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

No translation

1.1. Approach to learning programming

Many people think that the essential skill of a programmer is to write code. In fact, programmers are more likely to read code and fix it. And the main criteria for the quality and code are understandability, readability, and simplicity. As Harold Abelson said: “Programs must be written for the people who will read them, and the machines that will execute these programs are secondary.”

The essential skills of a programmer are reading and fixing code.

Each topic contains examples of good code and bad code. These examples are collected from programming practice and project reviews. Tailor-made examples of bad code will work, but they are full of anti-patterns and problems that need to be identified and fixed. Even the very first practical work in the course will be related to correcting the code and increasing its readability. If you give traditional tasks (write a function by signature, algorithm, class), then the beginner does not implement it in the best way but will protect his code because this is the first thing he wrote. And if the task is to “take an example of someone else’s bad code, find problems and fix” not rewrite from scratch but improve in several steps, fixing and realizing these steps, then a critical approach is turned on.

Fixing bad code is one of the most effective ways to learn.

The beginner receives code review examples and, by analogy, aims to correct his task. Such iterations are repeated many times without losing the critical approach. Perfect if there is a mentor who observes the improvements and can correct and suggest. By no means should the mentor do the work for the beginner, but rather direct him on how to think about programming and where to search for a solution.

A mentor is an indispensable assistant in the development of professional growth.

The next step is to write code on your own. We highly recommend that the beginners share these solutions for cross-review. Of course, before that, you need to use linters and code formatters that will analyze the syntax, find errors in it and identify problem areas for a large number of code templates. It is highly important to ensure that a colleague understands your thought and does not waste time on syntax and formatting.

Use friendly code review, cross reviews, linters, and formatters.

We move on to exercises on decoupling between several abstractions, then between modules, i.e. it should be done in such a way that you need to know as little as possible about the data structures of one part of the program from another part of it. The reduction of language fanaticism is achieved by learning in parallel several programming languages from the beginning and translations from one language to another. It is very easy to translate from **JavaScript** to **Python**. With **C** it is a bit more harder, but these three languages, whatever they are, cannot be left out of the course.

From the first steps, do not allow any fanaticism: language, framework, paradigm.

Decrease in framework fanaticism — prohibition for beginners to use libraries and frameworks and focus on the most native code without dependencies. While decreasing paradigm fanaticism, try to combine procedural, functional, OOP, reactive and automatic programming. We will try to show how these combinations allow us to simplify patterns and principles from GoF and SOLID.

The next important part of the course is the study of antipatterns and refactoring. Firstly, we will give an overview, and then we will practice using real code examples from live projects.

1.2. Examples in JavaScript, Python and C languages

We will write code examples in different languages, but preference will not be given to the best, beautiful and fast, but to those that are indispensable. We will take **JavaScript** as the most common, **Python**, because there are areas where you cannot do without it and **C**, language close enough to assembly language, which is still very relevant and has had the significant influence on modern languages in terms of syntax and built-it ideas. All three are very far from the language of my dreams, but this is what we have. At first glance, **Python** is very different from **JavaScript** and other C-like languages, although this is only at first glance, we will show that it is very similar to **JavaScript** since the type system, data structures and especially the built-in collections are very similar in them. Although syntactically, the difference in the code blocks organization using indentation and curly brackets **{}** is striking the eye, in reality, such a difference is not so significant, and there is much more in common between **JavaScript** and **Python** than between any of them and the language **C**.

We will not start over by learning the syntax, but immediately by reading bad code and searching for errors in it. Let's take a look at the following snippets, the first one will be in **JavaScript**:

