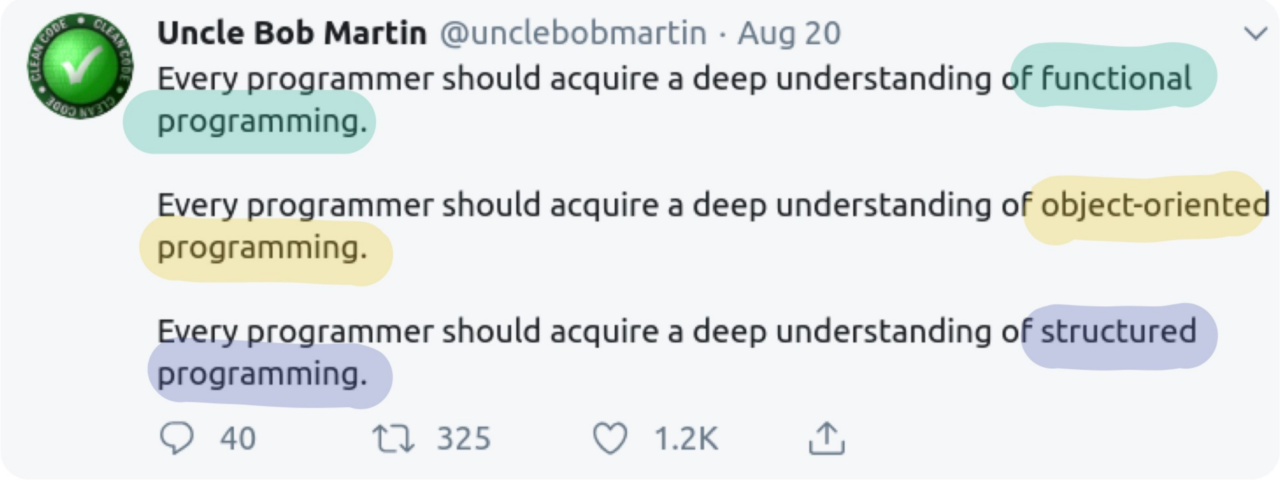
Functional vs Structured vs Procedural vs O-O programming



Uncle Bob Martin @unclebobmartin · Aug 20

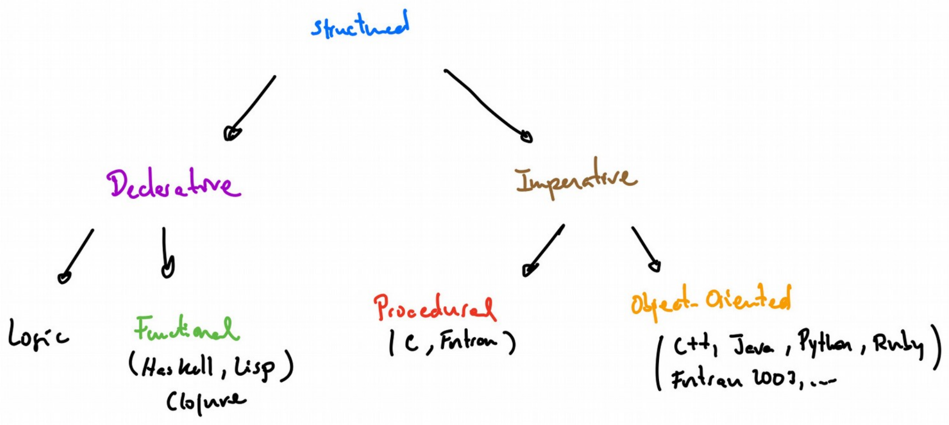
Every programmer should acquire a deep understanding of functional programming.

Every programmer should acquire a deep understanding of object-oriented programming.

Every programmer should acquire a deep understanding of structured programming.

Non-structured programming : Go to ⇒ Line to line

structured programming : Data-structures, functions, classes....



1. Brief introduction

Imperative programming

- focuses on how to execute, defines control flow as statements that change a program state

Declarative programming

- focuses on what to execute, defines program logic, but not detailed control flow

Procedural programming, structured programming –

- specifies the steps a program must take to reach a desired state.

Functional programming

- treats programs as evaluating mathematical functions
- avoids state and mutable data

Object-oriented programming (OOP)

- organizes programs as objects
- data structures consisting of data-fields
- methods together with their interactions.

