



# CONCEPTS OF PROGRAMMING LANGUAGE

**GROUP NAME: Electron**

## **Slecture 03**

Course Code: CSE425 Semester: Summer 2019

Faculty Initial: MSK1

Section:03

| <u>Name</u>      | <u>ID</u>  |
|------------------|------------|
| Shahriar Rahman  | 1530730643 |
| Md Yeahea Shibly | 1610258042 |
| Sourov Sarker    | 1610427042 |

## ABSTRACTION

### **GENERAL DEFINITION:**

In software engineering and computer science, abstraction is the process of taking away or removing characteristics from something in order to reduce it to a set of essential characteristics. Simply, it is a technique for arranging complexity of computer systems. It is one of the key concepts of object-oriented programming (OOP) languages and the concept of abstraction has itself become a declarative statement in C++, Object Pascal, or Java, using the keywords `virtual` (in C++) or `abstract` and `interface` (in Java). After such a declaration, it is the responsibility of the programmer to implement a class to instantiate the object of the declaration.

### **EXAMPLE 1: OWNER AND CAR**

A general explanation of how an abstract works and what is that imagining a car. For instance, most people do not know mechanically how a car actually works (what is happening underneath the hood when we drive or brake), however, many knows how to drive a car and how to interact with its interface (wheels, gear, accelerator, etc.), implying that he/she abstracted a way of what a car actually is to only needing to understand its basic interface. Hence, they do not have to know the details of what is really happening underneath the hood, but only need to know the basic set of instructions for it.

This is the most basic Example of abstraction.

User owns a Car.

The car runs properly with its necessary parts.

An engine is necessary to run the car.

User → Car;

Car → Necessary Parts;

Engine → Car;

### **pseudocode:**

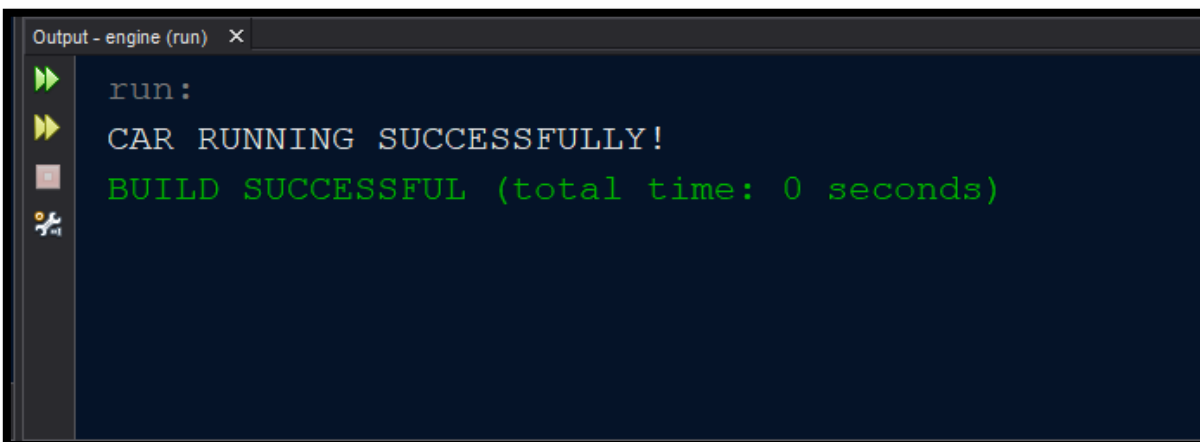
```
Abstract class User{
    Abstract Car method declaration;
}
Class Engine extends User {
    Car method definition;
    Main Engine Method{
        Create Object;
        Call Car Method;}}
```

## CODE DEMONSTRATION (Using Java):

```
package engine;
//@author SHAHRIAR RAHMAN

abstract class User{
    abstract void car();
}
class engine extends User{
    void car(){
        System.out.println("CAR RUNNING SUCCESSFULLY!");
        //This method contains all the necessary parts required to run the car.
    }
    public static void main(String[] args) {
        User obj = new engine();
        obj.car();
    }
}
```

## DEMONSTRATION OUTPUT:



```
Output - engine (run) x
run:
CAR RUNNING SUCCESSFULLY!
BUILD SUCCESSFUL (total time: 0 seconds)
```

## EXAMPLE 2: NSU DATABASE DATA ACCESS

For instance, a User or a student of North South University is completing an application form from which his/her new information will be stored in the NSU database. The form has information boxes that must be completely filled (First and Last Name, ID, Phone, Hobby, Expeditions, etc.). When required, it is possible to fetch all the relevant information regarding the student's credentials. However, there are some redundant information that is not required to fetch even though it is already stored in the database. So, the system will automatically ignore information such as Hobby and Expeditions while fetching necessary data.

