Reflection

Week 4 Presentation 3

Reflection

- Generally speaking, "reflection" is your program asking the JVM what it knows about it – and the JVM knows a lot of things.
- As all this happens of course while the program is running, it allows for a lot of on-the-fly operations that would be impossible with a compiled program written in C, for instance.
- Reflection is considered rather advanced programming, but some of its features are frequently used, for instance with JDBC which is the standard Java way to access a database.

Reflection

- Works because of the JVM. Once again, it only works because of the JVM
- The JVM stores the description of the classes when it loads them
- The loading subsystem needs to read a lot of information to make the program runnable, and this information is stored and made available when the program runs

Reflection

- The JVM stores objects, of class **Class**, that describe every class used in the application

- Class called **Class**
  - *Metadata*
- The objects represent classes in the running application

- Class objects have no constructor – they are built by the JVM

Class objects

- There are two ways two retrieve class information from the JVM.

ClassName obj = new ClassName();

Method inherited from object

1. Method inherited from the object:

   The get class method of the object: obj.getClass()

2. The .class attribute when there is no object: ClassName.class

Static version

no constructor – built by the JVM

Uses of Reflection #1: Getting Class Names

```
class OuterClass {
        private int dummy;
        OuterClass(){}
}
public class MyClass {

        class InnerClass {
                private int dummy;
                InnerClass(){}
        }
public static void main(String[] args) {
        OuterClass obj = new OuterClass();
        System.out.println(obj.getClass().getName());
        System.out.println(InnerClass.class.getName());
}
```

For instance, you can retrieve class names.

```
$
$ java MyClass
OuterClass
MyClass$InnerClass
$
```

Uses of Reflection #2: Locating Files used by Your Program

- One common problem is locating files used by your program – the properties file to start with if there is one.
- Location of files read by your program
  - parameter file
  - data file
    multimedia, etc

When people click on an icon to launch your program, the idea of "current directory" becomes extremely hazy. If you want to start by reading a properties file, of if you want to display the logo of your company (an image) while initialization is going on, where should you look?

The default directory for installing programs varies from system to system (and don't forget that a Java application can run on Windows as well as on Linux or Mac OSX), and additionally users often have the option of installing software elsewhere than the default location. Your only hope to find out is to get it when the program runs.

Uses of Reflection #2: Locating Files used by Your Program

- As the loader knows where it got the .class from, you can just ask the JVM.
- **Solution** get location at runtime

Uses of Reflection #2: Locating Files used by Your Program

- As the loader knows where it got the .class from, you can just ask the JVM.

file:/Users/... ... .../Reflection.class

- **Solution** get location at runtime

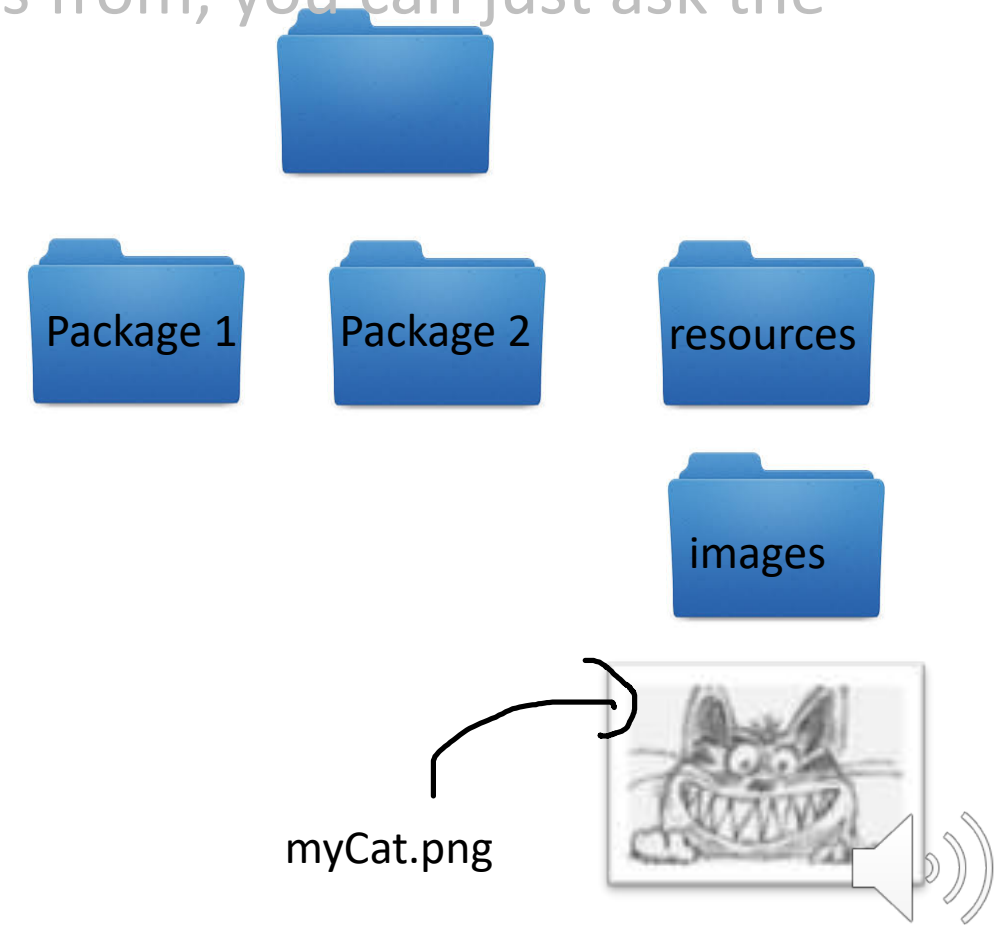
```
public class Reflection {

        public static void main(String[] args) {
                System.out.println(Reflection
                                .class
                                .getClassLoader() .getResource("Reflection.class") .toString());

        }
}
```