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

Try to understand what is written here, and where there may be errors. And then compare this code with its translation to **C**.

```
#include <stdio.h>

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

The errors here are the same, they can be easily identified by a person, who does not even know the basics of programming, if he examines the code. And the next piece of code will be in **Python**, it does exactly the same and contains the same errors.

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

Further, we will often compare code examples in different languages, search for errors and fix them, optimize the code, primarily improving its readability and understandability.

1.3. Modeling: abstractions and reuse

The heart of programming is modeling, that is, the creation of a domain model or a model of objects and processes in the memory of the computers. Programming languages provide syntaxes for constructing constraints when creating models. Any construction and structure designed to expand functionality and introduced into the model leads to additional restrictions. Increasing the level of abstraction, on the contrary, can remove some of the restrictions and reduce the complexity of the model and the program code that expresses this model. We are constantly balancing between expanding functions and collapsing them into a more generalized model. This process can and should be iterative multiple times.

Surprisingly people are able to successfully solve tasks with complexity exceeds of their memory and thinking abilities, with the help of building models and abstractions. The accuracy of these models determines their usefulness for decision-making and the development of managerial influences. The model is always inaccurate and reflects only a small part of reality, one or more of its sides or aspects. However, in limited conditions of use, the model can be identical to the real object of the subject area. There are physical, mathematical, simulation and other models, but we will be interested primarily in informational and algorithmic models.

Abstraction is a method of generalization that reduces many different but similar cases to a single model. We are interested in data abstractions and abstract algorithms. The simplest examples of abstraction in algorithms are loops (iterative generalization) and functions (procedures and routines). With the help of a loop, we can describe many iterations with one block of commands, assuming its repetition several times, with different values of variables. Functions are also repeated many times with different arguments. Examples of data abstraction are arrays, associative arrays, lists, sets, etc. In applications there are abstractions need to be combined in layers, we call them abstraction layers. Low-level abstractions are built into the programming language (variables, functions, arrays, events). Higher-level abstractions are contained in software platforms, runtimes, standard libraries, and external libraries, or can be built independently from simple abstractions. Abstractions have such a name because they solve abstract generalized tasks (general purpose manipulations) not related to the subject area.

Building abstraction layers is perhaps the most important programming task, the successful solution of which depends on the characteristics of the software solution, such as the flexibility, the cost of modification, the ability to integrate with other systems, and the life span of the solution. We will call all layers that are not tied to the subject area and specific applied tasks system. Above the system layers, the programmer superimposes application layers, the abstraction of which, on the contrary, decreases, the universality decreases, and the application becomes more specific, being tied to specific tasks.

Abstractions of different levels can be both in the same address space (one process or one application) and in different ones. It is possible to separate them from each other and to implement interaction between them with the help of programming interfaces, modularity, component approach and simply by willpower, avoiding direct calls from the middle of one software component to the middle of another one, if the programming language or platform does not take care to prevent such an ability. This should be done even within one process, where any functions, components, and modules could be accessed from any other, even if they logically belong to different layers. The reason for this is the need to reduce the interdependence of layers and software components, ensuring their interchangeability, reuse and making their separate development possible. At the same time, it is necessary to increase the connectivity within the layers, components and modules, which ensures the growth of the productivity of the code, its ease of reading, understanding and modification. If we manage to avoid the connection between different levels of abstractions, and with the help of decomposition to ensure that one module can always be fully covered by the attention of one engineer, the development process becomes scalable, manageable and more predictable. A similar idea is the basis of microservices architecture, but the more general principle applies to any system, and it does not matter whether they are independently running microservices or modules running in the same process.

We must note that the better distributed the system is the better centralized ones. Therefore, how to solve tasks in such systems are at adequate levels, where there is already enough information for decision-making, processing and obtaining a result, there is no rigid connection of models of different levels of abstraction. With this approach, there are no unnecessary escalations of the task to higher levels, “overheating” of decision-making nodes is avoided, data transfer is minimized and operational speed is increased.

1.4. Algorithm, program, syntax, language

There are many terms associated with programming. To be sure, we should clarify the difference between them. It is better to focus on the informal understanding than the formal definition. The oldest concept here is the algorithm, which we remember from the school mathematics course. For example, Euclid’s algorithm for finding the greatest common divisor of two integers.

Algorithm

An algorithm is not yet a program, it is an idea of solving problems, described formally. So that it is understood by others, it is checked and implemented. The algorithm cannot be run, it can be converted into code in some programming language. The algorithm contains a description of operations and can be written in different ways: a formula, a block diagram, a list of actions, human language. It is always limited to a specific class of tasks, which it solves in a finite time. We can often simplify and optimize the algorithm by narrowing down the class of problems. For example, by switching from summing integers and fractions to summing only integers, we can make a more efficient implementation. It is also possible to expand the class of problems for this example by allowing input of string representations of numbers as well. This will make the algorithm more versatile, but less efficient. We have to choose what we optimize for. In this case, it is better to divide the algorithm into two, one will convert all numbers to the required data type, and the other will sum a total.

An example of the implementation of the GCD algorithm

In JavaScript, Euclid’s algorithm for finding the common denominator (common measure or greatest common divisor) can be written as follows:

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

Type — a set of values and operations that can be performed on these values.

For example, in **JavaScript** the **Boolean** type assumes two values **true** and **false**, and logical operations on them, the **Null** type assumes one **null** value, and the **Number** type is a set of rational numbers with additional restrictions on the minimum and maximum values, as well as restrictions on precision and mathematical operations **+ — * ** / % ++ — > < ≥ ≤ & | ~ ^** □ □.

Data Types

```
const values = [5, 'Kiev', true, { size: 10 }, (a) => ++a];

const types = values.map((x) => typeof x);
console.log({ types });
```

Scalar (Primitive, Atomic value) — the value of the primitive data type.

The scalar is copied on assignment and passed to the function by value.

Reference points to a value of a reference type, i.e. not a scalar value.

For **JavaScript** these are the subtypes: **Object**, **Function**, **Array**.

Structural types (Composed types) — composite types or structures, which consist of several scalar values.

Scalar values are combined into one in such a way, that a set of operations can be performed on this combined value. For example: object, array, set, tuple.

Enumerated type

Flag — a boolean value that determines the state of something.

For example: a status of a closed connection, a status of completion of a search over data structure, etc.

