

Amrita Vishwa Vidyapeetham
TIFAC-CORE in Cyber Security

20CYS312 - Principles of Programming Languages
Assignment-01: Exploring Programming Paradigms

Ram Surya Suresh Kumar

21st January, 2024

Paradigm 1: Meta-Programming

Metaprogramming is writing programs that operate on other programs and as well operate on themselves.

Programs operating on other programs:

Writing:

- Compilers,
- Assemblers,
- Interpreters,
- Linkers,
- Loaders,
- Debuggers, and
- Profilers

Programs operating on themselves:

- Introspection - If a program simply looks at and reports on itself, we call this introspection .
- Reflection - If the program also modifies itself, we call this reflection.

In general, where code itself needs to change with the data, metaprogramming can be an effective approach.
Metaprograms treat code as data, and data as code.

History:

While metaprogramming has been around for decades, it only from the mid-1990s that it received increasing attention. More and more languages are supporting metaprogramming.

Language for Paradigm 1: Groovy

Groovy is a dynamic and powerful JVM language which has numerous features like closures and traits. In Groovy, it's possible to perform metaprogramming at both runtime and compile-time.

Run time Metaprogramming:

Runtime metaprogramming enables us to alter the existing properties and methods of a class. Also, we can attach new properties and methods; all at runtime.

- *propertyMissing* :

```
1 class Employee {
2     String firstName
3     String lastName
4     int age
5     def propertyMissing(String propertyName) {
6         "property '$propertyName' is not available"
7     }
8 }
9 Employee emp = new Employee(firstName: "Norman", lastName: "Lewis")
10 println emp.address
```

Figure 1: Code

```
property 'address' is not available
```

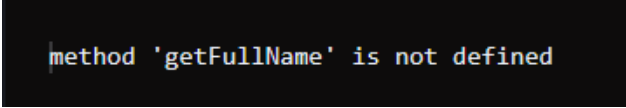
Figure 2: Output

- *methodMissing* :

The *methodMissing* method is similar to *propertyMissing*. However, *methodMissing* intercepts a call for any missing method, thereby avoiding the *MissingMethodException*.

```
1 class Employee {
2     String firstName
3     String lastName
4     int age
5     def methodMissing(String methodName, def methodArgs) {
6         println "method '$methodName' is not defined"
7     }
8 }
9 Employee emp = new Employee(firstName: "Norman", lastName: "Lewis")
10 emp.getFullName()
```

Figure 3: Code



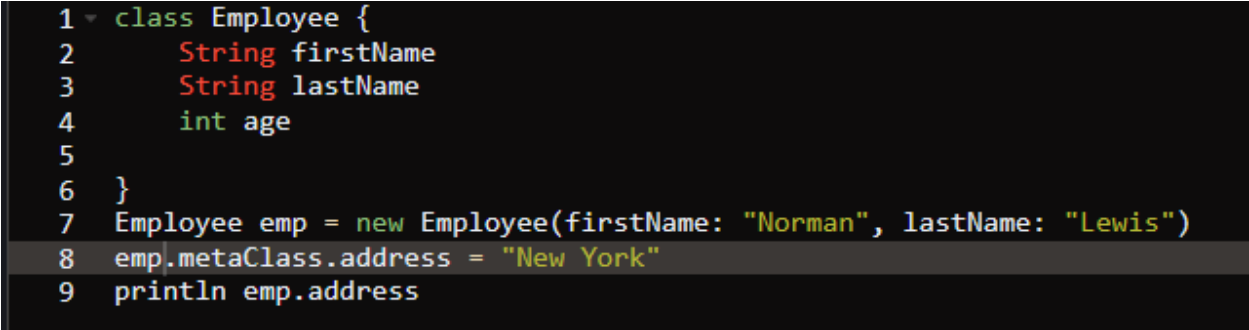
```
method 'getFullName' is not defined
```

Figure 4: Output

- ***ExpandoMetaClass*** : Groovy provides a *metaClass* property in all its classes. The *metaClass* property refers to an instance of the *ExpandoMetaClass*.