Uses of Reflection #2: Locating Files used by Your Program

- As the loader knows where it got the .class from, you can just ask the JVM.
- **Solution** get location at runtime




```
URL url = this
            .getClass()
             .getClassLoader()
             .getResource("resources/images/myCat.png");
```

Uses of Reflection #3: Reading Annotations

- We saw in the previous presentation that annotations could be read by a program, it's through reflection.
- Done by many tools such as JUNIT and more we will see later.

Uses of Reflection #3: Reading Annotations

- We saw in the previous presentation that annotations could be read by a program, it's through reflection.
- Done by many tools such as JUNIT and more we will see later.

- By default annotations are **NOT** visible at runtime so we make them so as follows

@Retention(RetentionPolicy.RUNTIME) `

Uses of Reflection #3: Reading Annotations

- By default annotations are **NOT** visible at runtime so we make them so as follows

@Retention(RetentionPolicy.RUNTIME) `

Remember that @Retention() is a meta-annotation, an annotation that applies to annotations.

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
public @interface ClassDoc {
        String author();
        String created();
        String[] revisions();
}
```

Uses of Reflection #3: Reading Annotations

- If SomeClass is annotated with an annotation available at runtime ...

```
@ClassDoc(
        author="S Faroult",
        created="21/03/2017",
        revisions={"24/05/2017 –  Constructor with String parameter",
                   "26/02/2018 –  toString() rewritten"})
class SomeClass {
}
```

Note: it must be recompiled if ClassDoc is changed for getAnnotations to see it…

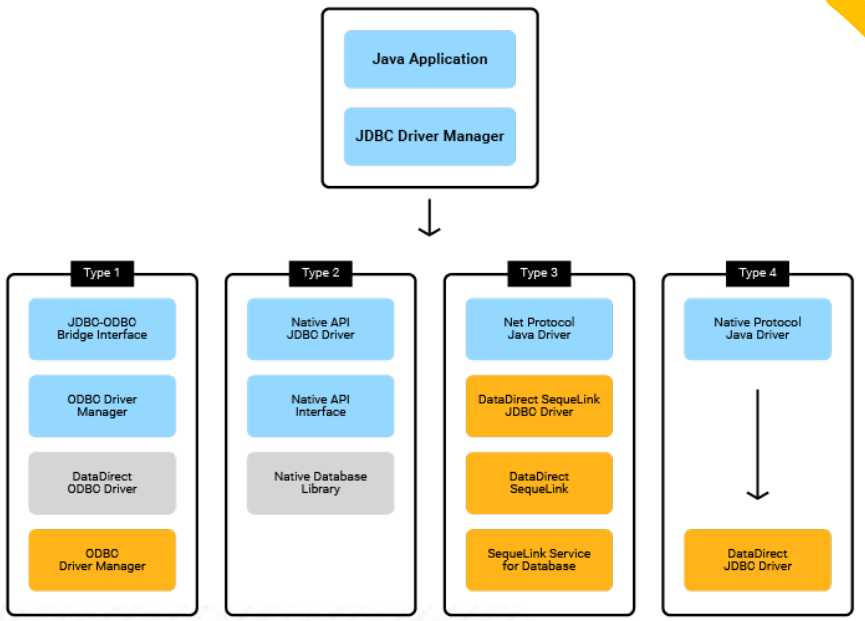
Uses of Reflection #3: Reading Annotations

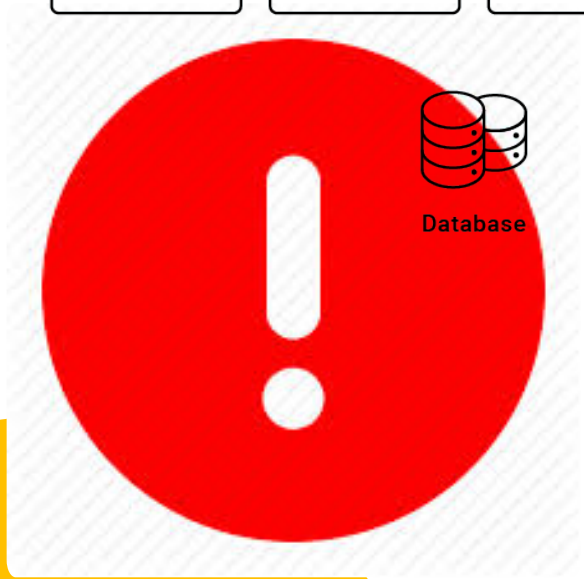
- Then annotations gets it…

