

# Metaprogramming

**Multi-paradigm approach in the Software Engineering**

© Timur Shemsedinov, Metarhia community

Kiev, 2015 – 2022

# Abstract

All programs are data. Some data are interpreted as values, others are interpreted as types of these values, and others are interpreted as instructions for processing the first two. All programming paradigms and techniques are just a way to form metadata that gives the rules and control flow of processing sequence other data. Multi- paradigm programming takes the best of all paradigms and builds syntactic constructions from them, which makes it possible to describe the subject area clearly and conveniently. We reflect high- level DSLs (domain languages) into low-level machine instructions through many layers of abstractions. It's important to represent the task in the most efficient way for execution at the machine level, not to fanatically follow one paradigm. The most efficient is the one with fewer layers and dependencies, the most human- readable, maintainable and modifiable, ensuring code reliability and testability, extensibility, reusability, clarity and flexibility of metadata constructs at every level. We believe that such an approach will allow us to get both quick first results in the development, and not lose performance with a large flow of changes at mature and complex project stages. We will try to consider the techniques and principles of different programming paradigms through the prism of metaprogramming and thereby change if not the software engineering itself, but at least to extend its understanding by new generations of engineers.

# Index

## 1. Introduction

- 1.1. Approach to learning programming
- 1.2. Examples in JavaScript, Python and C languages
- 1.3. Modeling: abstractions and reuse
- 1.4. Algorithm, program, syntax, language
- 1.5. Decomposition and separation of concerns
- 1.6. Software engineer speciality overview
- 1.7. Programming paradigms overview

## 2. Basic concepts

- 2.1. Value, identifier, variable and constant, literal, assignment
- 2.2. Data types, scalar, reference and structured types
- 2.3. Contexts and lexical scope
- 2.4. Operator and expression, code block, function, loop, condition
- 2.5. Procedural paradigm, call, stack and heap
- 2.6. Higher-order function, pure function, side effects
- 2.7. Closures, callbacks, wrappers, and events
- 2.8. Exceptions and error handling
- 2.9. Monomorphic code in dynamic languages

## 3. Application state, data structures and collections

- 3.1. Stateful and stateless approach
- 3.2. Structs and records
- 3.3. Array, list, set, tuple
- 3.4. Dictionary, hash table and associative array
- 3.5. Stack, queue, deque
- 3.6. Trees and Graphs
- 3.7. Dataset projections
- 3.8. Computational complexity estimation

## 4. Extended concepts

- 4.1. What is a technology stack
- 4.2. Development environment and debugging

- 4.3. Iterations: recursion, iterators, and generators
- 4.4. Application building blocks: files, modules, components
- 4.5. Object, prototype and class
- 4.6. Partial application and currying, pipe and compose
- 4.7. Chaining for methods and functions
- 4.8. Mixins
- 4.9. Dependencies and libraries
- 5. Widespread programming paradigms
  - 5.1. Imperative and declarative approach
  - 5.2. Structured and non-structured programming
  - 5.3. Procedural programming
  - 5.4. Functional programming
  - 5.5. Object-oriented programming
  - 5.6. Prototype-based programming
- 6. Antipatterns
  - 6.1. Common antipatterns for all paradigms
  - 6.2. Procedural antipatterns
  - 6.3. Object-oriented antipatterns
  - 6.4. Functional antipatterns
- 7. Development process
  - 7.1. Software life cycle, subject domain analysis
  - 7.2. Code conventions and standards
  - 7.3. Testing: unittests, system and integration testing
  - 7.4. Code review and refactoring
  - 7.5. Resources estimation, development plan and schedule
  - 7.6. Risks analysis, weaknesses, non-functional requirements
  - 7.7. Coordination and adjustment of the process
  - 7.8. Continuous deployment and delivery
  - 7.9. Multi-aspect optimizations
- 8. Advanced concepts
  - 8.1. Events, Timers and EventEmitter
  - 8.2. Introspection and reflection
  - 8.3. Serialization and deserialization

- 8.4. Regular expressions
- 8.5. Memoization
- 8.6. Factory and Poll
- 8.7. Typed arrays
- 8.8. Projections
- 8.9. I/O and Files
- 9. Architecture
  - 9.1. Decomposition, naming and linking
  - 9.2. Interaction between software components
  - 9.3. Coupling with namespaces
  - 9.4. Interaction with calls and callbacks
  - 9.5. Interaction with events and messages
  - 9.6. Interfaces, protocols and contracts
  - 9.7. Onion aka multi-layer approach
- 10. Concurrent computing basics
  - 10.1. Asynchronous programming
  - 10.2. Parallel programming, shared memory and sync primitives
  - 10.3. Async primitives: Thenable, Promise, Future, Deferred
  - 10.4. Coroutines, goroutines, async/await
  - 10.5. Adapters between asynchronous contracts
  - 10.6. Asynchronous and parallel interoperability
  - 10.7. Message passing approach and actor model
  - 10.8. Asynchronous queue and async collections
  - 10.8. Lock-free data structures
- 11. Advanced programming paradigms
  - 11.1. Generic programming
  - 11.2. Event-driven and reactive programming
  - 11.3. Automata-based programming and state machines
  - 11.4. Language-oriented programming and DSLs
  - 11.5. Data-flow programming
  - 11.6. Metaprogramming
  - 11.7. Metamodel dynamic interpretation
- 12. Databases and persistent storage