This Example consists of multiple abstract methods.

Student → First Name, Last Name, ID, Phone, Hobby, Expeditions;

NSU DATABASE extends Student:

First Name → x;                      Last Name → y;                      ID → 153xxx;  
Phone → 017xxx;                      Hobby → xyz;                      Expeditions → abc;

NSU SYSTEM → First Name, Last Name, ID, Phone;

### Pseudocode:

Abstract class User{

    All Abstract Method Declarations;

    FirstName Method; LastName Method; ID Method; Phone Method; Hobby Method;

    Expeditions Method;

}

Class NSU{

    Method definition;

    Main Engine Method{

        Create Object;

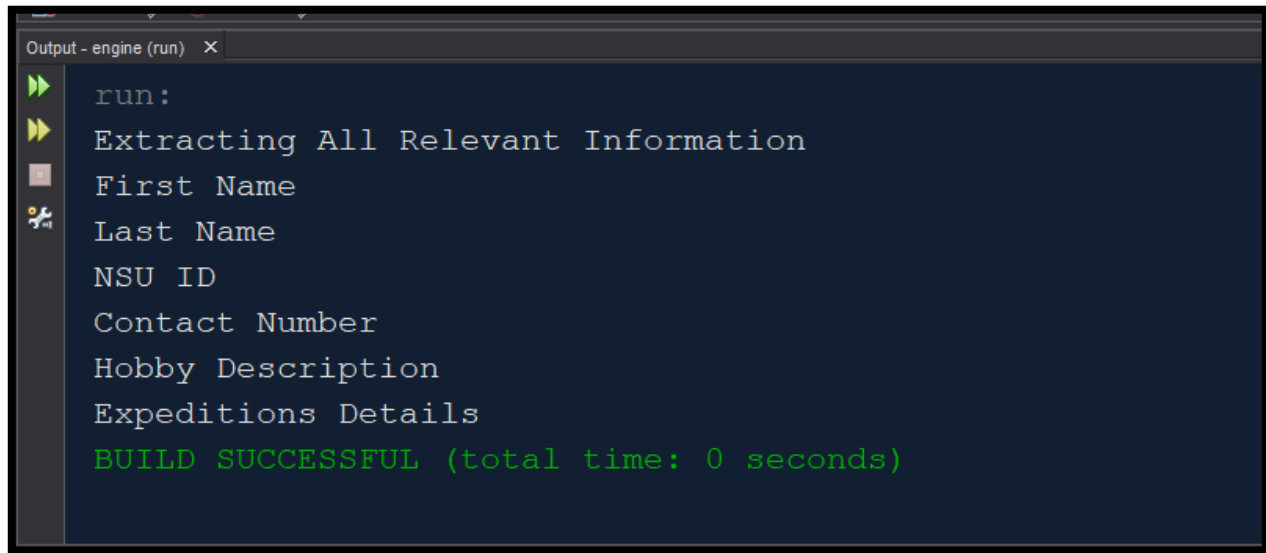
        Call All Relevant Methods;

    }}

### CODE DEMONSTRATION (Using Java):

```
1  package NSU;
2  //@author SHAHRIAR RAHMAN
3
4  abstract class User{
5      abstract void FirstName();
6      abstract void LastName();
7      abstract void NSU_ID();
8      abstract void ContactNumber();
9      abstract void Hobby();
10     abstract void Expeditions(); }
11
12     class NSU extends User{
13         void FirstName() {
14             System.out.println("First Name"); }
15         void LastName() {
16             System.out.println("Last Name"); }
17         void NSU_ID() {
18             System.out.println("NSU ID"); }
19         void ContactNumber() {
20             System.out.println("Contact Number"); }
21         void Hobby() {
22             System.out.println("Hobby Description"); }
23         void Expeditions() {
24             System.out.println("Expeditions Details"); }
25
26         public static void main(String[] args) {
27             User obj = new NSU();
28             System.out.println("Extracting All Relevant Information");
29             obj.FirstName(); obj.LastName(); obj.NSU_ID();
30             obj.ContactNumber(); obj.Hobby(); obj.Expeditions(); }
31     }
```

