**Tutorial 1**

**Aim**: Implement Calculator with addition, multiplication, division and subtraction operations using **Client Server**and**Monolithic** Architectures; also draw class diagrams for the architectures.

**Monolithic Architecture:**

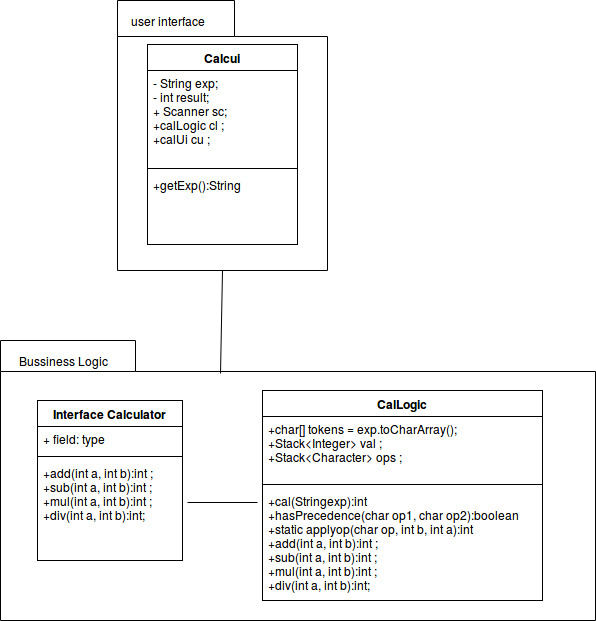
In [software engineering](https://en.wikipedia.org/wiki/Software_engineering), a monolithic application describes a single-tiered [software application](https://en.wikipedia.org/wiki/Application_software) in which the [user interface](https://en.wikipedia.org/wiki/User_interface) and data access code are combined into a single program from a single [platform](https://en.wikipedia.org/wiki/Platform_(computing)).

A monolithic application is self-contained, and independent from other computing applications. The design philosophy is that the application is responsible not just for a particular task, but can perform every step needed to complete a particular function.

In software engineering, a monolithic application describes a software application which is designed without modularity. Modularity is desirable, in general, as it supports reuse of parts of the application logic and also facilitates maintenance by allowing repair or replacement of parts of the application without requiring wholesale replacement.

Modularity is achieved to various extents by different modularization approaches. Code-based modularity allows developers to reuse and repair parts of the application, but development tools are required to perform these maintenance functions (e.g. the application may need to be recompiled). Object-based modularity provides the application as a collection of separate executable files which may be independently maintained and replaced without redeploying the entire application.

**Class Diagram for Calculator:**



Code:

**package** UserInterface;  
  
*/\*\*  
 \* Created by Amay on 1/21/2018.  
 \*/***import** java.io.\*;  
**import** java.util.\*;  
**import** BusinessLogic.\*;  
  
**public class** calUi {  
 **private** String **exp** = **new** String();  
 **private int result**;  
 **public** String getExp(){  
 Scanner sc = **new** Scanner(System.***in***);  
 System.***out***.println(**"Enter Expression:"**);  
 **exp** = sc.nextLine();  
 **if**(**exp**.length() > 20){  
 System.***out***.println(**"Maximum length of 20"**);  
 getExp();  
 }  
 **return exp**;  
 }  
 **public static void** main(String[] args) {  
 Scanner sc = **new** Scanner(System.***in***);  
  
 calLogic cl = **new** calLogic();  
 calUi cu = **new** calUi();  
  
 **while**(**true**){  
 cu.**result** = cl.cal(cu.getExp());  
 System.***out***.println(**"Result: "** + cu.**result** +**"\nContinue (Y/N)? "**);  
 String choice = sc.nextLine();  
 **if**(choice.equals(**"N"**) || choice.equals(**"n"**))  
 **break**;  
 }  
 }  
}

**package** BusinessLogic;  
  
*/\*\*  
 \* Created by Amay on 1/21/2018.  
 \*/***public interface** Calculator {  
 **public int** add(**int** a, **int** b);  
 **public int** sub(**int** a, **int** b);  
 **public int** mul(**int** a, **int** b);  
 **public int** div(**int** a, **int** b);  
}

**package** BusinessLogic;  
  
*/\*\*  
 \* Created by Amay on 1/21/2018.  
 \*/***import** java.util.Stack;  
  
**public class** calLogic **implements** Calculator{  
 **public int** cal(String exp){  
 **char**[] tokens = exp.toCharArray();  
  
 Stack<Integer> val = **new** Stack<Integer>();  
 Stack<Character> ops = **new** Stack<Character>();  
  
