CSC305 Lecture Note on Programming Paradigms 24/10/2022

There are several kinds of major programming paradigms:

1. **Procedural**
2. **Imperative**
3. **Declarative**
4. **Logical**
5. **Functional**
6. **Object-Oriented**

It can be shown that anything solvable using one of these paradigms can be solved using the others; however, certain types of problems lend themselves more naturally to specific paradigms.

1) Procedural Programming

Procedural programming is a derivation of imperative programming, adding to it the feature of functions (also known as "procedures" or "subroutines").
In procedural programming, the user is encouraged to subdivide the program execution into functions, as a way of improving modularity and organization.

Example of procedural programming:

```
function pourIngredients() {
   - Pour flour in a bowl
   - Pour a couple eggs in the same bowl
   - Pour some milk in the same bowl
}

function mixAndTransferToMold() {
   - Mix the ingredients
   - Pour the mix in a mold
}

function cookAndLetChill() {
   - Cook for 35 minutes
   - Let chill
}
pourIngredients()
mixAndTransferToMold()
cookAndLetChill()
```

You can see that, thanks to the implementation of functions, we could just read the three function calls at the end of the file and get a good idea of what our program does.
That simplification and abstraction is one of the benefits of procedural programming. But within the functions, we still got same old imperative code.

2) Imperative

***Introduction*:**

The *imperative* programming paradigm assumes that the computer can maintain through environments of variables any changes in a computation process. It consists of sets of detailed instructions that are given to the computer to execute in a given order. It's called "imperative" because as programmers we dictate exactly what the computer has to do, in a very specific way. It focuses on describing *how* a program operates, step by step. Computations are performed through a guided sequence of steps, in which these variables are referred to or changed. The order of the steps is crucial, because a given step will have different consequences depending on the current values of variables when the step is executed.

Example, let us assume that you want to bake a cake. Your imperative program to do this might look like this (This steps maybe different from your steps because I am not a great cook, so don't judge me with it)

1- Pour flour in a bowl
2- Pour a couple eggs in the same bowl
3- Pour some milk in the same bowl
4- Mix the ingredients
5- Pour the mix in a mold
6- Cook for 35 minutes
7- Let chill down

Another example, using an actual code example, let us filter an array of numbers in such that we can only keep any number or elements in the array that is greater than 5. Our imperative code might look like this:

```
const nums = [1,4,3,6,7,8,9,2]
const result = [ ]
for (let i = 0; i < nums.length; i++) {
   if (nums[i] > 5) result.push(nums[i])
}

console.log(result) // Output: [ 6, 7, 8, 9 ]
```

See how we are instructing the program to iterate through each element in the array, compare the item value with 5, and if the item is greater than 5, push it into an array. We are being detailed and specific in our instructions, and that's what imperative programming stands for.

- ***Imperative Languages*:**
  Popular programming languages are imperative more often than they are any other paradigm studies in this course. There are two reasons for such popularity:

1. the imperative paradigm most closely resembles the actual machine itself, so the programmer is much closer to the machine;
2. because of such closeness, the imperative paradigm was the only one efficient enough for widespread use until recently.

- *Advantages*
    - efficient;
    - close to the machine;
    - popular;
    - familiar.

- ***Disadvantages***

- The semantics of a program can be complex to understand or prove, because of ***referential transparency*** does not holds (due to side effects)
    - Side effects also make debugging harder;
    - Abstraction is more limited than with some other paradigms;
    - Order is crucial, which doesn't always suit itself to problems.

Example Referential Transparency

It is a characteristic of the functional programming paradigm or simply implies that:
Given P(x) and x=y at one point in the program, then P(x) = P(y) throughout the program.
In simple language: ***whenever x=y, a true property involving x will remain true after substituting y for all occurrences of x in the property.***
Example
If x = 2 and x + y > 5 then 2 + y > 5 if the referential transparency property holds.

Side effects

However, side effects (changes in the state of a machine), which are common in imperative programming languages, violate referential transparency.
For instance, the following "**function**" violates referential transparency by changing the value of the global variable y:

```
function f(x:integer):integer;
        begin
                y := y + 1;
                f := y + x
        end
```

Assume that referential transparency holds. Therefore, ***if z1 = z2 then f(z1) = f(z2).*** Suppose that ***z1=z2=y=0***. Then the expression ***f(z1) == f(z2)*** should return ***true***. But whichever side of the *"=="* is evaluated first will increment ***y***, which will result in the other side being evaluated differently. Therefore, the referential transparency cannot hold.

3) Declarative Programming

Declarative programming is all about hiding away complexity and bringing programming languages closer to human language and thinking. It's the direct opposite of imperative programming in the sense that the programmer doesn't give instructions about *how* the computer should execute the task, but rather on *what* result is needed.
It is a type of programming paradigm that describes what programs to be executed. Developers are more concerned with the answer that is received. It declares what kind of results we want and leave programming language aside focusing on simply figuring out how to produce them. In simple words, it mainly focuses on end result. It expresses the logic of computation.
This will be much clearer with an example. Following the same array filtering example, a declarative approach might be:

const nums = [1,4,3,6,7,8,9,2]

console.log(nums.filter(num => num > 5)) // Output: [6, 7, 8, 9]

See that with the filter function, we are not explicitly telling the computer to iterate over the array or store the values in a separate array. We just say what we want ("filter") and the condition to be met ("num > 5").