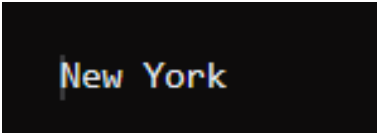
The <http://docs.groovy-lang.org/latest/html/api/groovy/lang/ExpandoMetaClass.html> *ExpandoMetaClass* class provides numerous ways to transform an existing class at runtime. For example, we can add properties, methods, or constructors.

Adding a property at run-time:



```
1 class Employee {
2     String firstName
3     String lastName
4     int age
5
6 }
7 Employee emp = new Employee(firstName: "Norman", lastName: "Lewis")
8 emp.metaClass.address = "New York"
9 println emp.address
```

Figure 5: code



```
New York
```

Figure 6: output

Adding a method at run-time:

```
class Employee {  
    String firstName  
    String lastName  
    int age  
}  
Employee emp = new Employee(firstName: "Norman", lastName: "Lewis")  
emp.metaClass.getFullName = {  
    println "$lastName, $firstName"  
}  
  
emp.getFullName()
```

Figure 7: code

```
Lewis, Norman
```

Figure 8: output

Adding a constructor to the *Employee* class at runtime:

```
class Employee {
    String firstName
    String lastName
    int age
    def "testMetaClassConstructor"() {
        when:
        Employee.metaClass.constructor = { String firstName ->
            new Employee(firstName: firstName)
        }

        and:
        Employee norman = new Employee("Norman")

        then:
        norman.firstName == "Norman"
        norman.lastName == null
    }
}
Employee emp = new Employee(firstName: "Norman", lastName: "Lewis")
println emp.testMetaClassConstructor()
```

Figure 9: code

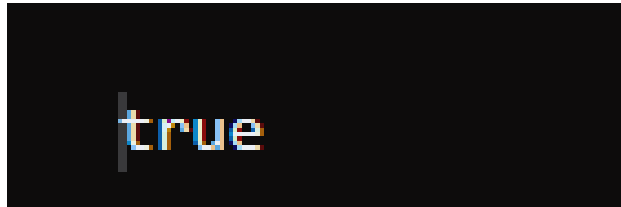


Figure 10: output

- **Extensions** : An extension can add a method to a class at runtime and make it accessible globally. The methods defined in an extension should always be static, with the *self* class object as the first argument.

To enable the *BasicExtensions*, we'll need to add the configuration file in the *META-INF/services* directory of our project.

So, let's add the *org.codehaus.groovy.runtime.ExtensionModule* file with the following configuration:

```
moduleName=core-groovy-2
```

```
moduleVersion=1.0-SNAPSHOT
```

```
extensionClasses=com.baeldung.metaprogramming.extension.BasicExtensions
```

```

1 class Employee {
2     String firstName
3     String lastName
4     int age
5
6 }
7
8 class BasicExtensions {
9     static int getYearOfBirth(Employee self) {
10         return 2003
11     }
12 }
13
14 Employee emp = new Employee(firstName: "Norman", lastName: "Lewis")
15 println emp.getYearOfBirth()

```

Figure 11: code

Similarly, to add *static* methods in a class, we'll need to define a separate extension class. Then, we enable

```

class Employee {
    String firstName
    String lastName
    int age
}

class StaticEmployeeExtension {
    static Employee getDefaultObj(Employee self) {
        return new Employee(firstName: "firstName", lastName: "lastName", age: 20)
    }
}

assert Employee.getDefaultObj().firstName == "firstName"
assert Employee.getDefaultObj().lastName == "lastName"
assert Employee.getDefaultObj().age == 20

```

Figure 12: code

the *StaticEmployeeExtension* by adding the following configuration to the *ExtensionModule* file:

```
staticExtensionClasses=com.baeldung.metaprogramming.extension.StaticEmployeeExtension
```

Compile time Metaprogramming:

Using specific annotations, we can effortlessly alter the class structure at compile-time. In other words, we can use annotations to modify the abstract syntax tree of the class at the compilation.

