Slecture-03

Abstraction And Context Free Grammar

Group Name: Infinite Loop

Members:

1)H.m Ishtiaq Salehin-1610510042

2)Quazi Rubayet Anjum Joy-1611016042

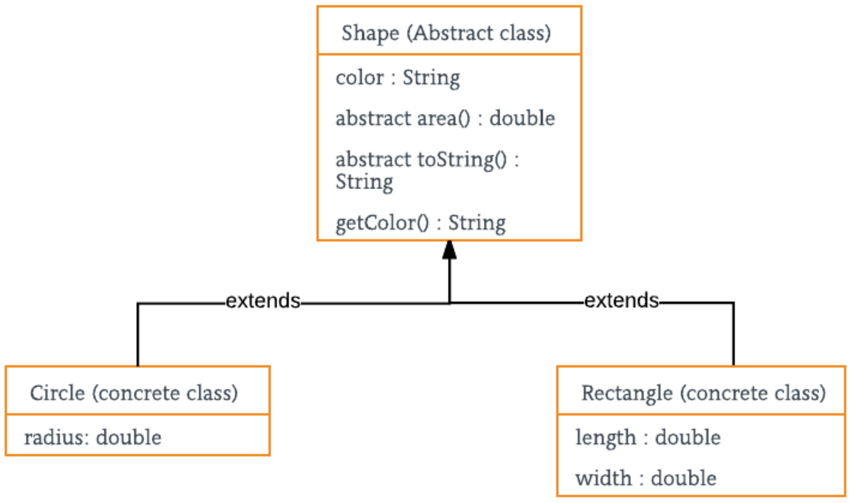
3)Irin Sultana-1610343042

Abstraction

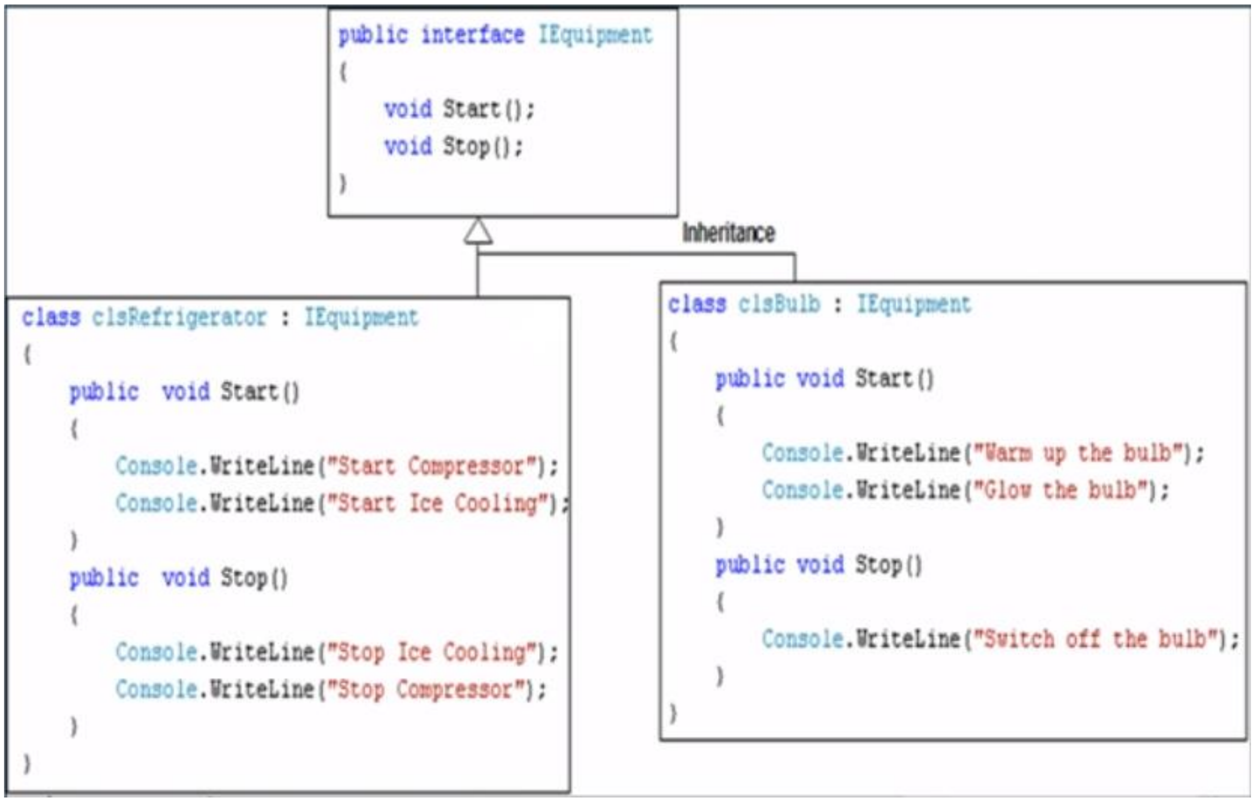
Abstraction is the process of making something easier to understand by ignoring some of the details that may be unimportant. What this actually means is that details are hidden and only important stuffs are shown. Abstraction gives the programmer a lot of flexibility and convenience. “To abstract” means simply to hide something. Programming languages are abstractions of the physical machine.