What is good about this is that it's easier to read and comprehend, and often shorter to write. JavaScript's filter, map, reduce and sort functions are good examples of declarative code.
Another good example is modern JS frameworks/libraries like React. Take this code for **example:** ***<button onClick={() => console.log('You clicked me!')}>Click me</button>***

Here we have a button element, with an event listener that fires a console.log function when the button is clicked.
JSX syntax (what React uses) mixes HTML and JS in the same thing, which makes it easier and faster to write apps. But that's not what browsers read and execute. React code is later on transpired into regular HTML and JS, and that's what browsers run in reality.
JSX is declarative, in the sense that its purpose is to give developers a friendlier and more efficient interface to work with.
An important thing to notice about declarative programming is that under the hood, the computer processes this information as imperative code anyway.
Following the array example, the computer still iterates over the array like in a for loop, but as programmers we don't need to code that directly. What declarative programming does is to **hide away** that complexity from the direct view of the programmer. After getting the basics of understanding of both the languages now let u do discuss the key differences between these two different types of programming

Imperative Programming	**Declarative Programming**
In this, programs specify how it is to be done.	In this, programs specify what is to be done.
It simply describes the control flow of computation.	It simply expresses the logic of computation.
Its main goal is to describe how to get it or accomplish it.	Its main goal is to describe the desired result without direct dictation on how to get it.
Its advantages include ease to learn and read, the notional model is simple to understand, etc.	Its advantages include effective code, which can be applied by using ways, easy extension, high level of abstraction, etc.
Its type includes procedural programming, object-oriented programming, parallel processing approach.	Its type includes logic programming and functional programming.
In this, the user is allowed to make decisions and commands to the compiler.	In this, a compiler is allowed to make decisions.

Imperative Programming	Declarative Programming
It has many side effects and includes mutable variables as compared to declarative programming.	It has no side effects and does not include any mutable variables as compared to imperative programming.
It gives full control to developers that are very important in low-level programming.	It may automate repetitive flow along with simplifying code s

Declarative

Pros	Cons
Better readability and understanding of the code	More lines of code, where a potential bug could hide
Better control over the actual execution of the changes to the world	Potential loss of performance, due to more memory allocation and intermediate function calls
	Longer debugging, due to bigger stack traces
	Developers are usually less comfortable with this way of programming

Imperative

Pros	Cons
Less code overall, as there is no need to wrap imperative code inside declarative functions	More time taken to read and understand what the code does
Shorter debugging, due to smaller stack traces	But harder debugging overall, due to state mutations and "less-controlled" changes to the world
Developers are usually more comfortable with this way of programming	

4) Logical

○ ***Introduction:***

The *Logical Paradigm* takes a declarative approach to problem-solving. Various logical assertions about a situation are made, establishing all known facts. Then queries are made. The role of the computer becomes maintaining data and logical deduction.
Logical Paradigm Programming*:* A logical program is divided into three sections:

1. a series of definitions/declarations that define the problem domain
2. statements of relevant facts
3. statement of goals in the form of a query

Any assumed solution to a query is returned. The definitions and declarations are constructed entirely from relations. i.e. ***X is a member of Y or X is in the internal between a and b etc.***

○ ***Advantages****:*

The advantages of logic oriented programming are:

1. The system solves the problem, so the programming steps themselves are kept to a minimum;
2. Proving the validity of a given program is simple.

Sample Code of Logical Paradigm.

```
domains
        being = symbol
predicates
animal(being) % all animals are beings
dog(being) % all dogs are beings
die(being) % all beings die
clauses
animal(X) :- dog(X) % all dogs are animals
dog(Jack). % Jack is a dog
die(X) :- animal(X) % all animals die
```

5) Functional

Introduction: In functional programming, functions are treated as **first-class citizens**, meaning that they can be assigned to variables, passed as arguments, and returned from other functions. This paradigm views all subprograms as functions in the mathematical sense-informally, they take in arguments and return a single solution. The solution returned is based entirely on the input, and the time at which a function is called has no relevance. The computational model is therefore one of function application and reduction.
Another key concept is the idea of **pure functions**. A **pure** function is one that relies only on its inputs to generate its result. And given the same input, it will always produce the same result. Besides, it produces no side effects (any change outside the function's environment).
With these concepts in mind, functional programming encourages programs written mostly with functions. It also defends the idea that code modularity and the absence of side effects makes it easier to identify and separate responsibilities within the codebase. This therefore improves the ***code maintainability***.