2. Small considerations

- All of them are good in their own ways
- They're simply different approaches to the same problems.
- the "advantage" of each paradigm is simply in the modeling:
  - your algorithms
  - data structures.
- The choice of which you use is simply
  - what makes more sense for your project
  - the abstractions your language supports.
- These paradigms **don't have** to be **mutually exclusive**
- If you look at **python:**
  - it supports functions and classes,
  - a function is a first-class citizen,
  - But it can contain arbitrary number of statements.
  - So you can have a function that contains procedural code, but you can pass it around just like functional languages.
  - but at the same time, everything is an object, including functions.
  - You can mix and match functional/oop/procedural style all in one piece of code.
- It is not a question of the language neither.
- You can write functional, procedural or object-oriented in almost any popular language, although it might be some additional effort in some.
- What makes more sense to model?
  - your code as the composition of functions?,
  - your data as objects?

3. Some statements comparisons

a) First comparison

Procedural is good for a model that follows a procedure,
OOP is good for design,
Functional is good for high level programming

b) Second comparison

Functional programming is like **describing your problem to a mathematician**.
Imperative programming is like **giving instructions to an idiot**.

c) Third comparison

procedural code: focus on “**Steps” and an ordered** way of writing a program? You’re likely writing.
OOP: focusing on **state transformations** and encapsulated abstractions,

d) Fourth comparison

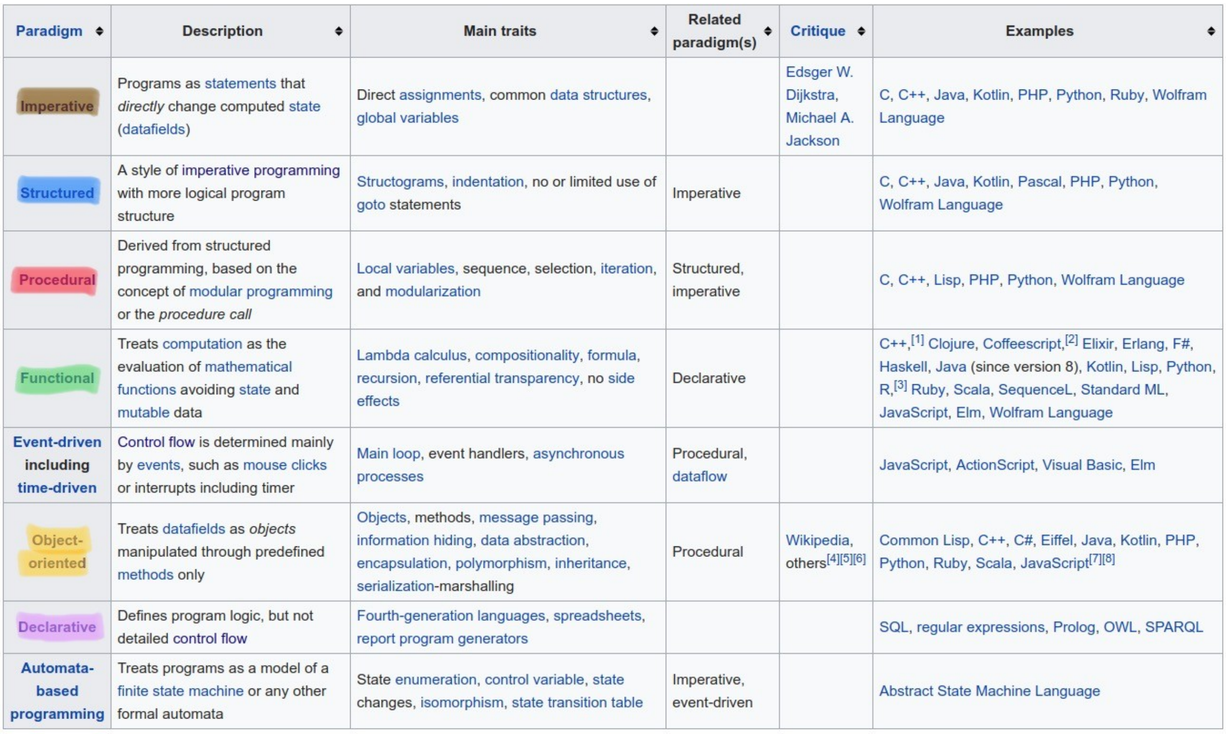
procedural languages: you might use a for loop,
functional languages: use recursive calls to functions to perform the same task.