```
let flagName = false;
```

Sometimes flags can be called not logical but enumerated types.

String — a sequence of characters

In most languages, each character can be accessed through array access syntax, such as square brackets.

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

```
(len - 1) * f(x, INTERVAL);
```

```
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');
```

2.3. Contexts and lexical scope

Scope

```
const level = 1;

const f = () => {
  const level = 2;
  {
    const level = 3;
    console.log(level); // 3
  }
  console.log(level); // 2
};
```

```
level = 1

def f():
  level = 2
  if level == 2:
    level = 3
    print(level) // 3
  print(level) // 3

f()
```

Lexical environment

Global context

2.5. Procedural paradigm, call, stack and heap

```
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

Function definition

```
function sum(a, b) {
```

```
  return a + b;
}
```

Named function expression

```
const sum = function sum(a, b) {
  return a + b;
};
```

Anonymous function expression

```
const sum = function (a, b) {
  return a + b;
};
```

Arrow function (Lambda function)

```
const sum = (a, b) => {
  return a + b;
};
```

Lambda expression

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

```
const sum = (a) => (b) => a + b;

const hash =
  (data = {}) =>
    (key, value) => ((data[key] = value), data);
```

Superposition

```
const expr2 = sum(
  pow(mul(5, 8), 2),
  div(inc(sqrt(20)), log(2, 7))
);
```

Composition

```
const compose = (f1, f2) => (x) => f2(f1(x));
```

```
const compose = (...funcs) => (...args) =>
  funcs.reduce((args, fn) => [fn(...args)], args);
```

Partial application

```
const partial = (fn, x) => (...args) => fn(x, ...args);
```

Currying

```
const result = curry((a, b, c) => a + b + c)(1, 2)(3);
```

Side effects

Higher-order Function

Wrapper

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

console.log('Add numbers: 5 + 2 = ' + sum(5, 2));
console.log('Add floats: 5.1 + 2.3 = ' + sum(5.1, 2.3));
console.log(`Concatenate: '5' + '2' = '${sum('5', '2')}'`);
```

```
console.log('Subtraction: 5 + (-2) = ' + sum(5, -2));
```

2.7. Closures, callbacks, wrappers, and events

Closure

```
const add = (x) => (y) => {
  const z = x + y;
  console.log(`${x} + ${y} = ${z}`);
  return z;
};
```

```
const res = add(3)(6);
console.log(res);
```

```
const add = (x) => (y) => x + y;
```

Recursive closure

```
const add = (x) => (y) => {
  const z = x + y;
  console.log(`${x} + ${y} = ${z}`);
  return add(z);
};
```

```
const add = (x) => (y) => add(x + y);
```

```
const a1 = add(5);
const a2 = a1(2);
const a3 = a2(3);
const a4 = a1(1);
const a5 = a2(10);
console.log(a1, a2, a3, a4, a5);
```

Functuin chaining

```
const res = add(5)(2)(3)(7);
console.log(res);
```

Abstraction (class sunstitution)

```
const COLORS = {
  warning: '\x1b[1;33m',
  error: '\x1b[0;31m',
  info: '\x1b[1;37m',
};

const logger = (kind) => {
  const color = COLORS[kind] || COLORS.info;
  return (s) => {
    const date = new Date().toISOString();
    console.log(color + date + '\t' + s);
  };
};
```

```
const warning = logger('warning');
const error = logger('error');
const debug = logger('debug');
const slow = logger('slow');

slow('I am slow logger');
warning('Hello');
error('World');
debug('Bye!');
```

Object method chaining

```
const adder = (a) => {
  const value = () => a;
```

```
  const add = (b) => adder(a + b);
  return { add, value };
};
```

```
const v = adder(3).add(-9).add(12).value();
console.log(v);
```

Alternative syntax

```
const adder = (a) => ({
  value() {
    return a;
  },
  add(b) {
    a += b;
    return this;
  },
});
```

```
const v = adder(3).add(-9).add(12).value();
console.log(v);
```

Alternative syntax

```
const adder = (a) => ({
  value: () => a,
  add: (b) => adder(a + b),
});
```

```
const v = adder(3).add(-9).add(12).value();
console.log(v);
```

Complex example

```
const adder = (a) => {
  let onZerro = null;
  const obj = {};
  const value = () => a;
  const add = (b) => {
    let x = a + b;
    if (x < 0) {
      x = 0;
      if (onZerro) onZerro();
    }
    return adder(x);
  };
  const on = (name, callback) => {
    if (name === 'zero') onZerro = callback;
    return obj;
  };
  return Object.assign(obj, { add, value, on });
};
```

```
const a = adder(3)
  .on('zero', () => console.log('Less than zero'))
  .add(-9)
  .add(12)
  .add(5)
  .value();

console.log(a);
```

Callback

```
// Return result
const sum = (a, b) => a + b;

// Pass result to a callback
const sum = (a, b, callback) => callback(a + b);
```

```
console.log('sum(5, 2) =', sum(5, 2));
```

```
sum(5, 2, console.log.bind(null, 'sum(5, 2) ='));
```

```
const fs = require('fs');

const reader = (err, data) => {
  console.log({ lines: data.split('\n').length });
};

fs.readFile('./file.txt', 'utf8', reader);
```

Named collbacks

```
const fs = require('fs');

const print = (fileName, err, data) => {
  console.log({ lines: data.split('\n').length });
};

const fileName = './file.txt';
`
const callback = print.bind(null, fileName);
fs.readFile(fileName, 'utf8', callback);
```

Timer implementation with callback

