sep 24, 18 22:35 **stack.h** Page 1/1

```c
#ifndef __STACK_H__
#define __STACK_H__

/*** Stack ***/

#include "node.h"

typedef struct {
  node_t* head;
} stack_t;

// Crea un stack.
// Pre: self apunta a un sector valido de memoria.
void stack_create(stack_t *self);
// Agrega el elemento pasado por parametro en el stack.
// Pre: self apunta un sector valido de memoria.
void stack_push(stack_t *self, int data);
// Devuelve el elemento que esta en le tope de la pila pero sin sacarlo.
// Pre: self apunta un sector valido de memoria.
int stack_top(stack_t *self);
// Devuelve el elemento que esta en le tope de la pila y lo saca.
// Pre: self apunta un sector valido de memoria.
int stack_pop(stack_t *self);
// Destruye la instancia self liberando sus recursos.
// Pre: self apunta un sector valido de memoria
//      self fue inicializado mediante stack_create
void stack_destroy(stack_t *self);

#endif
```

sep 24, 18 22:35 **stack.c** Page 1/1

```c
#include "stack.h"
#include <stdio.h>
#include <stdlib.h>
#include "node.h"

void stack_create(stack_t *self) {
  self→head = NULL;
}

void stack_push(stack_t *self, int data) {
  node_t *node = malloc(sizeof(node_t));
  node_create(node);
  node_set_data(node, data);

  if(self→head ≠ NULL) {
    node_set_next(node, self→head);
  }
  self→head = node;
}

int stack_pop(stack_t *self) {
  int data = node_get_data(self→head);
  node_t* to_delete = self→head;
  self→head = node_get_next(self→head);
  free(to_delete);
  return data;
}

int stack_top(stack_t *self) {
  return node_get_data(self→head);
}

void stack_destroy(stack_t *self) {
  if(self→head ≠ NULL) {
    stack_pop(self);
    stack_destroy(self);
  }
}
```

```c
#ifndef __SOCKET_H__
#define __SOCKET_H__

/*** Socket ***/

#include <stddef.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>

typedef struct {
  int skt_fd; // socket file descriptor
} socket_t;

// Crea un socket.
// Pre: self apunta a un sector valido de memoria.
int socket_create(socket_t *self);
// Deshabilita la lectura de datos del socket.
// Pre: self apunta un sector valido de memoria.
int socket_shutdown_read(socket_t *self);
// Deshabilita la escritura de datos del socket.
// Pre: self apunta un sector valido de memoria.
int socket_shutdown_write(socket_t *self);
// Deshabilita la lectura y escritura de datos del socket.
// Pre: self apunta un sector valido de memoria.
int socket_shutdown(socket_t *self);
// Destruye la instancia self liberando sus recursos.
// Pre: self fue inicializado mediante socket_create o socket_accept.
int socket_destroy(socket_t *self);
// Deshabilita la lectura y escritura de datos del socket.
// Destruye la instancia self liberando sus recursos.
// Pre: self fue inicializado mediante socket_create o socket_accept.
int socket_shutdown_destroy(socket_t *self);
// Establece a que IP y puerto se quiere asociar el Socket.
// Pre: self apunta un sector valido de memoria.
int socket_bind(socket_t *self, unsigned short port);
/* Define cuantas conecciones en espera pueden esperar hasta ser
 * aceptadas en el Socket.
 */
// Pre: self apunta un sector valido de memoria
int socket_listen(socket_t *self, int max_standby_conn);
/* Establece una conneccion a la maquina remota, indicada
 * mediante host_name y port.
 */
// Pre: self apunta un sector valido de memoria.
int socket_connect(socket_t *self, const char *host_name, unsigned short port);
// Crea un nuevo socket servidor para la comunicacion con el socket cliente.
// Pre: self apunta un sector valido de memoria.
// Post: Retorna el file descriptor del socket aceptado (socket servidor).
int socket_accept(socket_t *self, socket_t *socket_accepted);
// Envia el contenido de buffer en forma de bytes.