 **for**(**int** i=0;i<tokens.**length**;i++){  
 **if**(tokens[i] == **' '**)  
 **continue**;  
  
 **if**(tokens[i] >= **'0'** && tokens[i] <= **'9'**){  
 StringBuffer sbuf = **new** StringBuffer();  
 **while**(i < tokens.**length** && tokens[i] >= **'0'** && tokens[i] <= **'9'**)  
 sbuf.append(tokens[i++]);  
 val.push(Integer.*parseInt*(sbuf.toString()));  
 }  
  
 **else if**(tokens[i] == **'('**)  
 ops.push(tokens[i]);  
  
 **else if**(tokens[i] == **')'**){  
 **while**(ops.peek() != **'('**)  
 val.push(*applyop*(ops.pop(), val.pop(), val.pop()));  
 ops.pop();  
 }  
  
 **else if**(tokens[i] == **'+'** || tokens[i] == **'-'** || tokens[i] == **'\*'** || tokens[i] == **'/'**){  
 **while**(!ops.empty() && hasPrecedence(tokens[i], ops.peek()))  
 val.push(*applyop*(ops.pop(), val.pop(), val.pop()));  
 ops.push(tokens[i]);  
 }  
 }  
 **while**(!ops.empty())  
 val.push(*applyop*(ops.pop(), val.pop(), val.pop()));  
  
 **return** val.pop();  
 }  
  
 **public boolean** hasPrecedence(**char** op1, **char** op2){  
 **if**(op2 == **'('** || op2 == **')'**)  
 **return false**;  
 **if**((op1 == **'\*'** || op1 == **'/'**) && (op2 == **'+'** || op2 == **'-'**))  
 **return false**;  
 **else  
 return true**;  
 }  
  
 **public static int** applyop(**char** op, **int** b, **int** a){  
 **switch**(op){  
 **case '+'**: **return** a + b;  
 **case '-'**: **return** a - b;  
 **case '\*'**: **return** a \* b;  
 **case '/'**: **if**(b == 0)  
 **throw new** UnsupportedOperationException(**"Cannot divide by zero"**);  
 **return** a/b;  
 }  
 **return** 0;  
 }  
  
 @Override  
 **public int** add(**int** a, **int** b) {  
 **return** a + b;  
 }  
  
 @Override  
 **public int** sub(**int** a, **int** b) {  
 **return** a - b;  
 }  
  
 @Override  
 **public int** mul(**int** a, **int** b) {  
 **return** a \* b;  
 }  
  
 @Override  
 **public int** div(**int** a, **int** b) {  
 **if**(b == 0)  
 **throw new** UnsupportedOperationException(**"Cannot divide by zero"**);  
 **return** a/b;  
 }  
}

**Client Server Architecture**:

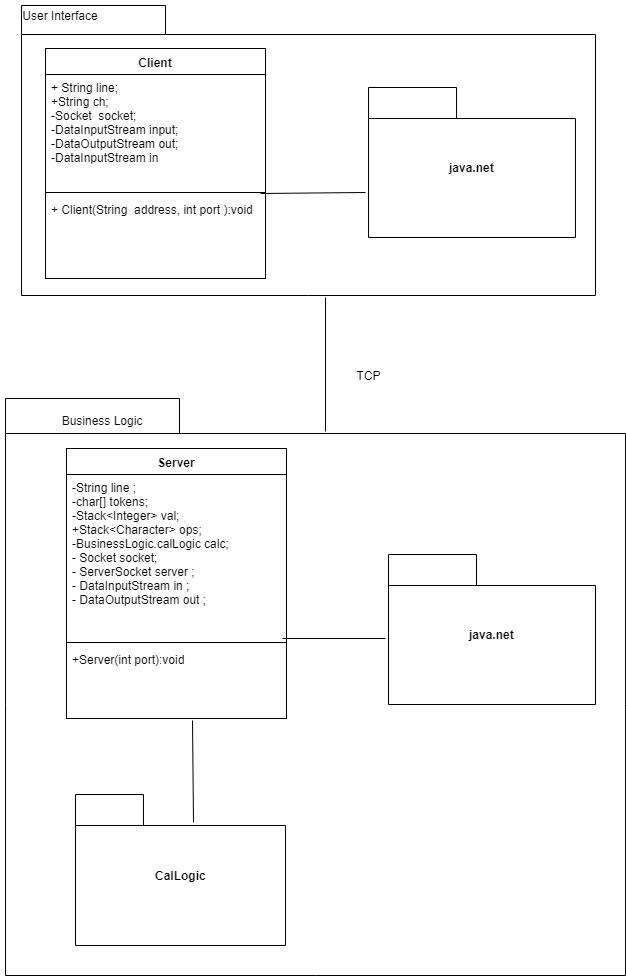
A client/server application is a piece of software that runs on a client computer and makes requests to a remote server. Many such applications are written in high-level visual programming languages where UI, forms, and most business logic reside in the client application. Often such applications are database applications that make database queries to a remote central database server (this can, however, get much more complicated than that and involve other communication methods).