```
const fn = () => {
  console.log('Callback from from timer');
};

const timeout = (interval, fn) => {
  setTimeout(fn, interval);
};

timeout(5000, fn);
```

Timer curry

```
const curry = (fn, ...par) => {
  const curried = (...args) => {
    if (fn.length <= args.length) return fn(...args);
    return curry(fn.bind(null, ...args));
  };
  return par.length ? curried(...par) : curried;
};
```

```
const fn = () => {
  console.log('Callback from from timer');
};

const timeout = (interval, fn) => {
  setTimeout(fn, interval);
};

const timer = curry(timeout);
timer(2000)(fn);

const timer2s = timer(2000);
timer2s(fn);
```

Iteration callbacks

```
const iterate = (array, listener) => {
  for (const item of array) {
    listener(item);
  }
};

const cities = ['Kiev', 'London', 'Beijing'];

const print = (city) => {
  console.log('City:', city);
};
```

```
iterate(cities, print);
```

Events

```
const adder = (initial) => {
  let value = initial;
  const add = (delta) => {
    value += delta;
    if (value >= add.maxValue) add.maxEvent(value);
    return add;
  };
  add.max = (max, event) => {
    add.maxValue = max;
    add.maxEvent = event;
    return add;
  };
  return add;
};
```

```
const maxReached = (value) => {
  console.log('max value reached, value: ' + value);
};

const a1 = adder(10).max(100, maxReached)(-12);

a1(25);
a1(50);
a1(75);
a1(100);
a1(-200)(50)(30);
```

EventEmitter

```
const { EventEmitter } = require('events');
```

```
const emitter = new EventEmitter();

emitter.on('new city', (city) => {
  console.log('Emitted city:', city);
});

emitter.on('data', (array) => {
  console.log(array.reduce((a, b) => a + b));
});

emitter.emit('new city', 'Delhi');
emitter.emit('new city', 'Berlin');
emitter.emit('new city', 'Tokyo');
emitter.emit('data', [5, 10, 7, -3]);
```

2.8. Exceptions and error handling

Throw

```
const isNumber = (value) => typeof value === 'number';

const sum = (a, b) => {
  if (isNumber(a) && isNumber(b)) {
    return a + b;
  }
  throw new Error('a and b should be numbers');
};
```

```
try {
  console.log(sum(2, 3));
} catch (err) {
  console.log(err.message);
}
```

```
try {
  console.log(sum(7, 'A'));
```

```
} catch (err) {
  console.log(err.message);
}
```

Return tuple or struct

```
const sum = (a, b) => {
  if (isNumber(a) && isNumber(b)) {
    return [null, a + b];
  }
  return [new Error('a and b should be numbers')];
};

console.log(sum(2, 3));

console.log(sum(7, 'A'));
```

Callback

```
const sum = (a, b, callback) => {
  if (isNumber(a) && isNumber(b)) {
    callback(null, a + b);
  } else {
    callback(new Error('a and b should be numbers'));
  }
};
```

```
sum(2, 3, (err, result) => {
  if (err) {
    console.log(err.message);
    return;
  }
  console.log(result);
});
```

```
sum(7, 'A', (err, result) => {
```

```
  if (err) {
    console.log(err.message);
    return;
  }
  console.log(result);
});
```

Promise

```
const sum = (a, b) =>
  new Promise((resolve, reject) => {
    if (isNumber(a) && isNumber(b)) {
      resolve(a + b);
    } else {
      reject(new Error('a and b should be numbers'));
    }
  });
```

```
sum(2, 3)
  .then((data) => {
    console.log(data);
  })
  .catch((err) => {
    console.log(err.message);
  });
```

```
sum(7, 'A')
  .then((data) => {
    console.log(data);
  })
  .catch((err) => {
    console.log(err.message);
  });
```

Async throw

```
const sum = async (a, b) => {
  if (isNumber(a) && isNumber(b)) {
    return a + b;
  }
  throw new Error('a and b should be numbers');
};
```

```
try {
  console.log(await sum(2, 3));
} catch (e) {
  console.log(e.message);
}
```

```
try {
  console.log(await sum(7, 'A'));
} catch (err) {
  console.log(err.message);
}
```

2.9. Tasks

No translation

3. Application state, data structures and collections

No translation

3.1. Stateful and stateless approach

No translation

3.2. Structs and records

```
#include <stdio.h>

struct date {
  int day;
  int month;
  int year;
};

struct person {
  char *name;
  char *city;
  struct date born;
};

int main() {
  struct person p1;
  p1.name = "Marcus";
  p1.city = "Roma";
  p1.born.day = 26;
  p1.born.month = 4;
  p1.born.year = 121;

  printf(
    "Name: %s\nCity: %s\nBorn: %d-%d-%d\n",
    p1.name, p1.city,
    p1.born.year, p1.born.month, p1.born.day
  );
```