Abstraction allows us to focus on what is relatively important for the current purpose. An example of abstraction can be a simple storage box. We use boxes in our everyday life. We label them on basis of what they contain. The box itself is the concept of abstraction. The box gives us the facility of storage. We can store whatever we need in the box. The purpose of the box is to store whatever we need to store. The box is not concerned about its contents. We can change the contents of the box however and whenever we want.

Another example is given below:



This is an example of abstraction in java language. Here we see the shape class which gives a bare minimum details we are supposed to get when we imagine any shape. The shape class is extended by the circle class and the rectangle class. Both have the same attributes of the shape class and also have some attributes that are particularly suitable for them. When circle shaped objects are needed to be used the radius of the circle has to be given. Again when rectangle shaped objects are needed to be used the length and width must be given. The important fact is that radius, length, width are not associated with shape super class. These attributes are only relevant when those exact shapes (circle, rectangle) are considered. Thus the shape class in the given picture is the concept of abstraction.



```
public interface IEquipment
{
    void Start();
    void Stop();
}
```

```
class clsRefrigerator : IEquipment
{
    public  void Start()
    {
        Console.WriteLine("Start Compressor");
        Console.WriteLine("Start Ice Cooling");
    }
    public  void Stop()
    {
        Console.WriteLine("Stop Ice Cooling");
        Console.WriteLine("Stop Compressor");
    }
}
```

```
class clsBulb : IEquipment
{
    public void Start()
    {
        Console.WriteLine("Warm up the bulb");
        Console.WriteLine("Glow the bulb");
    }
    public void Stop()
    {
        Console.WriteLine("Switch off the bulb");
    }
}
```

In the given code we see the interface IEquipment has two methods start and stop but their exact functions are not given but in clsRefrigerator, the start method and stop method are distinctly defined for a refrigerator. Also in clsBulb we see that start and stop function is clearly defined. Point to be noted is that though the name of the methods are same but their functions are different for clsRefrigerator and clsBulb. Here IEquipment is an abstract class which just gives the concept of start and stop. And in clsRefrigerator and clsBulb, the job of start and stop is unambiguously defined. So we can see that the details are hidden in IEquipment and we change the methods as we need for our purpose. This is why abstraction is very important and widely used by programmers.

There are certain advantages to abstraction. Such as the programmer does not have to write the low-level code. The programmer does not have to specify all the register/binary-level steps or care about the hardware or instruction set details. Code duplication is avoided and thus programmer does not have to repeat fairly common tasks every time a similar operation is to be performed. It allows internal implementation details to be changed without affecting the users of the abstraction.

Types of Abstraction

There are mainly two types of abstraction.