// Pre: self apunta un sector valido de memoria.
/* Post: Valores de retorno:
 * s < 0 : Hubo un error inesperado.
 * s == 0 : El socket fue cerrado.
 * s > 0 : s bytes fueron enviados.
 */
int socket_send(socket_t *self, const void *buffer, size_t length);
// Envia el int con signo "i" a traves del socket.
// Si detecta un error aplica un shutdown y destroy del socket.
// Post: Retorna  0 en caso de funcionar bien.
//       Retorna −1 en caso de error.
int socket_send_int(socket_t *self, int i);
// Envia el char con signo "c" a traves del socket.
// Si detecta un error aplica un shutdown y destroy del socket.
```



```c
// Post: Retorna  0 en caso de funcionar bien.
//       Retorna −1 en caso de error.
int socket_send_char(socket_t *socket, char c);
// Recibe el contenido de buffer en forma de bytes.
// Pre: self apunta un sector valido de memoria.
/* Post: Valores de retorno:
 * s < 0 : Hubo un error inesperado.
 * s == 0 : El socket fue cerrado.
 * s > 0 : s bytes fueron recibidos.
 */
int socket_receive(socket_t *self, void *buffer, size_t length);
// Recive el int con signo "i" a traves del socket.
// Si detecta un error aplica un shutdown y destroy del socket.
// Post: Retorna  0 en caso de funcionar bien.
//       Retorna −1 en caso de error.
int socket_recv_int(socket_t *socket, int *i);
// Recive el char con signo "c" a traves del socket.
// Si detecta un error aplica un shutdown y destroy del socket.
// Post: Retorna  0 en caso de funcionar bien.
//       Retorna −1 en caso de error.
int socket_recv_char(socket_t *socket, char *c);
// Recive el char sin signo "c" a traves del socket.
// Si detecta un error aplica un shutdown y destroy del socket.
// Post: Retorna  0 en caso de funcionar bien.
//       Retorna −1 en caso de error.
int socket_recv_unsigned_char(socket_t *socket, unsigned char *c);

#endif
```

```c
#define _POSIX_C_SOURCE 200112L

#include <string.h>
#include "socket.h"
#include <stddef.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <stdbool.h>

#define SUCCESS 0
#define ERROR -1

#define SIZE_OF_INT   4 // bytes
#define SIZE_OF_CHAR  1 // bytes

int socket_create(socket_t *self) {
  self->skt_fd = socket(AF_INET, SOCK_STREAM, 0);
  return self->skt_fd;
}

int socket_bind(socket_t *self, unsigned short port) {
  struct sockaddr_in skt_addr;
  skt_addr.sin_family = AF_INET;
    skt_addr.sin_port = htons(port);
    skt_addr.sin_addr.s_addr = INADDR_ANY;

  return bind(self->skt_fd, (struct sockaddr *) &skt_addr, sizeof(skt_addr));
}

int socket_connect(socket_t *self, const char* hostname, unsigned short port) {
    bool are_we_connected = false;
    int s = 0;
    struct addrinfo hints;
    struct addrinfo *result, *ptr;

    char portString[6];
    snprintf(portString, sizeof(portString), "%d", port);

    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET;       /* IPv4 (or AF_INET6 for IPv6)     */
    hints.ai_socktype = SOCK_STREAM; /* TCP  (or SOCK_DGRAM for UDP)    */
    hints.ai_flags = 0;              /* None (or AI_PASSIVE for server) */

    s = getaddrinfo(hostname, portString, &hints, &result);

    for (
        ptr = result;
        ptr ≠ NULL ∧ are_we_connected ≡ false;
        ptr = ptr->ai_next
    ) {
        if (self->skt_fd ≠ ERROR) {
            s = connect(self->skt_fd, ptr->ai_addr, ptr->ai_addrlen);
            if (s ≡ ERROR) close(self->skt_fd);
            are_we_connected = (s ≠ -1);
        }
    }

    freeaddrinfo(result);

```



```c
    if (are_we_connected ≡ false) return ERROR;

    return SUCCESS;
}

int socket_listen(socket_t *self, int max_standby_conn) {
  return listen(self->skt_fd, max_standby_conn);
}

int socket_accept(socket_t *self, socket_t *socket_accepted) {
  socket_accepted->skt_fd = accept(self->skt_fd, (struct sockaddr*) NULL, NULL);
  return socket_accepted->skt_fd;
}

int socket_send(socket_t *self, const void *buffer, size_t lenght) {
