CS209

Computer system design and application

Stéphane Faroult
faroult@sustc.edu.cn

Zhao Yao zhaoy6@sustc.edu.cn
Liu Zijian liuzijian47@163.com
Li Guansong intofor@163.com

Practical Example #2

The second example uses the Tokenizer of Lab1.
Goal: finding the 10 most used words in a speech. We need to associate with each word a counter (so, we need a Map<String,Integer>). If we don't know the word, we store it wih "1". Otherwise, we retrieve the counter, increase its value by one, and store it back. When we have counted words, we need to find the 10 most used – we need a map (associating a number of occurrences to a list of words, as there may be ties) but we also need some ordering. We need to go through our hash map and store its objects in, for instance, a TreeMap<Integer,TreeSet<String>>. Then we can iterate on the tree map and retrieve the most common words.

Practical Example #2

The result is usually very disappointing, because in an English speech the most common words are likely to be "the", "is", "a", and so forth. Those words are not very significant words and are usually called "stop words" (search engines on the Internet ignore them).
What we need to do is have a list of stop words, read it into an easily searchable structure such as a tree, and start counting words only when we cannot find them in this list of not important words. It gives a completely different vision of a speech.

A sample program (and a few speeches) has been uploaded to Sakai.

Java Goodies

What I'll present now are very interesting features from Java, which are mostly absent from the textbook, for mostly two reasons:
- or they appeared less than 10 years ago, when the book was written
- or they have taken an importance not suspected 10 years ago.

Annotations

The first feature is annotations. You may have noticed some annotations already; it's common when you use inheritance and you redefine in the child class a method defined in the parent class to precede the child class definition with
`@Override`
(which means *replace* the existing method).
This is an annotation, which is completely optional but warns javac of your intent. If you mistype the name, it will enable javac to detect an error if there is no such method in the parent class.

Annotations = Tags

Completely optional

Change nothing to what the program does

Help Javac – or the program

Much used by code-generating tools

As annotations can be accessed by programs, many tools that generate code – for tests, for instance – use annotations to collect information they cannot get otherwise.

Marker `@Override`

Single parameter

Multiple parameters

Annotations can take different forms, from the simple marker to some kinds of function calls that are outside the program itself.

METADATA

= DATA about the CODE

Metadata is a big concern in real life. Companies consider programs as assets, on which several generations of developers can work, which must be written in an easy-to-comprehend, standard way, and well documented. Metadata allows, among many other things, to industrialize code production and to standardize everything.

3 standard annotations

Java provides three standard annotations, which are all a way to give hints to javac.

@Override

@Deprecated = Obsolete

@SuppressWarnings (*warnings to suppress*)

For instance
`@SuppressWarnings({"deprecation", "unchecked"})`
`javac -X` gives the list of warnings, associated to `-Xlint`

2 annotations added in Java 7 and 8

@SafeVarargs

@FunctionalInterface

"Varargs" stands for "Variable [number of] Arguments"
We'll talk soon about what is a functional interface ...

You can create your own annotations!

Declared as interfaces

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

public @interface MyAnnotation {
}
```

Annotation-based tools use their own set of annotations, which you just need to import before using.

They can have methods but:

Methods should not have any parameters.

Methods declarations should not have any **throws** clauses.

As annotations are a bit special (it's a kind of program in the program) they are constrained by a number of rules.

They can have methods but:

Methods should not have any parameters.

Methods declarations should not have any **throws** clauses.

Return type must be one of: (or array of these types)

primitive type String enum Class

May provide structured documentation

```
class SomeClass {
      //  Created by S Faroult
      //  Creation date: 01/10/17
      //  Revision history:
      //    02/10/17 – Constructor
      //        with String parameter
      //    04/10/17 – toString() rewritten

}
```

What can you use annotations for in practice? Any Software Development Manager dreams of seeing comments like this. But every developer will not write them, and those who do may use a different format.

May provide structured documentation

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

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

methods

ClassDoc.java

Annotations may help turning readable but unparsable comments into data usable by a program.

May provide structured documentation