- **@ToString** : The *@ToString* annotation adds a default implementation of the *toString* method to a class at compile-time. All we need is to add the annotation to the class.

```
@ToString
class Employee {
    long id
    String firstName
    String lastName
    int age
}

Employee employee = new Employee()
employee.id = 1
employee.firstName = "norman"
employee.lastName = "lewis"
employee.age = 28

assert employee.toString() == "com.baeldung.metaprogramming.Employee(1, norman, lewis, 28)"
```

Figure 13: code

- **@TupleConstructor** : Use *@TupleConstructor* in Groovy to add a parameterized constructor in the class. This annotation creates a constructor with a parameter for each property.

```
@TupleConstructor
class Employee {
    long id
    String firstName
    String lastName
    int age
}

Employee snake = new Employee(2, "snake")
assert snake.toString() == "Employee(snape, null, 0)"
```

Figure 14: code

-
- **@EqualsAndHashCode** : We can use *@EqualsAndHashCode* to generate the default implementation of *equals* and *hashCode* methods at compile time.

```
@EqualsAndHashCode
class Employee {
    long id
    String firstName
    String lastName
    int age

    def "testEqualsAndHashCodeAnnotation"() {
        when:
        Employee norman = new Employee(1, "norman", "lewis", 28)
        Employee normanCopy = new Employee(1, "norman", "lewis", 28)

        then:
        norman == normanCopy
        norman.hashCode() == normanCopy.hashCode()
    }
}
```

Figure 15: code

- **@Canonical** : It is a combination of *@ToString*, *@TupleConstructor*, and *@EqualsAndHashCode* annotations.

Just by adding it, we can easily include all three to a Groovy class. Also, we can declare *@Canonical* with any of the specific parameters of all three annotations.

- **@AutoClone** : A quick and reliable way to implement *Cloneable* interface is by adding the *@AutoClone* annotation.

```
@AutoClone
class Employee {
    long id
    String firstName
    String lastName
    int age
}

try {
    Employee norman = new Employee(1, "norman", "lewis", 28)
    def normanCopy = norman.clone()
    assert norman == normanCopy
} catch (CloneNotSupportedException e) {
    e.printStackTrace()
}
```

Figure 16: code

- **Logging support with @Log**: Let's enable the logging provided by JDK by adding the *@Log* annotation to the *Employee* class.

```
@Log
class Employee {
    long id
    String firstName
    String lastName
    int age

    def logEmp() {
        log.info "Employee: $lastName, $firstName is of $age years age"
    }
}

Employee employee = new Employee(1, "Norman", "Lewis", 28)
employee.logEmp()
```

Figure 17: code

Paradigm 2: Scripting

Scripting is a programming paradigm that is characterized by the use of scripts, which are sequences of commands written in a scripting language. The scripting paradigm is often associated with interpreted languages and environments. Some common characteristics and principles associated with the scripting paradigm:

- **Interpreted Execution:** Scripts are typically interpreted rather than compiled. This means that the source code is executed directly by an interpreter or runtime environment without the need for a separate compilation step. This allows for a more flexible and dynamic development process.
- **Dynamic Typing:** Scripting languages often employ dynamic typing, where variable types are determined at runtime rather than being explicitly declared. This can simplify code and make it more flexible but may also introduce challenges related to type errors.
- **Rapid Development:** The scripting paradigm is often associated with rapid development cycles. Since there is no need for compilation, developers can quickly write, test, and modify code. This makes scripting languages well-suited for tasks that require frequent experimentation and iteration.
- **Scripting Languages:** Scripting languages cover a wide range of domains, including web development (e.g., JavaScript, Python, Ruby), system administration (e.g., Bash, PowerShell), automation (e.g., Python, Perl), and more.

Example: Structure of a script in web development

```
Fetch input parameters from the environment.  
Authenticate the user against a username/password  
database.  
Issue a database query.  
Format the results as a web page.  
Print the web page to the standard output stream.
```

Language for Paradigm 2: Shell

A shell script is a text file that contains a sequence of commands for a UNIX-based operating system. It is called a shell script because it combines a sequence of commands, that would otherwise have to be typed into the keyboard one at a time, into a single script.