## DEMONSTRATION OUTPUT:



```
Output - engine (run) x
run:
Extracting All Relevant Information
First Name
Last Name
NSU ID
Contact Number
Hobby Description
Expeditions Details
BUILD SUCCESSFUL (total time: 0 seconds)
```

### EXAMPLE 3: MATHEMATICAL SHAPES

This example is related to basic mathematics. A Shape is an abstract class, and its implementation is provided by the Rectangle, Circle and Triangle classes. Mostly, it is unknown about the implementation class (which is hidden to the end user), and an object of the implementation class is provided by the Factory Method.

A Factory Method is a method that returns the instance of the class.

In this example, if an instance of Rectangle class is created, draw() method of Rectangle class will be invoked.

Shape → draw

Rectangle → draw → output “drawing a Rectangle”;

Circle → draw → output “drawing a Circle”;

Triangle → draw → output “drawing a Triangle”;

Example (get Circle & Triangle only) → draw Circle and draw Triangle;

#### Pseudocode:

```
Abstract Class Shape {
    Abstract draw Method declared;}
class Circle extends Shape {
    draw Method defined within; }
class rectangle extends Shape{
    draw Method defined within; }
```

```
class triangle extends Shape {
    draw Method defined within; }
Class Exp3 {
    Create Only Relevant Objects;
    Relevant Method Calls;
}
```

### CODE DEMONSTRATION (Using Java):

```
1 package Example3;
2 //@author SHAHRIAR RAHMAN
3 abstract class Shape{
4     abstract void draw();}
5
6 class Rectangle extends Shape{
7     @Override
8     void draw(){
9         System.out.println("drawing a rectangle");} }
10 class Circle extends Shape{
11     @Override
12     void draw(){System.out.println("drawing a circle");} }
13 class Triangle extends Shape{
14     @Override
15     void draw(){System.out.println("drawing a triangle");} }
16 class Example3{
17     public static void main(String args[]){
18         Shape s1=new Circle();//getShape() method 1
19         Shape s2=new Triangle();//getShape() method 1
20         s1.draw(); s2.draw(); }
21 }
```

### DEMONSTRATION OUTPUT:

```
>> run:
>> drawing a circle
drawing a triangle
BUILD SUCCESSFUL (total time: 0 seconds)
```

## EXAMPLE 4: BANKING

This example is based on Rate of Interest.

Bank → interest

BANK\_ASIA → Bank → set interest rate;

JANATA\_BANK → Bank → set interest rate;

TestBank → BANK\_ASIA interest rate and JANATA\_BANK interest rate

### Pseudocode:

Abstract class Bank{

Abstract GetInterest method;

}

Class BANK\_ASIA extends Bank {

Return GetInterest with a Value;

}

Class JANATA\_BANK extends Bank {

Return GetInterest with a Value;

}

Class TestBank{

Object Creation;

Print interest Rate for each Bank;

}

### CODE DEMONSTRATION (Using Java):

```
1 package TestBank;
2 // @author SHAHRIAR RAHMAN
3
4 abstract class Bank{
5     abstract float getRateOfInterest();
6     // This will return rate of interest;
7 }
8 class BANK_ASIA extends Bank{
9     float getRateOfInterest(){
10         return 7;
11     }
12 }
13 class JANATA extends Bank{
14     float getRateOfInterest(){
15         return 8;
16     }
17 }
18 class TestBank{
19     public static void main(String args[]){
20         Bank b;
21         b=new BANK_ASIA();
22         System.out.println("Rate of Interest for BANK ASIA is: "+b.getRateOfInterest()+" %");
23         b=new JANATA();
24         System.out.println("Rate of Interest for JANATA BANK is: "+b.getRateOfInterest()+" %");
25     }
26 }
```

## DEMONSTRATION OUTPUT:

```
run:
Rate of Interest for BANK ASIA is: 7.0 %
Rate of Interest for JANATA BANK is: 8.0 %
BUILD SUCCESSFUL (total time: 0 seconds)
```

## EXAMPLE 5: BIKE

In this example, an abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method.