In a database application, data related number crunching can occur on the remote database server where the processing is close to physical data. An example of a database query might be to return the sum of a field named "dollar amount" where the field name year is "2001". There may be hundreds of thousands of records but the client computer does not have to worry about fetching or sorting through all of them itself. The database server will sort through that and just return one small record with the result to the client.

A client / server application can be cross platform if it is written in a cross platform language, or it can be platform specific. In the case of a cross platform language there is an advantage that the application can potentially provide a user interface that is native in appearance to the OS or platform environment it is running under.

An issue of client/server is that the application must be installed on each users computer. Depending on the complexity of the program, the environment it is written in, and the care the developer took to package the program, this can be as easy as creating a shortcut to an executable on a shared network drive or it can be as hard as spending hours installing and configuring runtime software and components on each client computer.

**Class Diagram for Calculator (Client-Server):**



**Code:**

**package** userInterface;  
**import** java.net.\*;  
**import** java.io.\*;  
**public class** Client  
{  
 **private** Socket **socket** = **null**;  
 **private** DataInputStream **input** = **null**;  
 **private** DataOutputStream **out** = **null**;  
 **private** DataInputStream **in** = **null**;  
 **public** Client(String address, **int** port)  
 {  
 **try** {  
 **socket** = **new** Socket(address, port);  
 System.***out***.println(**"Connected"**);  
 **input** = **new** DataInputStream(System.***in***);  
 **out** = **new** DataOutputStream(**socket**.getOutputStream());  
 **in** = **new** DataInputStream(**socket**.getInputStream());  
 }  
 **catch**(UnknownHostException u)  
 {  
 System.***out***.println(u);  
 }  
 **catch**(IOException i)  
 {  
 System.***out***.println(i);  
 }  
 String line = **""**;  
 String ch;  
 **while**(**true**)  
 {  
 **try** {  
 System.***out***.println(**"Enter Expression..."**);  
 line = **input**.readLine();  
 **out**.writeUTF(line);  
 line = **in**.readUTF();  
 System.***out***.println(line + **"\nContinue (Y/N)?:"**);  
 ch = **input**.readLine();  
 **if**(ch.equals(**"N"**) || ch.equals(**"n"**)){  
 **out**.writeUTF(**"Over"**);  
 **break**;  
 }  
 }  
 **catch**(IOException i)  
 {  
 System.***out***.println(i);  
 }  
 }  
 **try** {  
 **input**.close();  
 **out**.close();  
 **socket**.close();  
 }  
 **catch**(IOException i)  
 {  
 System.***out***.println(i);  
 }  
 }  
 **public static void** main(String args[])  
 {  
 Client client = **new** Client(**"127.0.0.1"**, 5000);  
 }  
}

**package** businessLogic;  
**import** java.net.\*;  
**import** java.io.\*;  
**import** java.util.\*;  
  
**public class** Server  
{  
  
 **private** Socket **socket** = **null**;  
 **private** ServerSocket **server** = **null**;  
 **private** DataInputStream **in** = **null**;  
 **private** DataOutputStream **out** = **null**;  
  
 **private** BusinessLogic.calLogic **calc**;  
  
 **private** String **line** = **""**;  
 **private char**[] **tokens**;  
 **private** Stack<Integer> **val** = **new** Stack<Integer>();  
 Stack<Character> **ops** = **new** Stack<Character>();  
  
 **public** Server(**int** port)  
 {  
  
 **try** {  
 **server** = **new** ServerSocket(port);  
 System.***out***.println(**"Server started"**);  
  
 System.***out***.println(**"Waiting for a client ..."**);  
  
 **socket** = **server**.accept();  
 System.***out***.println(**"Client accepted"**);  
  
  
 **in** = **new** DataInputStream(  
 **new** BufferedInputStream(**socket**.getInputStream()));  
  
 **out** = **new** DataOutputStream(**socket**.getOutputStream());  
  
 **calc** = **new** BusinessLogic.calLogic();  
  
 **while** (**true**)  
 {  
 **try** {  
 **line** = **in**.readUTF();  
 System.***out***.println(**line**);  
 **if**(**line**.equals(**"Over"**) || **line**.equals(**"over"**))  
 **break**;  
 **int** result = **calc**.cal(**line**);  
 System.***out***.println(result);  
 **out**.writeUTF(Integer.*toString*(result));  
  
 }  
 **catch**(IOException i)  
 {  
 System.***out***.println(i);  
 }  
 }  
 System.***out***.println(**"Closing connection"**);  
  
  
 **socket**.close();  
 **in**.close();  
 }  
 **catch**(IOException i)  
 {  
 System.***out***.println(i);  
 }  
 }  
  
 **public static void** main(String args[])  
 {  
 Server server = **new** Server(5000);  
 }  
}