```
  return 0;
}
```

Pascal

```
program Example;

type TDate = record
  Year: integer;
  Month: 1..12;
  Day: 1..31;
end;

type TPerson = record
  Name: string[10];
  City: string[10];
  Born: TDate;
end;

var
  P1: TPerson;
  FPerson: File of TPerson;

begin
  P1.Name := 'Marcus';
  P1.City := 'Roma';
  P1.Born.Day := 26;
  P1.Born.Month := 4;
  P1.Born.Year := 121;
  WriteLn('Name: ', P1.Name);
  WriteLn('City: ', P1.City);
  WriteLn(
    'Born: ',
    P1.Born.Year, '-',
    P1.Born.Month, '-',
    P1.Born.Day
  );
  Assign(FPerson, './record.dat');
  Rewrite(FPerson);
```

```
  Write(FPerson, P1);
  Close(FPerson);
end.
```

Rust

```
struct Date {
    year: u32,
    month: u32,
    day: u32,
}

struct Person {
    name: String,
    city: String,
    born: Date,
}

fn main() {
    let p1 = Person {
        name: String::from("Marcus"),
        city: String::from("Roma"),
        born: Date {
            day: 26,
            month: 4,
            year: 121,
        },
    };

    println!(
        "Name: {}\nCity: {}\nBorn: {}-{}-{}\n",
        p1.name, p1.city,
        p1.born.year, p1.born.month, p1.born.day
    );
}
```

TypeScript: Interfaces

```
interface IDate {
  day: number;
  month: number;
  year: number;
}

interface IPerson {
  name: string;
  city: string;
  born: IDate;
}
```

```
const personToString = (person: IPerson): string => {
  const { name, city, born } = person;
  const { year, month, day } = born;
  const fields = [
    `Name: ${name}`,
    `City: ${city}`,
    `Born: ${year}-${month}-${day}`,
  ];
  return fields.join('\n');
};
```

```
const person: IPerson = {
  name: 'Marcus',
  city: 'Roma',
  born: {
    day: 26,
    month: 4,
    year: 121,
  },
};

console.log(personToString(person));
```

TypeScript: Classes

```
class DateStruct {
  day: number;
  month: number;
  year: number;
}

class Person {
  name: string;
  city: string;
  born: DateStruct;
}
```

JavaScript: Classes

```
class DateStruct {
  constructor(year, month, day) {
    this.day = day;
    this.month = month;
    this.year = year;
  }
}

class Person {
  constructor(name, city, born) {
    this.name = name;
    this.city = city;
    this.born = born;
  }
}
```

```
const personToString = (person) => {
  const { name, city, born } = person;
  const { year, month, day } = born;
  const fields = [
    `Name: ${name}`,
    `City: ${city}`,
    `Born: ${year}-${month}-${day}`,
  ];
```

```
  return fields.join('\n');
};
```

```
const date = new DateStruct(121, 4, 26);
const person = new Person('Marcus', 'Roma', date);
console.log(personToString(person));
```

JavaScrip: Objects

```
const person = {
  name: 'Marcus',
  city: 'Roma',
  born: {
    day: 26,
    month: 4,
    year: 121,
  },
};

console.log(personToString(person));
```

JavaScrit: strucute serialization

```
const v8 = require('v8');
const fs = require('fs');
```

Take from previous example:

- class DateStruct
- class Person

```
const date = new DateStruct(121, 4, 26);
const person = new Person('Marcus', 'Roma', date);

const v8Data = v8.serialize(person);
const v8File = './file.dat';
```

```
fs.writeFile(v8File, v8Data, () => {
  console.log('Saved ' + v8File);
});
```

File: file.dat

```
FF 0D 6F 22 04 6E 61 6D 65 22 06 4D 61 72 63 75
73 22 04 63 69 74 79 22 04 52 6F 6D 61 22 04 62
6F 72 6E 6F 22 03 64 61 79 49 34 22 05 6D 6F 6E
74 68 49 08 22 04 79 65 61 72 49 F2 01 7B 03 7B
03
```

Nested structures

```
#include <stdio.h>
#include <map>
#include <string>
#include <vector>

struct Product {
  std::string name;
  int price;
};

void printProduct(Product item) {
  printf("%s: %d\n", item.name.c_str(), item.price);
}

void printProducts(std::vector<Product> items) {
  for (int i = 0; i < items.size(); i++) {
    printProduct(items[i]);
  }
}

int main() {
  std::map<std::string, std::vector<Product>> purchase {
    { "Electronics", {
```

```
      { "Laptop", 1500 },
      { "Keyboard", 100 },
      { "HDMI cable", 10 },
    } },
    { "Textile", {
      { "Bag", 50 },
    } },
  };

  std::vector electronics = purchase["Electronics"];
  printf("Electronics:\n");
  printProducts(electronics);

  std::vector textile = purchase["Textile"];
  printf("\nTextile:\n");
  printProducts(textile);

  Product bag = textile[0];
  printf("\nSingle element:\n");
  printProduct(bag);

  int price = purchase["Electronics"][2].price;
  printf("\nHDMI cable price is %d\n", price);
}
```

Python