e) Fifth comparison

procedural style: data tends to be highly decoupled from the functions that operate on it.
object oriented style: data tends to carry with it a collection of functions

e) Sixth comparison

functional algorithm: your algorithm involves lists and trees,
object-oriented: your data is highly structured



Paradigm	Description	Main traits	Related paradigm(s)	Critique	Examples
Imperative	Programs as statements that *directly* change computed state (datafields)	Direct assignments, common data structures, global variables		Edsger W. Dijkstra, Michael A. Jackson	C, C++, Java, Kotlin, PHP, Python, Ruby, Wolfram Language
Structured	A style of imperative programming with more logical program structure	Structograms, indentation, no or limited use of goto statements	Imperative		C, C++, Java, Kotlin, Pascal, PHP, Python, Wolfram Language
Procedural	Derived from structured programming, based on the concept of modular programming or the *procedure call*	Local variables, sequence, selection, iteration, and modularization	Structured, imperative		C, C++, Lisp, PHP, Python, Wolfram Language
Functional	Treats computation as the evaluation of mathematical functions avoiding state and mutable data	Lambda calculus, compositionality, formula, recursion, referential transparency, no side effects	Declarative		C++,[1] Clojure, Coffeescript,[2] Elixir, Erlang, F#, Haskell, Java (since version 8), Kotlin, Lisp, Python, R,[3] Ruby, Scala, SequenceL, Standard ML, JavaScript, Elm, Wolfram Language
Event-driven including time-driven	Control flow is determined mainly by events, such as mouse clicks or interrupts including timer	Main loop, event handlers, asynchronous processes	Procedural, dataflow		JavaScript, ActionScript, Visual Basic, Elm
Object-oriented	Treats datafields as *objects* manipulated through predefined methods only	Objects, methods, message passing, information hiding, data abstraction, encapsulation, polymorphism, inheritance, serialization-marshalling	Procedural	Wikipedia, others[4][5][6]	Common Lisp, C++, C#, Eiffel, Java, Kotlin, PHP, Python, Ruby, Scala, JavaScript[7][8]
Declarative	Defines program logic, but not detailed control flow	Fourth-generation languages, spreadsheets, report program generators			SQL, regular expressions, Prolog, OWL, SPARQL
Automata-based programming	Treats programs as a model of a finite state machine or any other formal automata	State enumeration, control variable, state changes, isomorphism, state transition table	Imperative, event-driven		Abstract State Machine Language

4. Describing each of them

Structured programming

- goes back to Djikstra's "Goto Considered Harmful" paper.
- using if/then/else/elif structures, do/while/until/for loops, etc. instead of resorting to goto.
- It's essentially abstracting a bit away from the compare/branch machine level instructions.
- Orthogonal to both functional and procedural programming.
- is an old term that I think would encompass functional, procedural, and much else. It basically means using explicit control-flow structures rather than jumping about directly from instruction to instruction.

Functional programming

- the structure given to your code corresponds to its meaning
- a program is a function that changes the state of the world.
- using functions as **first-class elements**
- Making use of higher order functions (taking and/or returning functions);
- leading to powerful constructs and well factored code.
- Functions should always return the same result, given the same input
- **Avoiding side effects**
- data and functions tend toward having more in common with each other (as in Lisp and Scheme) while offering more flexibility in terms of how functions are actually used.
- Algorithms tend also to be defined in terms of **recursion** and **composition** rather than **loops** and **iteration**.
- Haskell
- there are no statements
- functions are only allowed one expression inside them
- functions are first-class citizens, you can pass them around as parameters
- functions first-class objects (you can pass them around like you would a number) but you can't do that in C
- Functional programming refers to the ability to treat functions as values.
- Let's consider an analogy with "regular" values.
- We can take two integer values and combine them using the + operator to obtain a new integer.
- Or we can multiply an integer by a floating point number to get a floating point number.
- we can combine two function values to produce a new function value using operators like compose or lift.
- Or we can combine a function value and a data value to produce a new data value using operators like map or fold.
- Note that many languages have functional programming capabilities -- even languages that are not usually thought of as functional languages.
- Even Grandfather FORTRAN supported function values, although it did not offer much in the way of function-combining operators. For a language to be called "functional", it needs to embrace functional programming capabilities in a big way.
- Functional programming or FP is a way of thinking about software construction based on some fundamental defining principles
- Functional programming concepts focuses on results, not the process