```
import java.lang.annotation.Annotation;
public class ReadingAnnotations {

        public static void main(String[] args) {
                Annotation[] annotations = SomeClass .class
                                                .getAnnotations();
                for (Annotation annot: annotations) {
                        System.out.println(annot.toString());
                }
        }
}
```

```
$ java ReadingAnnotations
@ClassDoc(author=S Faroult, created=21/03/2017,
revisions=[24/05/2018 – Constructor with String
parameter, 26/02/2018 – toString() rewritten])
$
```



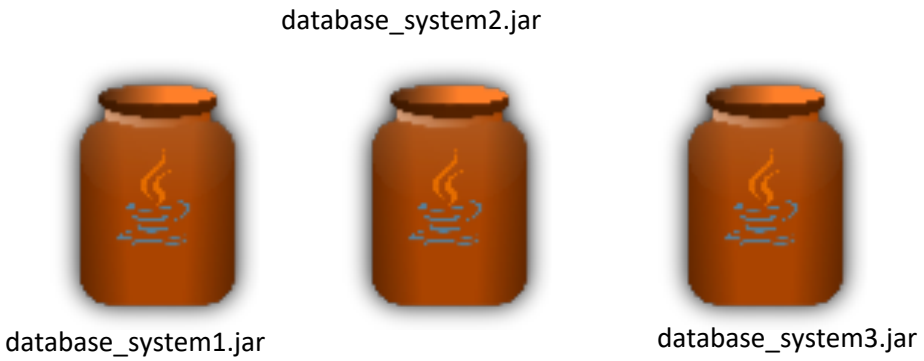


Uses of Reflection #4: Dynamically loading a class

- This is another very useful application of reflection
- Much used for "drivers" of hardware
- Because of the many different standards (National, International, Proprietary, etc) identical functionality is often achieved by different classes, that work with one special piece of hardware or software.

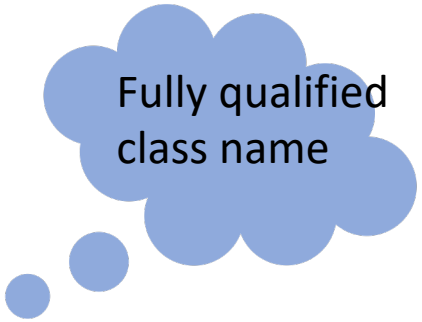
Uses of Reflection #4: Dynamically loading a class

- This is particularly useful with database access. Although there is a common language for accessing databases, database providers supply (as java archives) classes that implement the required methods to talk to THEIR system.