Bike → Bike Constructor → Run → Change Gear

HONDA → Bike → Run

Example → Run AND Change Gear

### Pseudocode:

```
Abstract Class Bike{
    Constructor();
    Abstract Run; Abstract method Declaration
    ChangeGear Method;
}
Class Honda extends Bike{
    Run Method Define;
}
Class Example5{
    Create Object;
    Run;
    Change Gear;
}
```



### CODE DEMONSTRATION (Using Java):

```
1 package Example5;
2 //@author SHAHRIAR RAHMAN
3
4 abstract class Bike{
5     Bike(){
6         System.out.println("bike is started");
7     }
8     abstract void run();
9     void changeGear(){
10        System.out.println("gear changed");
11    }
12 }
13
14 class Honda extends Bike{
15     @Override
16     void run(){
17         System.out.println("engine ready..");
18     }
19 }
20
21 class Example5{
22     public static void main(String args[]){
23         Bike obj = new Honda();
24         obj.run();
25         obj.changeGear();
26     }
27 }
```

### DEMONSTRATION OUTPUT:

```
Output - engine (run) x
run:
bike is started
engine ready..
gear changed
BUILD SUCCESSFUL (total time: 0 seconds)
```

## **TYPES OF ABSTRACTION**

Typically abstraction can be seen in two ways:

- Data abstraction
- Control abstraction

### **DATA ABSTRACTION**

Data abstraction is the way to create complex data types and exposing only meaningful operations to interact with the data type, whereas hiding all the implementation details from outside works. The benefit of this approach involves capability of improving the implementation over time e.g. solving performance issues is any. The idea is that such changes are not supposed to have any impact on client code since they involve no difference in the abstract behavior. Data abstraction can be defined in two ways.

Abstraction using Classes: In C++, abstraction implementation is possible by creating classes. Class helps the user to group data members and member functions using available access specifiers. A Class can make a decision on which data member will be able to be seen to outside world and which is not.

Abstraction in Header files: For example In C++, considering the `sqrt()` method present in `math.h` header file. Whenever a user need to calculate square root of a number, he/she can simply call the function `sqrt ()` present in the `math.h` header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating power of numbers.

### **CONTROL ABSTRACTION**

A software is essentially a collection of numerous statements written in any programming language. Most of the times, statement are similar and repeated over places multiple times. Control abstraction is the process of identifying all such statements and expose them as a unit of work. This feature normally used when creating a function to perform any work. In most programming languages, a principal mechanism for control abstraction is known as Subroutines. A subroutine performs its operation on behalf of a caller, who waits for the subroutine to complete before continuing execution. Most subroutines are parameterized: the caller passes arguments that impacts the subroutine's operations, or provide it with data on which to operate. A function is when a subroutine returns any type of value, but when it does not return any value then it is known as a procedure. Subroutine is required to be declared in most languages before they are used. However, some do not such as in Fortran, C, and LISP).

## **ADVANTAGES OF ABSTRACTION**

- It helps the user to avoid writing the low level code.
- It increases code efficiency. Avoids code duplication and increases reusability.
- Can change internal implementation of class independently without affecting the user.
- Helps to increase security of an application or program as only important details are provided to the user.
- It helps having more self-contained modules.
- From a programmer's perspective, abstraction makes the application extendable in much easier way. It makes refactoring much easier.
- Improves Readability. If things are properly abstracted, the reader doesn't need to retain everything in his/her head at once.
- It hides implementation logic from another class. The other class does not need to know about the logic. It simply uses the class without knowing about implementation. Even any update to the logic doesn't require modification of another class.