- The objective of any FP language is to mimic the mathematical functions
- Some most prominent Functional programming languages: 1)Haskell 2)SM 3) Clojure 4) Scala 5) Erlang 6) Clean
- A 'Pure function' is a function whose inputs are declared as inputs and none of them should be hidden. The outputs are also declared as outputs.
- Immutable Data means that you should easily able to create data structures instead of modifying ones which is already exist
- Allows you to avoid confusing problems and errors in the code
- Functional code is not easy, so it is difficult to understand for the beginner
- FP uses Immutable data while OOP uses Mutable data

The benefits of functional programming

- Allows you to avoid confusing problems and errors in the code
- Easier to test and execute Unit testing and debug FP Code.
- Parallel processing and concurrency
- Hot code deployment and fault tolerance
- Offers better modularity with a shorter code
- Increased productivity of the developer
- Supports Nested Functions
- Functional Constructs like Lazy Map & Lists, etc.
- Allows effective use of Lambda Calculus

Limitations of Functional Programming

- Functional programming paradigm is **not easy**, so it is difficult to understand for the beginner
- **Hard to maintain** as many objects evolve during the coding
- Needs lots of mocking and extensive environmental setup
- **Re-use** is very **complicated**
- **Needs** constantly **refactoring**
- Objects may not represent the problem correctly

Procedural Programming

- ability to encapsulate a common sequence of instructions into a procedure so that those instructions can be invoked from many places without resorting to copy-and-paste.
- As procedures were a very early development in programming, the capability is almost invariably linked with the style of programming demanded by machine- or assembly-language programming: a style that emphasizes the notion of storage locations and instructions that move data between those locations.
- imperative "subroutines" (as opposed to pure "functions") consisting generally of a series of "statements" (as opposed to "expressions") leaving behind side effects.
- C
- The only way you can pass functions around is by using function pointers
- That alone doesn't enable many powerful tasks.
- is what you'd consider "typical" programming in any C language or its descendants, including OO languages such as Java and C++.
- a program is a series of instructions, to be executed serially, and invoking subprocedures along the way.

Object oriented programming

- Java
- The only way to pass a function around is to wrap it in an object that implements that function, and then pass that object around.
- For GUI I'd say that is very well suited, the Window is an Object, the Textboxes are Objects, and the Okay-Button is one too.

5. Some comparisons

Functional Programming	**OOP**
FP uses Immutable data.	OOP uses Mutable data.
Follows **Declarative** Programming based Model.	Follows **Imperative** Programming Model.
What it focuses is on: "**What** you are doing. in the programme."	What it focuses is on "**How** you are doing your programming."
Supports Parallel Programming.	**No supports** for Parallel Programming.
Its functions have **no-side effects**.	Method can produce **many side effects**.
Flow Control is performed using function calls & **function calls with recursion**.	Flow control process is conducted using **loops and conditional statements.**
Execution order of statements is **not very important.**	**Execution order** of statements **is important.**
Supports both "**Abstraction over Data**" and "**Abstraction over Behavior**."	Supports only "**Abstraction over Data**".

Functional Programming	**OOP**
Uses Immutable data.	Uses Mutable data.
Follows Declarative Programming Model.	Follows Imperative Programming Model.
Focus is on: “**What** you are doing”	Focus is on “**How** you are doing”
Supports Parallel Programming	Not suitable for Parallel Programming
Its functions have no-side effects	Its methods can produce serious side effects.
Flow Control is done using function calls & function calls with recursion	Flow control is done using loops and conditional statements.
It uses "Recursion" concept to iterate Collection Data.	It uses "Loop" concept to iterate Collection Data. For example: For-each loop in Java
Execution order of statements is not so important.	Execution order of statements is very important.
Supports both "Abstraction over Data" and "Abstraction over Behavior".	Supports only "Abstraction over Data".