  return send(self->skt_fd, buffer, lenght, MSG_NOSIGNAL);
}

int socket_send_int(socket_t *socket, int i) {
    int i_big_endian = htonl(i);
    int bytes_sent = 0;
    int skt_still_open = 1;
    while (SIZE_OF_INT > bytes_sent ∧ skt_still_open) {
        int s = socket_send(socket, &i_big_endian , SIZE_OF_INT - bytes_sent);
        if (s ≡ -1) {
            socket_shutdown(socket);
            socket_destroy(socket);
            return ERROR;
        } else if (s ≡ 0) {
            skt_still_open = 0;
            return ERROR;
        } else {
            bytes_sent += s;
        }
    }

    return SUCCESS;
}

int socket_send_char(socket_t *socket, char i) {
    int bytes_sent = 0;
    int skt_still_open = 1;
    while (SIZE_OF_CHAR > bytes_sent ∧ skt_still_open) {
        int s = socket_send(socket, &i , SIZE_OF_CHAR - bytes_sent);
        if (s ≡ -1) {
            socket_shutdown(socket);
            socket_destroy(socket);
            return ERROR;
        } else if (s ≡ 0) {
            skt_still_open = 0;
            return ERROR;
        } else {
            bytes_sent += s;
        }
    }
    return SUCCESS;
}

int socket_receive(socket_t *self, void *buffer, size_t lenght) {
    return read(self->skt_fd, buffer, lenght);
}

int socket_recv_int(socket_t *socket, int *i) {
    int bytes_recv = 0;
    int skt_still_open = 1;
    while (SIZE_OF_INT > bytes_recv ∧ skt_still_open) {
```

```
133         int s = socket_receive(socket, i, SIZE_OF_INT − bytes_recv);
134         if (s ≡ −1) {
135             socket_shutdown(socket);
136             socket_destroy(socket);
137             return ERROR;
138         } else if (s ≡ 0) {
139             skt_still_open = 0;
140             return ERROR;
141         } else {
142             bytes_recv += s;
143         }
144     }
145
146     *i = ntohl(*i);
147
148     return SUCCESS;
149 }
150
151 int socket_recv_char(socket_t *socket, char *c) {
152     int bytes_recv = 0;
153     int skt_still_open = 1;
154     while (SIZE_OF_CHAR > bytes_recv ∧ skt_still_open) {
155         int s = socket_receive(socket, c, SIZE_OF_CHAR − bytes_recv);
156         if (s ≡ −1) {
157             socket_shutdown(socket);
158             socket_destroy(socket);
159             return ERROR;
160         } else if (s ≡ 0) {
161             skt_still_open = 0;
162             return ERROR;
163         } else {
164             bytes_recv += s;
165         }
166     }
167     return SUCCESS;
168 }
169
170 int socket_recv_unsigned_char(
171     socket_t *socket,
172     unsigned char *c
173 ) {
174     int bytes_recv = 0;
175     int skt_still_open = 1;
176     while (SIZE_OF_CHAR > bytes_recv ∧ skt_still_open) {
177         int s = socket_receive(socket, c, SIZE_OF_CHAR − bytes_recv);
178         if (s ≡ −1) {
179             socket_shutdown(socket);
180             socket_destroy(socket);
181             return ERROR;
182         } else if (s ≡ 0) {
183             skt_still_open = 0;
184             return ERROR;
185         } else {
186             bytes_recv += s;
187         }
188     }
189     return SUCCESS;
190 }
191
192 int socket_shutdown_read(socket_t *self) {
193   return shutdown(self→skt_fd, SHUT_RD);
194 }
195
196 int socket_shutdown_write(socket_t *self) {
197   return shutdown(self→skt_fd, SHUT_WR);
198 }
```



```
199
200 int socket_shutdown(socket_t *self) {
201     return shutdown(self→skt_fd, SHUT_RDWR);
202 }
203
204 int socket_destroy(socket_t *self) {
205   return close(self→skt_fd);
206 }
207
208 int socket_shutdown_destroy(socket_t *self) {
209     if (
210         (shutdown(self→skt_fd, SHUT_RDWR) ≡ ERROR) ∨
211         (close(self→skt_fd)               ≡ ERROR)
212     ) {
213         return ERROR;
214     } else {
215         return SUCCESS;
216     }
217 }
```

sep 24, 18 22:35 **server.h** Page 1/1

```c
#ifndef __SERVER_H__
#define __SERVER_H__

// Funcion encargada de realizar las tareas del servidor.
