Java Reflection In Action

By

Ira. R. Forman & Nate Forman

Manning Publications CO.

Csc 668 Presentation by Jose Ortiz

Overview of Reflection

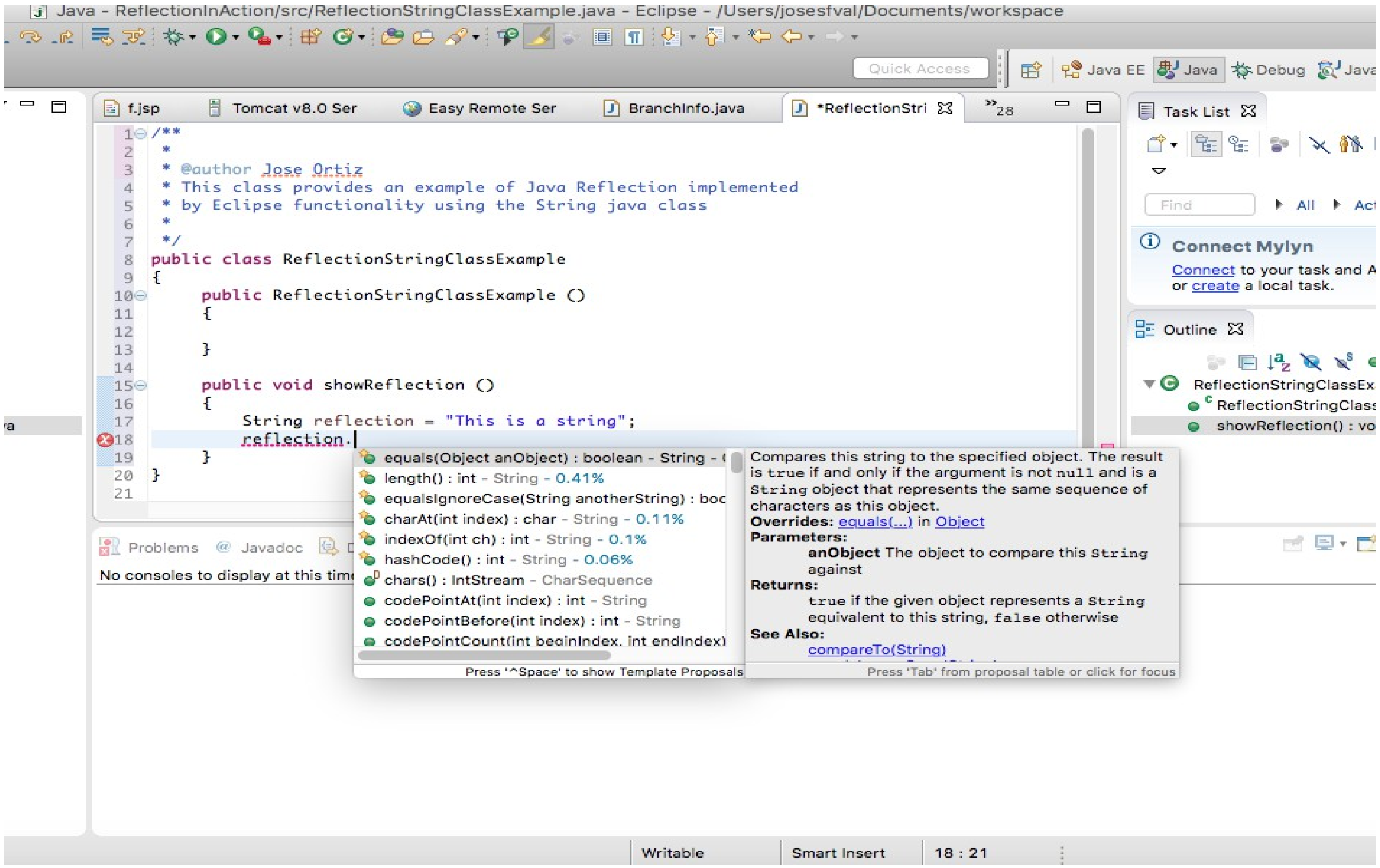
- **Introduction**
- Definition
- When to Use Reflection?
- Implementing Reflective Programs
- Summary and Conclusion

Introduction

- Reflection is a very important and powerful tool, for programmers, and can be implemented in many programing languages such as Java or C++
- Many modern IDE's implement Reflection as part of their functionality
- The next slide, provides an example of Java reflection implemented in Eclipse using the Java String class