```
purchase = {
  'Electronics': [
    { 'name': 'Laptop', 'price': 1500 },
    { 'name': 'Keyboard', 'price': 100 },
    { 'name': 'HDMI cable', 'price': 10 },
  ],
  'Textile': [
    { 'name': 'Bag', 'price': 50 },
  ],
}

electronics = purchase['Electronics']
```

```
print({ 'electronics': electronics })

textile = purchase['Textile']
print({ 'textile': textile })

bag = textile[0]
print({ 'bag': bag })

price = purchase['Electronics'][2]['price']
print({ 'price': price })
```

JavaScript

```
const purchase = {
  Electronics: [
    { name: 'Laptop', price: 1500 },
    { name: 'Keyboard', price: 100 },
    { name: 'HDMI cable', price: 10 },
  ],
  Textile: [{ name: 'Bag', price: 50 }],
};

const electronics = purchase.Electronics;
console.log(electronics);

const textile = purchase['Textile'];
console.log(textile);

const bag = textile[0];
console.log(bag);

const price = purchase['Electronics'][2].price;
console.log(price);

const json = JSON.stringify(purchase);
console.log(json);
const obj = JSON.parse(json);
console.log(obj);
```

3.3. Array, list, set, tuple

Array

```
const ages = [10, 12, 15, 15, 17, 18, 18, 19, 20];

const first = ages[0];
const last = ages[ages.length - 1];

console.log({ first, last });
```

List

```
ages = [10, 12, 15, 15, 17, 18, 18, 19, 20]

first = ages[0]
last = ages[-1]

print({ 'first': first, 'last': last })
```

Array

```
#include <stdio.h>

int main() {
  int ages[] = { 10, 12, 15, 15, 17, 18, 18, 19, 20 };

  int first = ages[0];
  int length = sizeof(ages) / sizeof(ages[0]);
  int last = ages[length - 1];

  printf("first: %d\n", first);
  printf("last: %d\n", last);
}
```

Access elements

```
const getFirstAndLast = (array) => {
  const first = array[0];
  const last = array[array.length - 1];
  return { first, last };
};
```

```
const ages = [10, 12, 15, 15, 17, 18, 18, 19, 20];
const { first, last } = getFirstAndLast(ages);
console.log({ first, last });
```

```
def getFirstAndLast(array):
  first = ages[0]
  last = ages[-1]
  return first, last
```

```
ages = [10, 12, 15, 15, 17, 18, 18, 19, 20]
first, last = getFirstAndLast(ages)
print({ 'first': first, 'last': last })
```

```
const getFirstAndLast = (array) => ({
  first: array[0],
  last: array[array.length - 1],
});
```

```
getFirstAndLast = lambda array: (array[0], array[-1])
```

```
const concat = (arr1, arr2) => {
  const arr = arr1.slice();
  arr.push(...arr2);
  return arr;
};
```

```
const schoolAges = [10, 12, 15, 15];
const studentAges = [17, 18, 18, 19, 20];
const ages = concat(schoolAges, studentAges);
```

```
concat = lambda arr1, arr2: arr1 + arr2
```

```
schoolAges = [10, 12, 15, 15]
studentAges = [17, 18, 18, 19, 20]
ages = schoolAges + studentAges
```

```
#include <stdio.h>

int main() {
  int schoolAges[] = { 10, 12, 15, 15 };
  int studentAges[] = { 17, 18, 18, 19, 20 };

  int schoolLength = sizeof(schoolAges) / sizeof(int);
  int studentLength = sizeof(studentAges) / sizeof(int);
  int length = schoolLength + studentLength;
  int ages[length];

  for (int i = 0; i < schoolLength; i++) {
    ages[i] = schoolAges[i];
  }

  for (int i = 0; i < studentLength; i++) {
    ages[i + schoolLength] = studentAges[i];
  }

  for (int i = 0; i < length; i++) {
    printf("ages[%d]: %d\n", i, ages[i]);
  }
}
```

```
const concat = (arr1, arr2) => [...arr1, ...arr2];
```

Set

```
const ages = new Set([10, 12, 15, 15, 17, 18, 18, 19, 20]);
console.log({ ages });

ages.add(16);
ages.delete(20);

console.log({
  10: ages.has(10),
  16: ages.has(16),
  19: ages.has(19),
  20: ages.has(20),
});

ages.clear();
console.log({ ages });
```

Union

```
const union = (s1, s2) => {
  const ds = s1.slice(0);
  for (let i = 0; i < s2.length; i++) {
    const item = s2[i];
    if (!ds.includes(item)) ds.push(item);
  }
  return ds;
};
```

```
const cities1 = ['Beijing', 'Kiev'];
const cities2 = ['Kiev', 'London', 'Baghdad'];
console.dir({ cities1, cities2 });

const results = union(cities1, cities2);
console.dir(results);
```

```js
const union = (s1, s2) => new Set([...s1, ...s2]);
```

Intersection

```js
const intersection = (s1, s2) => {
  const ds = [];
  for (let i = 0; i < s1.length; i++) {
    const item = s1[i];
    if (s2.includes(item)) ds.push(item);
  }
  return ds;
};
```