int server(char *argv[]);

#endif
```

sep 24, 18 22:35 **server.c** Page 1/2

```c
#include <stdio.h>
#include <stdlib.h>
#include "socket.h"
#include "java_virtual_machine.h"
#include "bytecodes.h"

#define MAX_STANDBY_CONNS 10

#define ERROR  -1
#define SUCCESS 0

int server(char *argv[]) {
  // argv[2] = Port

  if (argv[2] == NULL) return ERROR;

  short unsigned int port = (short unsigned int) strtol(argv[2], NULL, 10);

  socket_t skt_acceptor;
  socket_t skt_server;

  if (socket_create(&skt_acceptor) == ERROR)                    return ERROR;
  if (socket_bind(&skt_acceptor, port) == ERROR)                return ERROR;
  if (socket_listen(&skt_acceptor, MAX_STANDBY_CONNS) == ERROR) return ERROR;
  if (socket_accept(&skt_acceptor, &skt_server) == ERROR)       return ERROR;

  int N;

  if (socket_recv_int(&skt_server, &N) == ERROR) {
    socket_shutdown_destroy(&skt_acceptor);
    return ERROR;
  }

  unsigned char bytecode;
  char c;

  jvm_t java_virtual_machine;
  jvm_create(&java_virtual_machine, N);

  printf("%s\n", "Bytecode trace");

  while(socket_recv_unsigned_char(&skt_server, &bytecode) == SUCCESS) {
    switch(bytecode) {
      case BIPUSH: printf("bipush\n");
        if (socket_recv_char(&skt_server, &c) == ERROR) {
          socket_shutdown_destroy(&skt_acceptor);
          jvm_destroy(&java_virtual_machine);
          return ERROR;
        }
        jvm_execute_instruction_bipush(&java_virtual_machine, c);  break;
      case DUP: printf("dup\n");
        jvm_execute_instruction_dup(&java_virtual_machine);        break;
      case ISTORE: printf("istore\n");
        if (socket_recv_char(&skt_server, &c) == ERROR) {
          socket_shutdown_destroy(&skt_acceptor);
          jvm_destroy(&java_virtual_machine);
          return ERROR;
        }
        jvm_execute_instruction_istore(&java_virtual_machine, c);  break;
      case ILOAD: printf("iload\n");
        if (socket_recv_char(&skt_server, &c) == ERROR) {
          socket_shutdown_destroy(&skt_acceptor);
          jvm_destroy(&java_virtual_machine);
          return ERROR;
        }
        jvm_execute_instruction_iload(&java_virtual_machine, c);  break;
```

sep 24, 18 22:35 **server.c** Page 2/2

```
67          case IADD: printf("iadd\n");
68            jvm_execute_instruction_iadd(&java_virtual_machine);      break;
69          case ISUB: printf("isub\n");
70            jvm_execute_instruction_isub(&java_virtual_machine);      break;
71          case IMUL: printf("imul\n");
72            jvm_execute_instruction_imul(&java_virtual_machine);      break;
73          case IDIV: printf("idiv\n");
74            jvm_execute_instruction_idiv(&java_virtual_machine);      break;
75          case IREM: printf("irem\n");
76            jvm_execute_instruction_irem(&java_virtual_machine);      break;
77          case INEG: printf("ineg\n");
78            jvm_execute_instruction_ineg(&java_virtual_machine);      break;
79          case IAND: printf("iand\n");
80            jvm_execute_instruction_iand(&java_virtual_machine);      break;
81          case IOR: printf("ior\n");