- ***Advantages***

  The following are desirable properties of a functional language:
  1. The high level of abstraction, especially when functions are used, suppresses many of the details of programming and thus removes the possibility of committing many classes of errors;
  2. The lack of dependence on assignment operations, allowing programs to be evaluated in many different orders. This evaluation order independence makes function-oriented languages good candidates for programming massively parallel computers;
  3. The absence of assignment operations makes the function-oriented programs much more amenable to mathematical proof and analysis than are imperative programs, because functional programs possess referential transparency.
- ***Disadvantages***

1. Perhaps less efficiency
2. Problems involving many variables or a lot of sequential activity are sometimes easier to handle imperatively or with object-oriented programming.

Sample Code of Functional Paradigm.

Function for computing the average of two numbers:

```
(defun avg(X Y) (/ (+ X Y) 2.0))
```

Function is called by:

```
> (avg 10.0 20.0)
```

Function returns:

15.0

Another Example:

```
const nums = [1,4,3,6,7,8,9,2]
const result = [] // External variable
for (let i = 0; i < nums.length; i++) {
   if (nums[i] > 5) result.push(nums[i])
}
console.log(result) // Output: [ 6, 7, 8, 9 ]
```

To transform this into functional programming, we could do it like this:

```
const nums = [1,4,3,6,7,8,9,2]
function filterNums() {
   const result = [] // Internal variable
   for (let i = 0; i < nums.length; i++) {
      if (nums[i] > 5) result.push(nums[i])
   }

   return result
}

console.log(filterNums()) // Output: [ 6, 7, 8, 9 ]
```

6) Object-Oriented

○ *Introduction*

Object Oriented Programming (OOP) is a paradigm in which real-world objects are each viewed as separate entities having their own state which is modified only by built in procedures, called ***methods***. Because objects operate independently, they are ***encapsulated*** into modules which contain both local environments and methods. Communication with an object is done by message passing.
Objects are organized into ***classes***, from which they ***inherit*** methods and equivalent variables. The object-oriented paradigm provides key benefits of ***reusable code and code extensibility***.

Features & Benefits
A new class (called a *derived class* or *subclass*) may be *derived* from another class (called a *base class* or *superclass*) by a mechanism called inheritance. The derived class *inherits* all the features of the base class: its structure and behavior (response to messages). In addition, the derived class may contain *additional* state (instance variables), and may exhibit additional behavior (new methods to respond to new messages). Significantly, the derived class can also *override* behavior corresponding to some of the methods of the base class: there would be a different *method* to respond to the same message. Also, the inheritance mechanism is allowed even **without access to the source code of the base class.**

The ability to use inheritance is the single most distinguishing feature of the OOP paradigm. Inheritance gives OOP its chief benefit over other programming paradigms - relatively easy *code reuse* and *extension* without the need to change existing source code. The mechanism of modeling a program as a collection of objects of various classes, and furthermore describing many classes as extensions or modifications of other classes, provides a high degree of modularity.
Ideally, the state of an object is manipulated and accessed only by that object's methods. (Most O-O languages allow direct manipulation of the state, but such access is stylistically discouraged). In this way, a class' *interface* (how objects of that class are accessed) is separate from the class' *implementation* (the actual code of the class' methods). Thus *encapsulation* and *information hiding* are inherent benefits of OOP.

<u>Sample</u> Code of Object-Oriented Programming Paradigm.

Class:

```
class data
{
public:
Date(int mn, int dy, int yr); //Constructor
void display(); //Function to print data
~Date(); //Destructor
Private:
int month, day, year; //Private data members
};
```

Inheritance:

```
class Employee
{
public:
Employee();
const char *getName() const;
private:
char name[30];
};
class WageEmployee : public Employee
{
public:
WageEmployee(const char *nm);
void setWage(double wg);
void setHours(double hrs);
private:
double:wage;
double:hours;
};
WageEmployee aWorker("Bill Gates");
const char *str;

aWorker.setHours(40.0); //call WageEmployee::setHours
str = aWorker.getName(); //call Employee::getname
```

Another Example:
Following our pseudo-code cooking example, now let's say in our bakery we have a main cook (called Miss Caleb) and an assistant cook (called Mr. Daniel Osaze) and each of them will have certain responsibilities in the baking process. If we used OOP, our program might look like this.

```
// Create the two classes corresponding to each entity
class Cook {
        constructor constructor (name) {
    this.name = name
  }

  mixAndBake() {
    - Mix the ingredients
        - Pour the mix in a mold
    - Cook for 35 minutes
  }
}

class AssistantCook {
  constructor (name) {
        this.name = name
    }

    pourIngredients() {
        - Pour flour in a bowl
        - Pour a couple eggs in the same bowl
        - Pour some milk in the same bowl
    }

    chillTheCake() {
            - Let chill
    }
}

// Instantiate an object from each class
const Frank = new Cook('Frank')
const Anthony = new AssistantCook('Anthony')

// Call the corresponding methods from each instance
Anthony.pourIngredients()
Frank.mixAndBake()
Anthony.chillTheCake()
```