameter called 'value'
    String value();
}
...

@Note("My super fast optimized QuickSort")
public void sort(int v[]) { ... }
```

- Meta-Annotations (annotations for annotations)

```
@Retention(RetentionPolicy.RUNTIME) // CLASS, SOURCE
@Target(ElementType.METHOD)
    //CONSTRUCTOR, FIELD, METHOD, PACKAGE, PARAMETER, TYPE)
```

Reflection API and Annotations Example

Create a framework for automated testing...

```
import java.lang.annotation.*;
/**
* We are going to use this annotation to mark methods.
* The marked methods will be subject to testing.
* It will only work for static methods with no parameters...
*/
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test { }
```

Example: The “Tested” Class

```
public class MyProgram {
    @Test public static void m1() { }

    public static void m2() { }

    @Test public static void m3() {
        throw new RuntimeException("Boom");
    }

    public static void m4() { }

    @Test public static void m5() { }

    public static void m6() { }

    @Test public static void m7() {
        throw new RuntimeException("Crash");
    }

    public static void m8() { }
    }
```

Example: “The Framework”

```
import java.lang.reflect.*;

public class MyTestFramework {
    public static void main(String[] args) throws Exception {
        int passed = 0, failed = 0;
        for (Method m : Class.forName(args[0]).getMethods()) {
            if (m.isAnnotationPresent(Test.class)) {
                try {
                    m.invoke(null);
                    passed++;
                } catch (Throwable ex) {
                    System.out.printf("Test %s failed: %s %n",
                        m, ex.getCause());
                    failed++;
                }
            }
        }
        System.out.printf("Passed: %d, Failed %d%n", passed, failed);
    }
}
```