82            jvm_execute_instruction_ior(&java_virtual_machine);       break;
83          case IXOR: printf("ixor\n");
84            jvm_execute_instruction_ixor(&java_virtual_machine);      break;
85        }
86      }
87
88      printf("\nVariables dump\n");
89
90      int *variables_array = jvm_get_variables_array(&java_virtual_machine);
91      for(int i=0; i<N; ++i) {
92        printf("%08x\n", variables_array[i]);
93        int n = variables_array[i];
94        if(socket_send_int(&skt_server, n) ≡ ERROR) {
95          socket_shutdown_destroy(&skt_acceptor);
96          jvm_destroy(&java_virtual_machine);
97          return ERROR;
98        }
99      }
100
101     socket_shutdown_destroy(&skt_acceptor);
102     socket_shutdown_destroy(&skt_server);
103     jvm_destroy(&java_virtual_machine);
104
105     return SUCCESS;
106 }
```

sep 24, 18 22:35 **node.h** Page 1/1

```
1  #ifndef __NODE_H__
2  #define __NODE_H__
3
4  /*** Nodo ***/
5
6  typedef struct node_t{
7    int data;
8    struct node_t* next;
9  } node_t;
10
11 // Crea el nodo.
12 // Pre: self apunta un sector valido de memoria
13 void node_create(node_t* self);
14 // Setea el dato "data" en el nodo.
15 // Pre: self apunta un sector valido de memoria
16 void node_set_data(node_t* self, int data);
17 // Devuelve el dato que guarda el nodo.
18 // Pre: self apunta un sector valido de memoria
19 int node_get_data(node_t* self);
20 // Setea la direccion del nodo al que apunta la instancia de nodo self.
21 // Pre: self apunta un sector valido de memoria
22 void node_set_next(node_t* self, node_t* next);
23 // Devuelve el nodo apuntado por la instacia "self".
24 // Pre: self apunta un sector valido de memoria
25 node_t* node_get_next(node_t* self);
26 // Destruye los recursos asosciados a la instacia de nodo self.
27 // Pre: self apunta un sector valido de memoria
28 void node_destroy(node_t* self);
29
30 #endif
```

sep 24, 18 22:35 **node.c** Page 1/1

```c
#include "node.h"
#include <stdio.h>

void node_create(node_t* self) {
  self→data = 0;
  self→next = NULL;
}

void node_set_data(node_t* self, int data) {
  self→data = data;
}

int node_get_data(node_t* self) {
  return self→data;
}

void node_set_next(node_t* self, node_t* next) {
  self→next = next;
}

node_t* node_get_next(node_t* self) {
  return self→next;
}

void node_destroy(node_t* self) {
}
```

sep 24, 18 22:35 **main.c** Page 1/1

```c
#include <string.h>

#include "client.h"
#include "server.h"

#include "stack.h"
#include <stdio.h>

#define SUCCESS 0
#define ERROR -1

int main(int argc, char* argv[]) {
  if (argv[1] ≡ NULL) return ERROR;

  if      (strcmp(argv[1], "client")  ≡ SUCCESS) return client(argv);
  else if (strcmp(argv[1], "server")  ≡ SUCCESS) return server(argv);
  else                                return ERROR;
}
```

sep 24, 18 22:35 **java_virtual_machine.h** Page 1/1

```c
#ifndef __JAVA_VIRTUAL_MACHINE_H__
#define __JAVA_VIRTUAL_MACHINE_H__

/*** Java Virtual Machine ***/

#include "stack.h"

typedef struct {
  stack_t* stack;
  int *numbers;
} jvm_t;

// Crea la virtual machine.
void jvm_create(jvm_t *self, int N);
// Ejecuta la instruccion bipush.