```js
const intersection = (s1, s2) =>
  new Set([...s1].filter((v) => s2.has(v)));
```

Difference

```js
const difference = (s1, s2) => {
  const ds = [];
  for (let i = 0; i < s1.length; i++) {
    const item = s1[i];
    if (!s2.includes(item)) ds.push(item);
  }
  return ds;
};
```

```js
const difference = (s1, s2) =>
  new Set([...s1].filter((v) => !s2.has(v)));
```

Complement

```js
const complement = (s1, s2) => difference(s2, s1);
```

3.4. Dictionary, hash table and associative array

Object

```
const ages = {
  'Vasia Pupkin': 19,
  'Marcus Aurelius': 1860,
};

console.log({ ages });
```

```
ages['Vasia Pupkin'] = 20;
console.log({ ages });
```

```
Reflect.deleteProperty(ages, 'Vasia Pupkin');
console.log({ ages });
```

```
console.log({
  'Vasia Pupkin': Reflect.has(ages, 'Vasia Pupkin'),
  'Marcus Aurelius': Reflect.has(ages, 'Marcus Aurelius'),
});
```

Map

```
const ages = new Map();

ages.set('Vasia Pupkin', 19);
ages.set('Marcus Aurelius', 1860);

console.log({ ages });
```

```
ages.set('Vasia Pupkin', 20);
console.log({ ages });
```

```
ages.delete('Vasia Pupkin');
console.log({ ages });
```

```
console.log({
  'Vasia Pupkin': ages.has('Vasia Pupkin'),
  'Marcus Aurelius': ages.has('Marcus Aurelius'),
});
```

```
ages.clear();
console.log({ ages });
```

Complex example: distinct

```
const distinct = (dataset) => {
  const keys = new Set();
  return dataset.filter((record) => {
    const cols = Object.keys(record).sort();
    const key = cols
      .map((field) => record[field])
      .join('\x00');
    const has = keys.has(key);
    if (!has) keys.add(key);
    return !has;
  });
};
```

```
const flights = [
  { from: 'Kiev', to: 'Rome' },
  { from: 'Kiev', to: 'Warsaw' },
  { from: 'Dublin', to: 'Riga' },
  { from: 'Riga', to: 'Dublin' },
  { from: 'Kiev', to: 'Rome' },
  { from: 'Cairo', to: 'Paris' },
];

console.table(flights);
```

```
const directions = distinct(flights);
console.table(directions);
```

3.5. Stack, queue, deque

```
class Stack {
  constructor() {
    this.last = null;
  }

  push(item) {
    const prev = this.last;
    const element = { prev, item };
    this.last = element;
  }

  pop() {
    const element = this.last;
    if (!element) return null;
    this.last = element.prev;
    return element.item;
  }
}
```

```
const obj1 = { name: 'first' };
const obj2 = { name: 'second' };
const obj3 = { name: 'third' };
const list = new Stack();
list.push(obj1);
list.push(obj2);
list.push(obj3);
```

```
console.dir(list.pop()); // { name: 'third' }
console.dir(list.pop()); // { name: 'second' }
console.dir(list.pop()); // { name: 'first' }
console.dir(list.pop()); // null
```

```
class Queue {
  constructor() {
    this.first = null;
    this.last = null;
  }

  put(item) {
    const last = this.last;
    const element = { next: null, item };
    if (last) {
      last.next = element;
      this.last = element;
    } else {
      this.first = element;
      this.last = element;
    }
  }

  pick() {
    const element = this.first;
    if (!element) return null;
    if (this.last === element) {
      this.first = null;
      this.last = null;
    } else {
      this.first = element.next;
    }
    return element.item;
  }
}
```

```
const obj1 = { name: 'first' };
const obj2 = { name: 'second' };
const obj3 = { name: 'third' };

const queue = new Queue();
queue.put(obj1);
```

```
queue.put(obj2);
queue.put(obj3);

console.dir(queue.pick()); // { name: 'first' }
console.dir(queue.pick()); // { name: 'second' }
console.dir(queue.pick()); // { name: 'third' }
console.dir(queue.pick()); // null
```

Dequeue

```
class Dequeue {
  constructor() {
    this.first = null;
    this.last = null;
  }

  push(item) {
    const last = this.last;
    const element = { prev: last, next: null, item };
    if (last) {
      last.next = element;
      this.last = element;
    } else {
      this.first = element;
      this.last = element;
    }
  }

  pop() {
    const element = this.last;
    if (!element) return null;
    if (this.first === element) {
      this.first = null;
      this.last = null;
    } else {
      this.last = element.prev;
    }
    return element.item;
  }
```

```
  unshift(item) {
    const first = this.first;
    const element = { prev: null, next: first, item };
    if (first) {
      first.prev = element;
      this.first = element;
    } else {
      this.first = element;
      this.last = element;
    }
  }

  shift() {
    const element = this.first;
    if (!element) return null;
    if (this.last === element) {
      this.first = null;
      this.last = null;
    } else {
      this.first = element.next;
    }
    return element.item;
  }
}
```