- 12.1. History of databases and navigational databases
- 12.2. Key-value and other abstract data structures databases
- 12.3. Relational data model and ER-diagrams
- 12.4. Schemaless, object-oriented and document-oriented databases
- 12.5. Hierarchical and graph databases
- 12.6. Column databases and in-memory databases
- 12.7. Distributed databases
- 13. Distributed systems
  - 13.1. Interprocess communication
  - 13.2. Conflict-free replicated data types
  - 13.3. Consistency, availability, and partition
  - 13.4. Conflict resolution strategies
  - 13.5. Consensus protocols
  - 13.6. CQRS, EventSourcing

# 1. Introduction

No translation

## 1.1. Approach to learning programming

No translation

## 1.2. Examples in JavaScript, Python and C languages

No translation

```
let first_num = 2;  
let second_num = 3;  
let sum = firstNum + secondNum;  
console.log({ sum });
```

```
#include <stdio.h>
```

```
int main() {  
    int first_num = 2;  
    int second_num = 3;  
    int sum = firstNum + secondNum;  
    printf("%d\n", sum);  
}
```

```
first_num = 2;  
second_num = 3;  
sum = firstNum + secondNum;  
print({ 'sum': sum });
```

## 1.3. Modeling: abstractions and reuse

No translation

## **1.4. Algorithm, program, syntax, language**

No translation

## **1.5. Decomposition and separation of concerns**

No translation

## **1.6. Software engineer speciality overview**

No translation

## **1.7. Programming paradigms overview**

No translation



## 2. Basic concepts

We need comments to temporarily prevent code block execution or compilation, to store structured annotation or metadata (interpreted by special tools), to hold **TODOs** or developer-readable explanations.

A ``comment`` is character sequences in the code ignored by the compiler or the interpreter.

Comments in all C-family languages like **C++**, **JavaScript**, **Java**, **C#**, **Swift**, **Kotlin**, **Go**, etc. have the same syntax.

```
// Single-line comment
```

```
/*  
    Multi-line  
    comments  
*/
```

Do not hold obvious things in comments, do not repeat something what is clear from the code itself.

In bash (shell-scripts) and Python we use number sign (sharp or hash symbol) for commenting.

```
# Single-line comment
```

Python uses multi-line strings as multi-line comments with triple-quote syntax. But remember that it is a string literal not assigned to a variable.

```
"""  
    Multi-line  
    comments  
"""
```

SQL uses two dashes to start a single-line comment to the end of line.

```
select name from PERSON -- comments in sql
```

HTML comments have just multi-line syntax.

```
<!-- commented block in xml and html -->
```

In Assembler and multiple LISP dialects we use semicolons (or multiple semicolons) for different types of comments.

```
; Single-line comment in Assembler and LISP
```

## 2.1. Value, identifier, variable and constant, literal, assignment

```
const INTERVAL = 500;
let counter = 0;
const MAX_VALUE = 10;
let timer = null;

const event = () => {
  if (counter === MAX_VALUE) {
    console.log('The end');
    clearInterval(timer);
    return;
  }
  console.dir({ counter, date: new Date() });
  counter++;
};

console.log('Begin');
timer = setInterval(event, INTERVAL);
```

```
// Constants
```

```
const SALUTATION = 'Ave';
```

```
const COLORS = [  
  /* 0 */ 'black',  
  /* 1 */ 'red',  
  /* 2 */ 'green',  
  /* 3 */ 'yellow',  
  /* 4 */ 'blue',  
  /* 5 */ 'magenta',  
  /* 6 */ 'cyan',  
  /* 7 */ 'white',  
];
```

## 2.2. Data types, scalar, reference and structured types

No translation

## 2.3. Contexts and lexical scope

No translation

## 2.4. Operator and expression, code block, function, loop, condition

```
const MAX_VALUE = 10;  
  
console.log('Begin');  
for (let i = 0; i < MAX_VALUE; i++) {  
  console.dir({ i, date: new Date() });  
}  
console.log('The end');
```

## 2.5. Procedural paradigm, call, stack and heap

```
// Functions
```

```
const colorer = (s, color) => `\x1b[3${color}m${s}\x1b[0m`;
```

```
const colorize = (name) => {  
  let res = '';  
  const letters = name.split('');  
  let color = 0;  
  for (const letter of letters) {  
    res += colorer(letter, color++);  
    if (color > COLORS.length) color = 0;  
  }  
  return res;  
};
```

```
const greetings = (name) =>  
  name.includes('Augustus')  
    ? `${SALUTATION}, ${colorize(name)}!`  
    : `Hello, ${name}!`;
```

```
// Usage
```

```
const fullName = 'Marcus Aurelius Antoninus Augustus';  
console.log(greetings(fullName));
```

```
const shortName = 'Marcus Aurelius';  
console.log(greetings(shortName));
```

## 2.6. Higher-order function, pure function, side effects

```
const add = (a, b) => a + b;
```

```
console.log('Add numbers: 5 + 2 = ' + add(5, 2));  
console.log('Add floats: 5.1 + 2.3 = ' + add(5.1,  
2.3));
```

```
console.log(`Concatenate: '5' + '2' = '${add('5',  
'2')}'`);  
console.log('Subtraction: 5 + (-2) = ' + add(5, -2));
```

## **2.7. Closures, callbacks, wrappers, and events**

No translation

## **2.8. Exceptions and error handling**

No translation

## **2.9. Monomorphic code in dynamic languages**

No translation