```
@ClassDoc(author="S Faroult",
          created="01/10/2017",
          revisions={"02/10/17 – Constructor
 with String parameter",
          "04/10/17 – toString() rewritten"})
class SomeClass {

}
```

Because the annotation is defined and checked by the javac, you can ensure a standard way of documenting code. This information can then be retrieved (we'll see how soon) to document programs.

Meta Annotations

5 other annotations about annotations

`@Retention`	Says whether the annotation is available to javac, or available at runtime.
`@Documented`	Make it appear in docs generated by the javadoc tool
`@Target`	What it applies to: Constructor, Method, Parameter ...
`@Inherited`	Passed to child classes (false by default)
`@Repeatable`	Can be applied more than once

JUNIT generates tests for checking your programs. Frameworks are software tools that try to generate automatically the boring bits of a program (which are often a lot of copy-and-paste).

Much used by tools

JUNIT

Frameworks

We'll see them later.

Reflection

I have said that annotations can be accessed by program, "reflection" is how to do it if you annotation was prefixed by `@Retention(RetentionPolicy.RUNTIME)`

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 and which will see in some detail in a few weeks.

Reflection

examine or modify the runtime behavior

Reflection

Works because of the JVM

Once again, it only works because of the JVM. 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.

stores in memory the description of classes when it loads them

Reflection

The JVM stores objects (of class Class) that describe every class used in the application.

Works because of the JVM

class called **Class**

objects represent classes in the running application

no constructor – built by the JVM

Reflection

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

```
ClassName obj = new ClassName();
```

```
obj.getClass()
```

method inherited from Object

1. The getClass() method of an object.

```
ClassName.class
```

"static" version

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

no constructor – built by the JVM

Reflection

```
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());
   }
}
```

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

For instance, you can retrieve class names.

Reflection

There are many useful uses for reflection. One common problem is locating files used by your program – the properties file to start with if there is one.

A few useful examples

1. Location of files read by your program
   - parameter file
   - data file
   - multimedia, and so forth

When people click on an icon to launch your program, the idea of "current directory" becomes extremey 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.

Reflection

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

Solution Get location at runtime

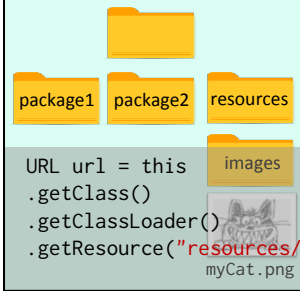
```
public class Reflection {

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

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

Reflection

And if you know the hierarchy of the files you need, you can easily derive the location of any file you supplied.

Solution Get location at runtime




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

Reflection

I have mentioned that annotations could be read by a program, it's through reflection

A few useful examples

2 Reading annotations

Done by many tools
(we'll see some of them later)

Reflection

There is a condition: the annotation must be available at runtime.

A few useful examples

2 Reading annotations

@Retention(RetentionPolicy.RUNTIME)

By default annotations are
NOT made available at 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();
  }
```

ClassDoc.java

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

```
@ClassDoc(author="S Faroult",
          created="01/10/2017",
          revisions={"02/10/17 – Constructor
 with String parameter",
          "04/10/17 – toString() rewritten"})
class SomeClass {



}
```

Must be recompiled if ClassDoc is changed

... then getAnnotations() 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=01/10/2017,
revisions=[02/10/17 – Constructor with String
parameter, 04/10/17 – toString() rewritten])
$
```

Reflection

There is another very important use of reflection.

A few useful examples

3. Dynamically loading a class

Much used for "drivers"

Because of the multiplication of standards, identical functionality is often achieved by different classes, that work with one special piece of hardware or software.

NEXT TIME

* More things that aren't in the book (lambda expressions, streaming)

* We'll start talking about graphical user interfaces, but in general terms. You can take a quick look at Chapter 14, however the book only talks about Swing, which now has a successor called JavaFx. I'll present JavaFx during lectures, so that you have some exposure both to Swing with the book (and there are probably still a lot of Swing applications in use) and to JavaFx, preferred for new applications. The general ideas are very much the same ones; class names and specific methods are different.