The shell is the operating system's command-line interface (CLI) and interpreter for the set of commands that are used to communicate with the system.

A shell script is usually created for command sequences in which a user has a need to use repeatedly in order to save time. Like other programs, the shell script can contain parameters, comments and subcommands that the shell must follow.

0.1 How shell scripting works

The basic steps involved with shell scripting are writing the script, making the script accessible to the shell and giving the shell execute permission.

Shell scripts contain ASCII text and are written using a text editor, word processor or graphical user interface (GUI). The content of the script is a series of commands in a language that can be interpreted by the shell. Functions that shell scripts support include loops, variables, if/then/else statements, arrays and shortcuts. Once complete, the file is saved typically with a .txt or .sh extension and in a location that the shell can access.

0.2 Types of shells

In Unix and Linux, the two major types of shell scripts are:

1. Bourne again shells (BASH)- BASH is the default shell for Unix version 7. The character for prompting a bourne again shell is \$.
2. C shells- A C shell is run in a text terminal window and is able to easily read file commands. The character for prompting a C shell is %

0.3 Examples of shell script applications

Using a shell script is most useful for repetitive tasks that may be time consuming to execute by typing one line at a time. A few examples of applications shell scripts can be used for include:

- Automating the code compiling process.
- Running a program or creating a program environment.
- Completing batch
- Manipulating files.
- Linking existing programs together.
- Executing routine backups.
- Monitoring a system.

Examples:

Automating the code compiling process:

```
#!/bin/bash

SOURCE_FILES="*.cpp"
OUTPUT_EXECUTABLE="my_program"
COMPILER="g++"

echo "Compiling source code..."
$COMPILER $SOURCE_FILES -o $OUTPUT_EXECUTABLE

if [ $? -eq 0 ]; then
    echo "Compilation successful. Running the program..."
    ./$OUTPUT_EXECUTABLE
else
    echo "Compilation failed."
fi
```

Running a program or creating a program environment:

```
#!/bin/bash
INPUT_FILE="input.txt"
TEMP_FILE="temp_output.txt"
PROGRAM1="program1"
PROGRAM2="program2"

echo "Running $PROGRAM1 with input from $INPUT_FILE..."
$PROGRAM1 $INPUT_FILE > $TEMP_FILE

if [ $? -eq 0 ]; then
    echo "$PROGRAM1 executed successfully. Running $PROGRAM2..."
    $PROGRAM2 $TEMP_FILE
else
    echo "Error: $PROGRAM1 failed to execute."
fi
rm $TEMP_FILE
```

Analysis

Provide an analysis of the strengths, weaknesses, and notable features of both paradigms and their associated languages.

The advantages of Metaprogramming:

- Template metaprogramming has no side effect since it is immutable, so we cannot modify an existing type
- There is better code readability compared to code that does not implement metaprogramming
- It reduces repetition of the code

The disadvantages of Metaprogramming:

- The syntax is quite complex.
- The compilation time takes longer since we now execute code during compile-time.
- The compiler can optimize the generated code much better and perform inlining, for instance, the C `qsort()` function and the C++ sort template.

The advantages of scripting:

1. Automation:

Shell scripting allows you to automate repetitive tasks, saving time and reducing the potential for human error.

2. Customization:

You can customize your scripts to fit your specific needs, making them highly flexible and adaptable.

3. Interoperability:

Shell scripts can interact with various system components and programs, making them useful for integrating different software tools.

4. Portability:

Shell scripts are typically portable across different Unix-like operating systems, allowing for consistent functionality across platforms.

5. Rapid Prototyping:

Shell scripting provides a quick and efficient way to prototype new processes or test ideas before implementing them in more complex programming languages.

6. System Administration:

Shell scripts are commonly used for system administration tasks such as system maintenance, file manipulation, and user management.