Introduction

Java Reflection in Eclipse using the Java String Class



Java - ReflectionInAction/src/ReflectionStringClassExample.java - Eclipse - /Users/josesfval/Documents/workspace

```java
/**
 *
 * @author Jose Ortiz
 * This class provides an example of Java Reflection implemented
 * by Eclipse functionality using the String java class
 *
 */
public class ReflectionStringClassExample
{
     public ReflectionStringClassExample ()
     {

     }

     public void showReflection ()
     {
         String reflection = "This is a string";
         reflection.
     }
}
```

```
equals(Object anObject) : boolean - String -
length() : int - String - 0.41%
equalsIgnoreCase(String anotherString) : boo
charAt(int index) : char - String - 0.11%
indexOf(int ch) : int - String - 0.1%
hashCode() : int - String - 0.06%
chars() : IntStream - CharSequence
codePointAt(int index) : int - String
codePointBefore(int index) : int - String
codePointCount(int beginIndex, int endIndex)
Press '^Space' to show Template Proposals
```

Compares this string to the specified object. The result is `true` if and only if the argument is not `null` and is a `String` object that represents the same sequence of characters as this object.
Overrides: equals(...) in Object
Parameters:
anObject The object to compare this `String` against
Returns:
`true` if the given object represents a `String` equivalent to this string, `false` otherwise
See Also:
compareTo(String)

Press 'Tab' from proposal table or click for focus

No consoles to display at this time

Writable | Smart Insert | 18 : 21

Overview of Reflection

- Introduction
- **Definition**
- When to Use Reflection?
- Implementing Reflective Programs
- Summary and Conclusion

Definition

- **Reflection** is the ability of a running program to examine itself and its software environment, and to change what is does depending on what it finds.
- A program implementing reflection needs to use **metadata** in order to have a representation of itself. **Metadata** is organized into objects called **meta-objects**.
- The runtime self-examination of the **meta-objects** is called **introspection**.

Definition

- **Advantages** of implementing reflection
  - ➔ Flexibility
  - ➔ Adapts more easily to changing requirements
  - ➔ Reflective components are more likely to be reused
- **Issues** of implementing reflection
  - ➔ Security
  - ➔ Run-time performance

Overview of Reflection

- Introduction
- Definition
- **When to Use Reflection?**
- Implementing Reflective Programs
- Summary and Conclusion

When to Use Reflection?

- When to **implement reflexion** in our software ?
- Use Case :

➔ Your company has developed a client-side application that is about to be launched. Before launching the application, marketing tells you that using a different remote mechanism will increase the sales. Although switching is a good business decision, you must now implement all your remote interfaces

When to Use Reflection

- The last scenario illustrates a common issue in the software industry; a change of requirements in the last stage of the development phase of our product. Reimplementing interfaces and modifying methods can be a tedious mechanical task.

- By making our software **reflective**, it will adapt itself to any changes in its requirements by using the flexibility that provides the **reflective introspection mechanism**

Overview of Reflection

- Introduction
- Definition
- When to Use Reflection?
- **Implementing Reflective Programs**
- Summary and Conclusion

Implementing Reflective Programs

- In our last use case, **reflection** is very likely to be needed. The following high level algorithm would be a good fit to redesign our software product. So, it can adapt itself to future **changes on its requirements**.

1. Examine the program for its **internal structure** and data
2. Make **decisions** using the result of our examinations
3. Based on our examinations, **change the behavior structure or data** of the program.