6. Examples

a) OddWords

Procedural

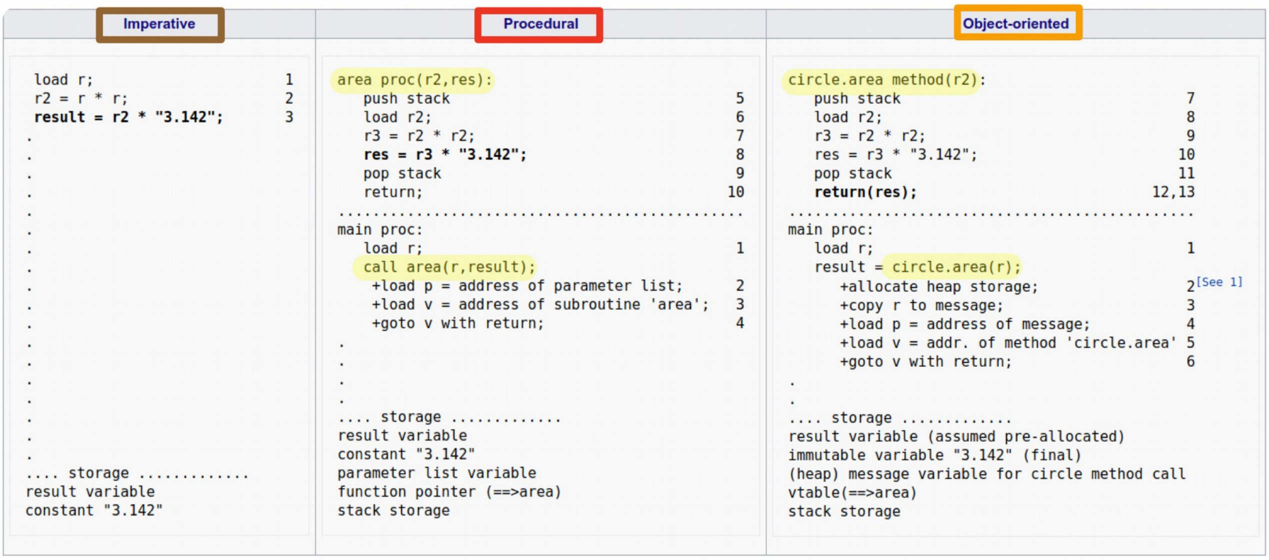
```
function allOdd(words) {
  var result = true;
  for (var i = 0; i < length(words); ++i) {
    var len = length(words[i]);
    if (!odd(len)) {
      result = false;
      break;
    }
  }
  return result;
}
```

Functional

```
function allOdd(words) {
  return apply(and, map(compose(odd, length), words));
}
```

- **compose**(odd, length) combines the odd and length functions to produce a new function that determines whether the length of a string is odd.
- **map**(..., words) calls that new function for each element in words, ultimately returning a new list of boolean values, each indicating whether the corresponding word has an odd number of characters.
- **apply**(and, ...) applies the "and" operator to the resulting list, and-ing all of the booleans together to yield the final result.

b) Area computation



Imperative

```
 load r;                      1
 r2 = r * r;                  2
 result = r2 * "3.142";       3
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.... storage .............
result variable
constant "3.142"
```