- More later when we discuss JDBC.

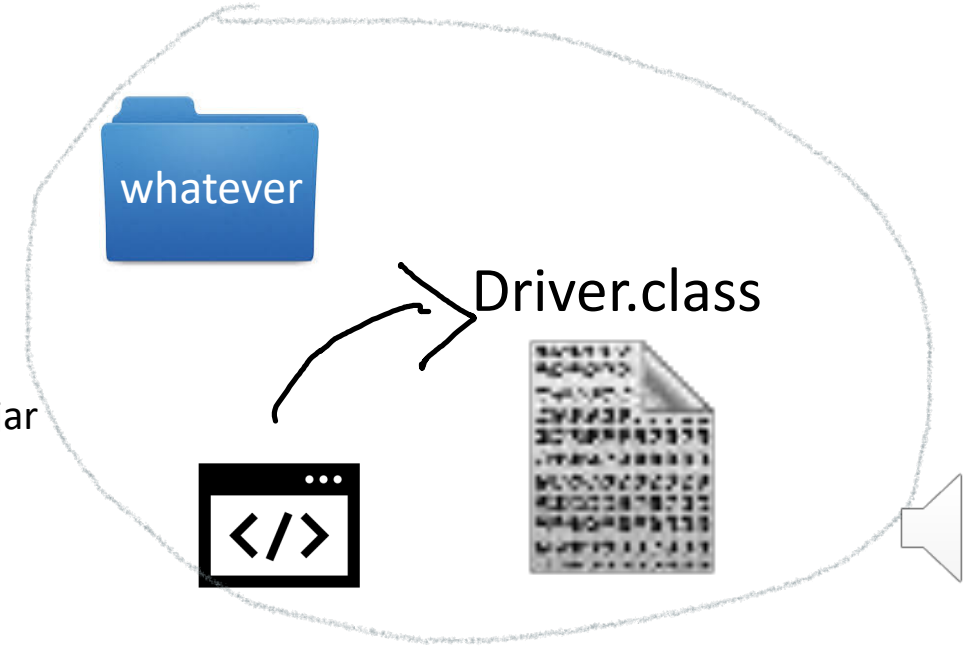


Uses of Reflection #4: Dynamically loading a class

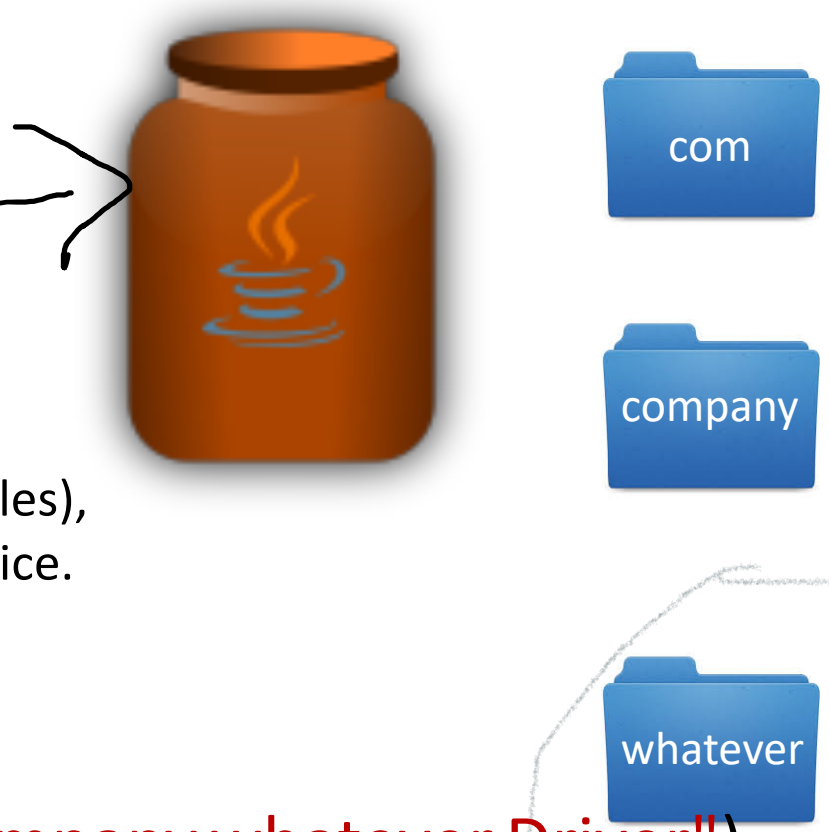
Usually the driver has a long complicated name to ensure that there is no conflict (two different drivers cannot have the same name).



Com.company.whateverdatabase_system.jar



Uses of Reflection #4: Dynamically loading a class



On the CLASSPATH

If the name of the .jar file is included in the CLASSPATH (where the loader looks for .class files), then the program can load the driver of its choice.

```
Class c = Class.forName("com.company.whatever.Driver");
Driver d = (Driver) c.newInstance();
```

Exercise: Download and run Squirrel SQL

There is a Java graphical tool called Squirrel SQL that uses this to let you query almost any database system, as long as you have the suitable .jar file added to your CLASSPATH. You can switch between very different systems.

Download link here: http://squirrel-sql.sourceforge.net

Follow to the tutorial here to view some databases:

http://squirrel-sql.sourceforge.net/kulvir/tutorial.html