- Data abstraction
- Control abstraction

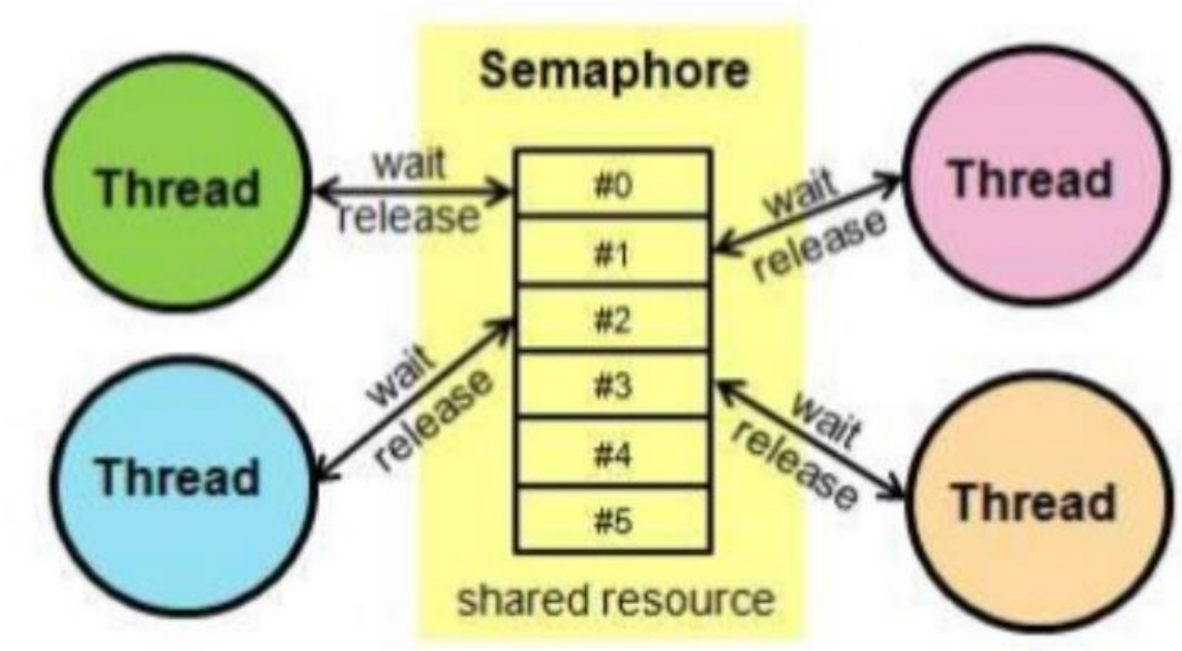
Data Abstraction

Data abstraction means hiding the details about the data. The concept of representing important details and hiding away the implementation details is called data abstraction. This programming technique separates the interface and implementation. Data abstraction allows the definition and use of data types without referring to how such types are implanted.

An example of data abstraction is the use of the data type double, which is an abstraction of real numbers. The computer hardware limits the range of real numbers that can be represented. Different computers use variety of representation schemes for type double. We can use the data type double and its associated operators (+,-,*, /,= , = =, <, and so on) without being concerned with the details of its implementation.

Control Abstraction

Control abstraction provides the programmer the ability to hide procedural data. It’s an abstraction of behavior. It provides an easier, higher level API to hide client from unnecessary execution details. In control abstraction, a module is specified by the function it performs. For example, a module to compute log of a value can be abstractly represented by function log. It is the basis of partitioning in function oriented approach. Another example of a control abstraction is the synchronization on semaphore used to coordinate activities in an operating system.



When we write a program in a high level language the code is compiled and turned into an intermediate level code which later gets transformed into target machine code and then the program does as it is instructed. These process are hidden from us by the machine, they just occur in back-end. This is also an example of control abstraction.

Context Free Grammar (CFG)

The word context means topic. So context free grammar refers to anything that is grammatically correct but the context is not important. A context free grammar is a grammar which satisfies certain properties. In computer science, grammars describe languages; specifically, they describe formal languages.

A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple (N, T, P, S) where

- N is a set of non-terminal symbols.
- T is a set of terminals
- P is a set of rules
- S is the start variable.

Terminal and nonterminal symbols are the lexical elements used in specifying the production rules constituting a formal grammar. Terminal symbols are the elementary symbols of the language defined by a formal grammar. Nonterminal symbols or syntactic variables are replaced by groups of terminal symbols according to the production rules.

For example

```
<digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<integer> ::= ['-'] <digit> {<digit>}
```

The symbols (-,0,1,2,3,4,5,6,7,8,9) are terminal symbols and <digit> and <integer> are nonterminal symbols.

A context-free grammar is simply a grammar where the thing that you're replacing (left of the arrow) is a single "non-terminal" symbol. A non-terminal symbol is any symbol you use in the grammar that can't appear in your final strings. For example:

S -> BBB

B -> 0

B -> 1

Here S is a non-terminal as it can't appear in the final string but the values of B are terminal because it has an exact value which can be used to replace B.

Example 1:

Grammar:

$S \rightarrow aSa$ (Rule: 1)

$S \rightarrow bSb$ (Rule: 2)

$S \rightarrow \varepsilon$ (Rule: 3) ['ε' means nothing or empty]

(i) We want to get 'aa' as the end product using the above given grammar

$S \Rightarrow aSa$, (Rule: 1)

$\Rightarrow a\varepsilon a$, (Rule: 3)

$\Rightarrow aa$

(ii) We want to get 'bb' as the end product using the above given grammar

$S \Rightarrow bSb$, (Rule: 2)

$\Rightarrow b\varepsilon b$, (Rule: 3)

$\Rightarrow bb$

(iii) We want to get ‘abba’ as the end product using the above given grammar

$S \Rightarrow aSa,$ (Rule: 1)

$\Rightarrow abSba,$ (Rule: 2)

$\Rightarrow ab\varepsilon ba$ (Rule: 3)

$\Rightarrow abba$

(iv) We want to get ‘baab’ as the end product using the above given grammar

$S \Rightarrow bSb,$ (Rule: 2)

$\Rightarrow baSab,$ (Rule: 1)

$\Rightarrow ba\ \varepsilon\ ab$ (Rule: 3)