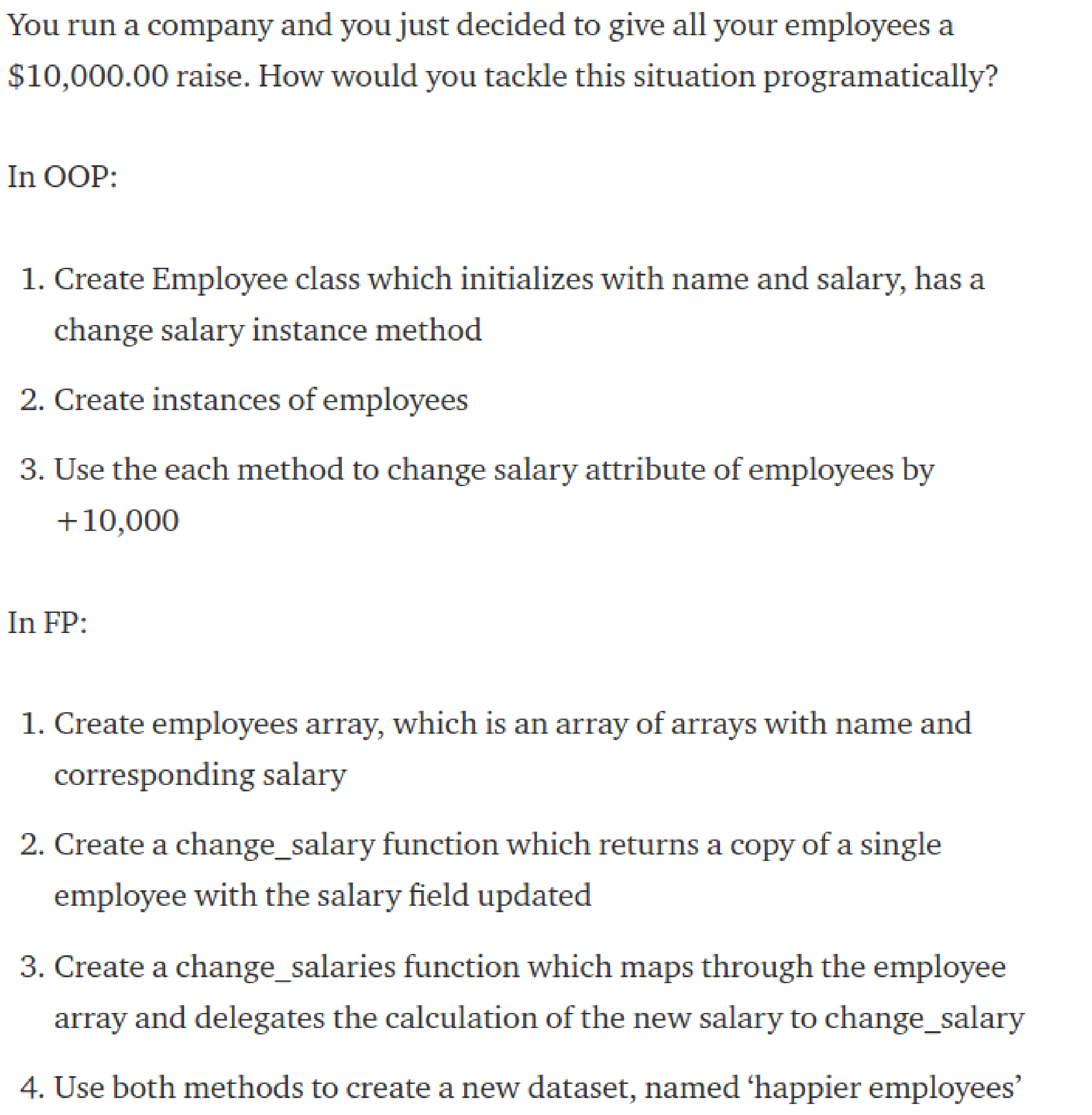
Procedural

```
area proc(r2,res):
   push stack                                 5
   load r2;                                   6
   r3 = r2 * r2;                              7
   res = r3 * "3.142";                        8
   pop stack                                  9
   return;                                   10
...............................................
main proc:
   load r;                                    1
   call area(r,result);
    +load p = address of parameter list;      2
    +load v = address of subroutine 'area';   3
    +goto v with return;                      4
.
.
.
.
.... storage .............
result variable
constant "3.142"
parameter list variable
function pointer (==>area)
stack storage
```

Object-oriented

```
circle.area method(r2):
   push stack                                  7
   load r2;                                    8
   r3 = r2 * r2;                               9
   res = r3 * "3.142";                        10
   pop stack                                  11
   return(res);                            12,13
...............................................
main proc:
   load r;                                     1
   result = circle.area(r);
      +allocate heap storage;                  2[See 1]
      +copy r to message;                      3
      +load p = address of message;            4
      +load v = addr. of method 'circle.area'  5
      +goto v with return;                     6
.
.
.... storage .............
result variable (assumed pre-allocated)
immutable variable "3.142" (final)
(heap) message variable for circle method call
vtable(==>area)
stack storage
```

c) Salary



You run a company and you just decided to give all your employees a $10,000.00 raise. How would you tackle this situation programatically?

In OOP:

1. Create Employee class which initializes with name and salary, has a change salary instance method
2. Create instances of employees
3. Use the each method to change salary attribute of employees by +10,000

In FP:

1. Create employees array, which is an array of arrays with name and corresponding salary
2. Create a change_salary function which returns a copy of a single employee with the salary field updated
3. Create a change_salaries function which maps through the employee array and delegates the calculation of the new salary to change_salary
4. Use both methods to create a new dataset, named ‘happier employees’