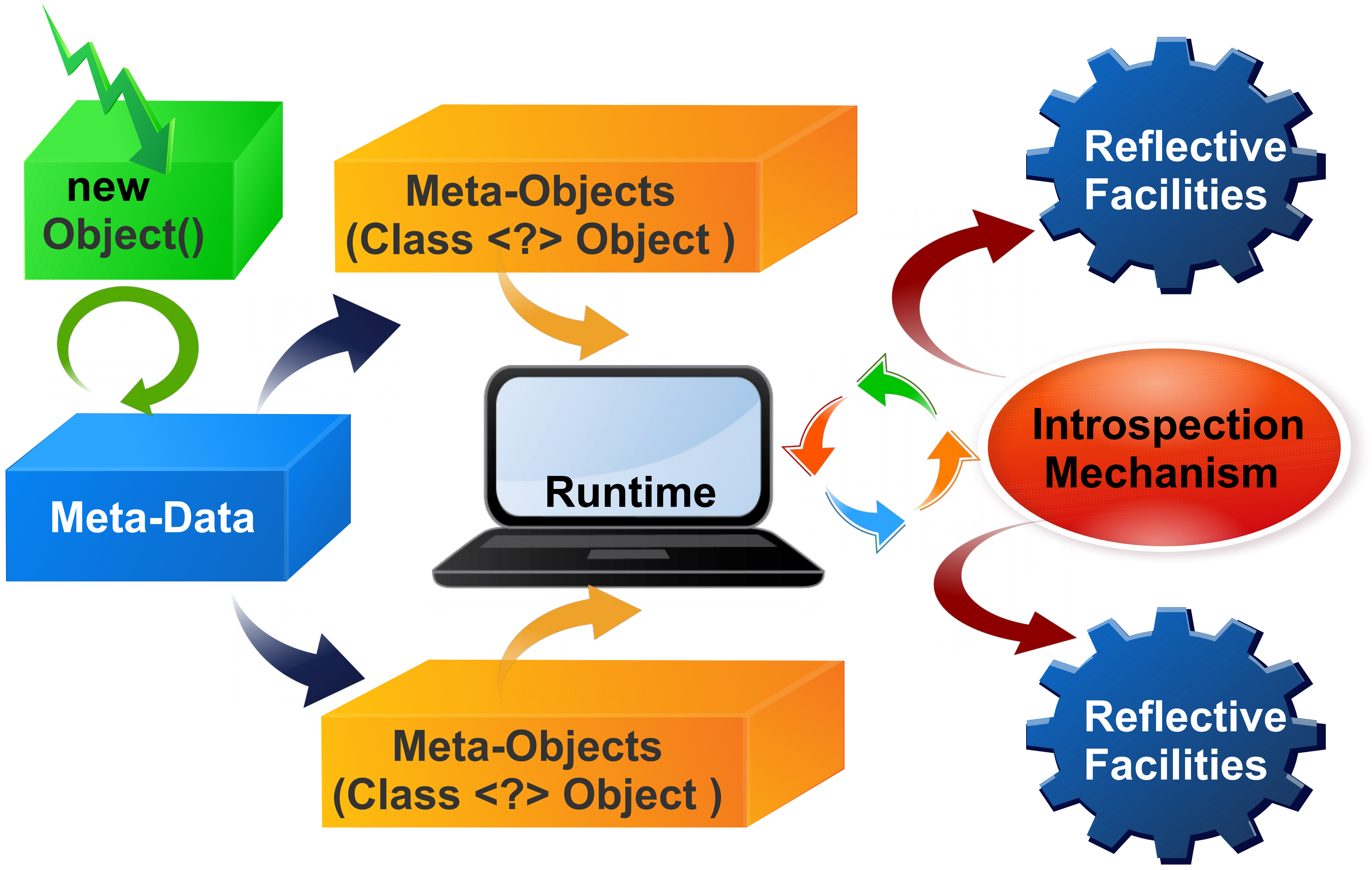
Implementing Reflective Programs

- The Class Object: Provides the **introspection** mechanism

```
Object obj = new Object ();
Class cls = obj.getClass();
```

- Instances of Class objects are **meta-objects**, and provide programing **metadata** about class's fields, methods, constructors, instance variables and modifiers. Also, they provide information about the inheritance hierarchy and provide access to **reflective facilities**.

Implementing Reflexive Programs



Implementing Reflective Programs

- In our next slides, we are going to build a reflective program **implementing the high level introspecting** algorithm presented in slide 11 in order to find information about the java.lang.String and java.util.Date Java classes
- The **introspection** of the meta-objects from those classes at runtime will return some of their **reflective facilities** from their methods such as their names, modifiers, types, and parameters' types.

Implementing Reflective Programs

```
import java.lang.reflect.*; // very important !!!
public Reflection {
   // Prints all the methods structure from the given meta-object
   public static void printMethods (Class <?> c )
   {
      String mod = "";
      String name = "";
      Class <?> type = null;  // meta-object type
      Method [] method = c.getDeclaredMethods(); // list of methods
      for (Method mt : method) // iterates over all the methods found
      {
            mt.setAccessible(true); // enables method accessibility
            int modifier = mt.getModifiers(); // modifiers e.g: private..
            mod = Modifier.toString(modifier);
            type = mt.getReturnType(); // int, boolean, String.....
            printParameterTypes(mt, mod, type);  // print param
       }
   }
```