$\Rightarrow baab$

(v) We want to get ‘aabbbbaa’ as the end product using the above given grammar

$S \Rightarrow aSa,$ (Rule: 1)

$\Rightarrow aaSaa,$ (Rule: 1)

$\Rightarrow aabSbaa,$ (Rule: 2)

$\Rightarrow aabbSbbaa,$ (Rule: 2)

$\Rightarrow aabb\ \varepsilon\ bbaa$ (Rule: 3)

$\Rightarrow aabbbbaa$

Example 2:

Grammar:

$S \rightarrow abB$, (Rule: 1)

$A \rightarrow aaBb$ (Rule: 2)

$A \rightarrow \varepsilon$ (Rule: 3) [‘ε’ means nothing or empty]

$B \rightarrow bbAa$ (Rule: 4)

(i) We want to get ‘ab bba’ as the end product using the above given grammar

$S \Rightarrow abB$, (Rule: 1)

$\Rightarrow ab\ bbAa$ (Rule: 4)

$\Rightarrow ab\ bb\ \varepsilon\ a$ (Rule: 3)

$\Rightarrow ab\ bba$

(ii) We want to get ‘ab bb aa bba ba’ as the end product using the above given grammar

$S \Rightarrow abB$, (Rule: 1)

$\Rightarrow ab\ bbAa$ (Rule: 4)

$\Rightarrow ab\ bb\ aaBb\ a$ (Rule: 2)

$\Rightarrow ab\ bb\ aa\ bbAa\ ba$ (Rule: 4)

$\Rightarrow ab\ bb\ aa\ bb\ \varepsilon\ a\ ba$ (Rule: 3)

$\Rightarrow ab\ bb\ aa\ bba\ ba$

(iii) We want to get 'ab bb aa bb aa bba ba ba' as the end product using the above given grammar

S ⇒ abB, (Rule: 1)

⇒ ab bbAa (Rule: 4)

⇒ ab bb aaBb a (Rule: 2)

⇒ ab bb aa bbAa ba (Rule: 4)

⇒ ab bb aa bb aaBb a ba (Rule: 2)

⇒ ab bb aa bb aa bbAa b a ba (Rule: 4)

⇒ ab bb aa bb aa bb ε a b a ba (Rule: 3)

⇒ ab bb aa bb aa bba ba ba

(iv) We want to get 'ab bb aa bb aa bb aa bba ba ba ba' as the end product using the above given grammar

S ⇒ abB, (Rule: 1)

⇒ ab bbAa (Rule: 4)

⇒ ab bb aaBb a (Rule: 2)

⇒ ab bb aa bbAa ba (Rule: 4)

⇒ ab bb aa bb aaBb a ba (Rule: 2)

⇒ ab bb aa bb aa bbAa b a ba (Rule: 4)

⇒ ab bb aa bb aa bb aaBb a ba ba (Rule: 2)

⇒ ab bb aa bb aa bb aa bbAa ba ba ba (Rule: 4)

⇒ ab bb aa bb aa bb aa bb ε a ba ba ba (Rule: 3)

⇒ ab bb aa bb aa bb aa bba ba ba ba

Example 3:

$S \rightarrow aSb$, (Rule: 1)

$S \rightarrow ab$ (Rule: 2)

(i) We want to get 'ab' as the end product using the above given grammar

$S \Rightarrow ab$, (Rule: 2)

(ii) We want to get 'a^2b^2' as the end product using the above given grammar

$S \Rightarrow aSb$, (Rule: 1)

$\Rightarrow aabb$, (Rule: 2)

$\Rightarrow aabb$

$\Rightarrow a^2b^2$

(iii) We want to get 'a^3b^3' as the end product using the above given grammar

$S \Rightarrow aSb$, (Rule: 1)

$\Rightarrow aaSbb$, (Rule: 1)

$\Rightarrow aaabbb$, (Rule: 2)

$\Rightarrow aaabbb$

$\Rightarrow a^3b^3$

(iv) We want to get ‘a^4b^4’ as the end product using the above given grammar

S ⇒ aSb, (Rule: 1)

⇒ aaSbb, (Rule: 1)

⇒ aaaSbbb, (Rule: 1)

⇒ aaaabbbb, (Rule: 2)

⇒ aaaabbbb

⇒ a^4b^4

(v)) We want to get ‘a^5b^5’ as the end product using the above given grammar

S ⇒ aSb, (Rule: 1)

⇒ aaSbb, (Rule: 1)

⇒ aaaSbbb, (Rule: 1)

⇒ aaaaSbbbb, (Rule: 1)

⇒ aaaaabbbbb, (Rule: 2)

⇒ aaabbb

⇒ a^5b^5