```
const obj1 = { name: 'first' };
const obj2 = { name: 'second' };
const obj3 = { name: 'third' };

const list = new Dequeue();
list.push(obj1);
list.push(obj2);
list.unshift(obj3);

console.dir(list.pop()); // { name: 'second' }
console.dir(list.shift()); // { name: 'third' }
console.dir(list.shift()); // { name: 'first' }
```

3.6. Trees and Graphs

Adjacency matrix

```
    1            +---+
  +--------------+ B +--+
  |              +---+  |
  |                     |
+-+-+    2              |
| A +----------+        |3
+-+-+          |        |
  |            |        |
  |          +-+-+      |
  |5         | C |      |
  |          +-+-+      |
  +--+         |        |
     |         |        |
   +-+-+     4 |      +-+-+
   | E |       +------+ D |
   +---+              +---+
```

```
const graph = [
  [0, 1, 1, 0, 1],
  [1, 0, 0, 1, 0],
  [1, 0, 0, 1, 0],
  [0, 1, 1, 0, 0],
  [1, 0, 0, 0, 0],
];

const graph = {
  A: [0, 1, 1, 0, 1],
  B: [1, 0, 0, 1, 0],
  C: [1, 0, 0, 1, 0],
  D: [0, 1, 1, 0, 0],
  E: [1, 0, 0, 0, 0],
};
```

Adjacency matrix as a plain array

```
    1            +---+
  +--------------+ B +--+
  |              +---+  |
  |                     |
+-+-+    2              |
| A +----------+        |3
+-+-+          |        |
  |            |        |
  |          +-+-+      |
  |5         | C |      |
  |          +-+-+      |
  +--+         |        |
     |         |        |
   +-+-+     4 |      +-+-+
```

```
const graph = [
  0, 1, 1, 0, 1,
  1, 0, 0, 1, 0,
  1, 0, 0, 1, 0,
  0, 1, 1, 0, 0,
  1, 0, 0, 0, 0,
];
```

```
   | E |       +------+ D |
   +---+              +---+
```

Incidence matrix — Vertex connections (array rows) with edges (array columns)

```
    1            +---+               const graph = [
  +--------------+ B +--+              [1, 1, 0, 0, 1],
  |              +---+  |              [1, 0, 1, 0, 0],
  |                     |              [0, 1, 0, 1, 0],
+-+-+    2              |              [0, 0, 1, 1, 0],
| A +----------+        |3             [0, 0, 0, 0, 1],
+-+-+          |        |            ];
  |            |        |
  |          +-+-+      |            const graph = {
  |5         | C |      |              A: [1, 1, 0, 0, 1],
  |          +-+-+      |              B: [1, 0, 1, 0, 0],
  +--+         |        |              C: [0, 1, 0, 1, 0],
     |         |        |              D: [0, 0, 1, 1, 0],
   +-+-+     4 |      +-+-+            E: [0, 0, 0, 0, 1],
   | E |       +------+ D |          ];
   +---+              +---+
```

List of adjacency — List of vertexes with list of adjacent vertices for each

```
    1            +---+               const graph = {
  +--------------+ B +--+              A: [],
  |              +---+  |              B: [],
  |                     |              C: [],
+-+-+    2              |              D: [],
| A +----------+        |3             E: [],
+-+-+          |        |            };
  |            |        |
  |          +-+-+      |            const { A, B, C, D, E } =
graph;
  |5         | C |      |            A.push(B, C, E);
```

```
  |          +-+-+      |            B.push(A, D);
  +--+         |        |            C.push(A, D);
     |         |        |            D.push(B, C);
   +-+-+     4 |      +-+-+          E.push(A);
   | E |       +------+ D |
   +---+              +---+          console.dir({ graph });
```

List of edges — a list of edges as a pair of vertices

```
    1            +---+               const graph = [
  +--------------+ B +--+               [A, B],
  |              +---+  |               [A, C],
  |                     |               [B, D],
+-+-+    2              |               [C, D],
| A +----------+        |3              [A, E],
+-+-+          |        |            ];
  |            |        |
  |          +-+-+      |            const graph = [
  |5         | C |      |               { from: A, to: B },
  |          +-+-+      |               { from: A, to: C },
  +--+         |        |               { from: B, to: D },
     |         |        |               { from: C, to: D },
   +-+-+     4 |      +-+-+             { from: A, to: E },
   | E |       +------+ D |          ];
   +---+              +---+
```

3.7. Dataset projections

No translation

3.8. Computational complexity estimation

No translation

4. Extended concepts

No translation

4.1. What is a technology stack

No translation

4.2. Development environment and debugging

No translation

4.3. Iterations: recursion, iterators, and generators

No translation

4.4. Application building blocks: files, modules, components

No translation

4.5. Object, prototype and class

No translation

4.6. Partial application and currying, pipe and compose

No translation

4.7. Chaining for methods and functions

No translation

4.8. Mixins

No translation

4.9. Dependencies and libraries

No translation