Implementing Reflective Programs

```
// Prints parameters types for the given method
// print modifier type methodName ( paramType1, paramType2.... );
public static void printParameterTypes (Method mt, String mod,
                                        Class <?> mType)
{
    Class <?> [] types = mt.getParameterTypes(); // list of types
    int paramCount = 0;
    String separator = ", ";
    System.out.print( mod + " " + mType + " " m.getName() + " ( " );
    for (Class <?> pt : types)
    {
        if ( paramCount == types.length – 1 )  separator = " ";
        System.out.print(getDescriptor( pt.getName() ) + separator );
        paramCount++;

    }
    System.out.print (" ) \n" );
}
```

Implementing Reflective Programs

```
// Decrypts a primitive parameter type
// Returns decrypted primitive array Otherwise, returns prDescriptor
public static String getDescriptor ( String prDescriptor )
{
        String type = prDescriptor;
        // Checks if parameter type encrypted is array.
        if (prDescriptor.equals("[C")) type = "[] Char";
        else if (prDescriptor.equals("[B")) type = "[] byte";
        else if (prDescriptor.equals("[D")) type = "[] double";
        else if (prDescriptor.equals("[L")) type = "[] Object";
        else if (prDescriptor.equals("[I")) type = "[] int";
        else if (prDescriptor.equals("[java.lang.String") ||
                prDescriptor.equals("String;")) type = "[] String";
        else if (prDescriptor.equals("[Z")) type = "[] boolean";
        return type;
}
```

Implementing Reflective Programs

```
/** Test the program introspecting the Java String and Date
    Classes, including this class too. Note that the getClass()
    Method is final and cannot be override for security reasons */
 public void main (String args [] )
 {
     Class <?> strClass = new String().getClass();
     Class <?> dateClass = new Date().getClass();
     Class <?> thisClass = new Reflection().getClass();
     System.out.println(strClass.getName().toString());
     printMethods(strClass); // reflective methods of String class
     System.out.println(dateClass.getName().toString());
     printMethods(dateClass); // reflective methods of Date class
     System.out.println(dateClass.getName().toString());
     printMethods(dateClass); // reflective methods of this class
  }
} // end of Reflection class
```

Implementing Reflective Programs

- A sample output of our program reflecting on the String class would be:

```
Java.lang.String
public boolean equals (java.lang.Object)
public String toString ()
public int hashCode ()
public volatile int compareTo (java.lang.Object)
public int compareTo (java.lang.String)
public int indexOf (java.lang.String, int)
static int indexOf ([] Char, int, int, java.lang.String, int)
………
```

Implementing Reflective Programs

- A sample output of our program reflecting on the Java Date class would be:

```
Java.util.Date
private final BaseCalendar$Date normalize (sun.util.calendar.BaseCalendar$Date );
public static long parse (java.lang.String);
public boolean before ();
public boolean after ();
public void setTime (long);
public static long UTC (int, int, int ,int, int, int);
public static final synchronized BaseCalendar getJulianCalendar();
public Instant toInstant ()
………..
```

- As you can see there are methods that you probably has never seen before That's the power and the spirit of making a program reflective

Implementing Reflective Programs

- Finally, the reflection output of our own class

```
Reflection
public static void main ( java.lang.String)
public static class java.lang.String getDescriptor ( java.lang.String )
public static void printMethods ( java.lang.Class )
public static void printParameterTypes ( java.lang.reflect.Method,
                                         java.lang.String, java.lang.Class )
```

Overview of Reflection

- Introduction
- Definition
- When to Use Reflection?
- Implementing Reflective Programs
- **Summary and Conclusion**

Summary and Conclusion

- As you can see, **reflection** is a powerful tool that once well implemented provides the **flexibility** to **adapt** to any external changes from other modules or API's connected to our software product.

- In addition, **reflection** provides us the mechanism to avoid reimplementation of interfaces and methods after changes in our software requirements. Our software, automatically can **adap**t itself to those changes by inspecting the **metadata** and **meta-objects** from those API's

Summary and Conclusion

- Building a reflective program is not restricted only to Java. There are many modern programing languages that support reflection including scripting languages such as python and ruby.

- We just scratched the basics about reflection. Mastering reflection may take months, but I hope this intro has awaken your interest for this powerful programing concept..

Summary and Conclusion

- If you are interested, you can find in my gitHub page all the slides for this presentation as well as my complete program to reflect constructors, methods, constants, variables, interfaces, and inherited classes from any class made in Java including your own classes

  https://github.com/joseortizcostadev/JavaReflection