## **DISADVANTAGES OF ABSTRACTION**

- Execution speed: In order to execute an abstraction, the code implementing needs to handle cases and situations which may not always be needed - or often are not needed - by many usage scenarios. This will typically make code using the abstractions slower than it would be if the code directly implemented the operation without using the abstraction.
- Code size. This rarely matters on larger systems nowadays, but it can be a significant problem in small devices or constrained environments. The extra code needed to fully implement an abstraction adds line counts and ultimately the code size, and if the code is not carefully implemented, it can end up having lots of extra code, leading to overlarge runtime executables and ultimately making the device itself more expensive.
- Problem while dealing with database schemas. Relational databases are extremely unforgiving of badly-executed abstractions and abstraction layers.

**Context Free Grammar:** A set of rules/ productions used to generate patterns of strings. It helps us to describe any language.

Context Free Grammar (CGF) has 4 tuples.

- $V \Rightarrow$  Variable/non-terminal
- $T \Rightarrow$  Set of terminal
- $P \Rightarrow$  Set of productions/rules
- $S \Rightarrow$  Start Variable

$S \Rightarrow 01 \mid 0S1$

Or , we can say it as

$S \Rightarrow 01$   
 $S \Rightarrow 0S1$

Here " $\Rightarrow$ " is Derivation sign

Variable,  $V = \{S\}$

- ❖ Left hand side's(LHS) capital letters are variable

Terminal,  $T = \{01\}$

- ❖ Except Variable Right hand side's(RHS) all contents are Terminals

Production,  $P = \{01\} \mid \{0S1\}$

Start Point,  $S = S$

- ❖ Start variable S is a Variable

Example:  $E \Rightarrow E+T \mid T$   
 $T \Rightarrow T*F \mid F$   
 $F \Rightarrow (E) \mid id$

Here Variables,  $V = \{E, T, F\}$

Terminal,  $T = \{+, *, (, ), id\}$

Start Point,  $S = E$

Example:  $S \Rightarrow (L) \mid a$   
 $L \Rightarrow L, S \mid S$

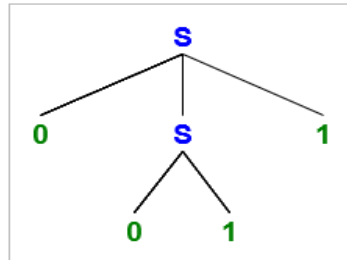
Here Variables,  $V = \{S, L\}$

Terminal,  $T = \{ (, ), a, , \}$   $\rightarrow$  Comma is a Terminal .

Start Point,  $S = S$

Context-free grammars can be modeled as **parse trees**. The nodes of the tree represent the symbols and the edges represent the use of production rules. The leaves of the tree are the end result (terminal symbols) that make up the string the grammar is generating with that particular sequence of symbols and production rules.

Parse tree for making 0011 using the grammar:  $S \rightarrow 01 \mid 0S1$



**Example:**

$S \rightarrow 01 \mid 0S1$

We want to make 0011 using this grammar

$S \rightarrow 0S1$

$\rightarrow 0011$

**Example:**

$S \rightarrow 01 \mid 0S1$

We have to make 000111 using the grammar

$S \rightarrow 0S1$

$\rightarrow 00S11$

### Context Free Grammar (CFG)

**Example 1:**

$L = \{a^n \mid n \geq 0\}$

$= \{\epsilon, a, aa, \dots\}$  here  $\epsilon$  means empty or nothing

**Grammar:**  $A \Rightarrow aA \mid \epsilon$

We want to make "aa" using this grammar

$A \Rightarrow aA$

$\Rightarrow aaA$

$\Rightarrow aa\epsilon$

$\Rightarrow aa$

**Example 2:**

$$L = \{a^n \mid n \geq 1\}$$

$$= \{a, aa, \dots\} \quad \text{here } \varepsilon \text{ means empty or nothing or stop}$$