The disadvantages of scripting:

They say that every good thing has its negatives, which also hold true for shell programming, here are some of its disadvantages:

- Slow execution speed
- Not suited for large and complex tasks
- Design flaws within the language syntax
- Provides minimal data structure, unlike other scripting languages
- Prone to costly errors

Comparison

Metaprogramming and scripting serve different purposes within the realm of programming. Metaprogramming is more concerned with manipulating code structures, often at a higher level of abstraction, while scripting is focused on automating tasks and interacting with the system through sequences of commands.

Differences:

Purpose:

- **Metaprogramming:** Primarily concerned with manipulating or generating code at a higher level, often for generic algorithms, frameworks, or code generation tools.
- **Scripting:** Primarily concerned with automating tasks, interacting with the operating system, and quickly developing and executing sequences of commands.

Level of Abstraction:

- **Metaprogramming:** Operates at a high level of abstraction, involving the manipulation of code structures, classes, or even the language itself. Involves generating or modifying code during compile-time or runtime.
- **Scripting:** Operates at a more user-friendly level, often involving the automation of tasks through sequences of commands. Scripting languages are designed for quick and easy interaction with the system.

Execution Model:

- **Metaprogramming:** Often involves compile-time metaprogramming, where code modifications occur during the compilation phase. Runtime metaprogramming modifies code structures during program execution.
- **Scripting:** Generally involves interpreted execution, where scripts are executed line by line at runtime. There is typically no separate compilation step.

Use Cases:

- **Metaprogramming:** Commonly used in the development of libraries, frameworks, and code generation tools. It allows for the creation of flexible and generic solutions.
- **Scripting:** Commonly used for automation, system administration, quick prototyping, and tasks involving sequential execution of commands.

Language Features:

- **Metaprogramming:** Often involves language features like macros, template metaprogramming, or reflective capabilities that allow the manipulation of code structures.
- **Scripting:** Typically characterized by dynamic typing, high-level abstractions, and features that simplify common tasks, such as file I/O, string manipulation, and process execution.

Runtime vs. Compile-time:

- **Metaprogramming:** Can occur both at runtime and compile-time. Runtime metaprogramming allows for dynamic modifications during program execution, while compile-time metaprogramming occurs during the compilation phase.
- **Scripting:** Primarily occurs at runtime, as scripting languages are often interpreted. However, scripts may interact with compiled code or other languages.

Similarities:**1. Dynamic Nature:**

Both metaprogramming and scripting often involve dynamic features. Metaprogramming allows the modification of a program's structure at runtime, and scripting languages are often dynamically typed and interpreted.

2. Flexibility and Expressiveness:

Both paradigms aim to provide flexibility and expressiveness to developers. Metaprogramming allows for code generation and modification, while scripting languages often have high-level abstractions and concise syntax for quick development.

3. Reflection:

Both paradigms frequently make use of reflection. Metaprogramming often relies on reflective features to inspect and modify the program structure at runtime. Scripting languages may use reflection for dynamic runtime behavior.

Challenges Faced

Even though I am proficient in shell scripting, when it came to understanding it as a programming paradigm took me some time. Also finding the resources for scripting as a paradigm was difficult.

The compile-time metaprogramming with the annotations when run in an online groovy compiler, didn't work for me.

Transitioning from one paradigm to another (Metaprogramming to scripting) was a bit challenging.

Conclusion

Both metaprogramming and scripting paradigms offer unique capabilities and cater to different programming needs. Metaprogramming provides a powerful toolset for creating abstract and dynamic systems, while scripting excels in automating tasks and simplifying interaction with the operating system. Choosing between them depends on the specific requirements of a project, with metaprogramming suitable for building robust frameworks and scripting ideal for quick and efficient automation. Understanding the strengths and trade-offs of each paradigm empowers developers to make informed decisions based on the demands of their projects.

References

<https://www.baeldung.com/groovy-metaprogramming>
<https://www.techtarget.com/searchdatacenter/definition/shell-script>
<https://www.jdoodle.com/execute-groovy-online/>