**Grammar:**  $A \Rightarrow aA \mid a$

We want to make “aaa” using this grammar

$$A \Rightarrow aA$$

$$\Rightarrow aaA$$

$$\Rightarrow aaa$$

**Example 3:**

$$L = \{\text{set of all strings over } a, b\}$$

**Grammar:**  $A \Rightarrow aA \mid bA \mid \varepsilon$

We have to make “aab” using the grammar

$$A \rightarrow aA \mid bA \mid \varepsilon$$

$$A \rightarrow aA$$

$$\rightarrow aaA$$

$$\rightarrow aabA$$

$$\rightarrow aabe$$

**Example 4:**

$$L = \{\text{set of all strings which length at least } 2\}$$

**Grammar:**

$$S \rightarrow AAB$$

$$A \rightarrow a \mid b$$

$$B \rightarrow aB \mid bB \mid \varepsilon$$

We have to make “abb” using the grammar

$S \rightarrow AAB$

$\rightarrow aAB$

$\rightarrow abB$

$\rightarrow abbB$

$\rightarrow abb\varepsilon$

$\rightarrow abb$

**Example 5:**

$L = \{\text{set of all strings of at least 3 o's}\}$

**Grammar:**

$S \rightarrow EOEEOEE$

$E \rightarrow OE \mid 1E \mid \varepsilon$

We have to make "1000" using the grammar

**Example 6:**

$L = \{\text{set of all strings which length at most 2}\}$

**Grammar:**

$S \rightarrow AA$

$A \rightarrow a \mid b \mid \varepsilon$

We have to make "aa" using the grammar

$S \rightarrow AA$

$\rightarrow aA$

$\rightarrow aa$

**Example 7:**

Make a CFG which starts with "a" and ends with "b"

**Solution:**

**Grammar:**  $S \rightarrow aAb$

$A \rightarrow aA \mid bA \mid \varepsilon$

**Solution: Example 5**

$S \rightarrow EOEEOEE$

$\rightarrow 1EOEOEOE$

$\rightarrow 1\varepsilon EOEOEOE$

$\rightarrow 10EOEOE$

$\rightarrow 10\varepsilon EOEOE$

$\rightarrow 100\varepsilon OE$

$\rightarrow 1000\varepsilon$

**Example 7**

$S \rightarrow aAb$

$\rightarrow abAb$

$\rightarrow abaAb$

$\rightarrow aba\varepsilon b$

**Example 8:**

Make a CFG which starts and ends with different symbol

**Solution:**

**Grammar:**

$S \rightarrow aAb \mid bAa$

$A \rightarrow aA \mid bA \mid \varepsilon$

**Example 9:**

Make a CFG which starts and ends with same symbol

**Solution:**

**Grammar:**

$S \rightarrow aAa \mid bAb \mid \varepsilon \mid a \mid b$

$A \rightarrow aA \mid bA \mid \varepsilon$

**Example 10:**

Make a CFG for even length of a String

**Solution:**

**Grammar:**

$S \rightarrow BS \mid \varepsilon$

$B \rightarrow AA$

$A \rightarrow a \mid b$

**Example 8:** We want to make "abab" using the grammar

$S \rightarrow aAb$

$\rightarrow abAb$

$\rightarrow abaAb$

$\rightarrow aba\epsilon b$

**Example 9:** We want to make "abaa" using the grammar

$S \rightarrow aAa$

$\rightarrow abAa$

$\rightarrow abaAa$

$\rightarrow aba\epsilon a$

**Example 10:**

$S \rightarrow BS$

$\rightarrow AAS$

$\rightarrow aabbS$

$\rightarrow aabb\epsilon$

$\rightarrow aabb$



**REFERENCE (APA FORMAT):**

- Margaret Rouse - techtarget, **Abstraction**,  
<https://whatis.techtarget.com/definition/abstraction>
- Computational thinking, **What is Abstraction**,  
<https://computersciencewiki.org/index.php/Abstraction>
- Dinesh Thakur, **What is Abstraction? Explain Type of Abstraction**,  
<http://ecomputernotes.com/cpp/classes-in-c/type-of-abstraction>
- IEEE, **Control abstraction in parallel programming languages**,  
<https://ieeexplore.ieee.org/document/185467>
- GURU99, **What is Abstraction in OOPs? Learn with Java Example**,  
<https://www.guru99.com/java-data-abstraction.html>
- DZone, **Why Abstraction is Really Important**,  
<https://dzone.com/articles/why-abstraction-really>
- Tutorialspoint, **Java - Abstraction**,  
[https://www.tutorialspoint.com/java/java\\_abstraction](https://www.tutorialspoint.com/java/java_abstraction)