// Pre: self apunta un sector valido de memoria.
void jvm_execute_instruction_bipush(jvm_t *self, int data);
// Ejecuta la instruccion dup.
// Pre: self apunta un sector valido de memoria.
void jvm_execute_instruction_dup(jvm_t *self);
// Ejecuta la instruccion store.
// Pre: self apunta un sector valido de memoria.
void jvm_execute_instruction_istore(jvm_t *self, char c);
// Ejecuta la instruccion load.
// Pre: self apunta un sector valido de memoria.
void jvm_execute_instruction_iload(jvm_t *self, char c);
// Ejecuta la instruccion and.
// Pre: self apunta un sector valido de memoria.
void jvm_execute_instruction_iand(jvm_t *self);
// Ejecuta la instruccion or.
// Pre: self apunta un sector valido de memoria.
void jvm_execute_instruction_ior(jvm_t *self);
// Ejecuta la instruccion xor.
// Pre: self apunta un sector valido de memoria.
void jvm_execute_instruction_ixor(jvm_t *self);
// Ejecuta la instruccion rem.
// Pre: self apunta un sector valido de memoria.
void jvm_execute_instruction_irem(jvm_t *self);
// Ejecuta la instruccion neg.
// Pre: self apunta un sector valido de memoria.
void jvm_execute_instruction_ineg(jvm_t *self);
// Ejecuta la instruccion div.
// Pre: self apunta un sector valido de memoria.
void jvm_execute_instruction_idiv(jvm_t *self);
// Ejecuta la instruccion add.
// Pre: self apunta un sector valido de memoria.
void jvm_execute_instruction_iadd(jvm_t *self);
// Ejecuta la instruccion mul.
// Pre: self apunta un sector valido de memoria.
void jvm_execute_instruction_imul(jvm_t *self);
// Ejecuta la instruccion sub.
// Pre: self apunta un sector valido de memoria.
void jvm_execute_instruction_isub(jvm_t *self);
// Devuelve las variables.
// Pre: self apunta un sector valido de memoria.
int* jvm_get_variables_array(jvm_t *self);
// Libera los recursos asociados a la instacia self.
// Pre: self apunta un sector valido de memoria.
void jvm_destroy(jvm_t *self);

#endif
```

sep 24, 18 22:35 **java_virtual_machine.c** Page 1/2

```c
#include "java_virtual_machine.h"
#include <stdio.h>
#include <stdlib.h>

void jvm_create(jvm_t *self, int N) {
  self→stack = malloc(sizeof(stack_t));
  stack_create(self→stack);

  self→numbers = malloc(sizeof(int)*N);
  for (int i=0; i<N; ++i) self→numbers[i] = 0;
}

void jvm_execute_instruction_bipush(jvm_t *self, int number) {
  stack_push(self→stack, number);
}

void jvm_execute_instruction_dup(jvm_t *self) {
  stack_push(self→stack, stack_top(self→stack));
}

void jvm_execute_instruction_istore(jvm_t *self, char c) {
  char hex[3];
  hex[0] ='\0';
  snprintf(hex, sizeof(hex), "%02X", c);
  hex[2] ='\0';
  int pos = (int) strtol(hex, (char **)NULL, 10);
  self→numbers[pos] = stack_pop(self→stack);
}

void jvm_execute_instruction_iload(jvm_t *self, char c) {
  char hex[3];
  hex[0] ='\0';
  snprintf(hex, sizeof(hex), "%02X", c);
  hex[2] ='\0';
  int pos = (int) strtol(hex, (char **)NULL, 10);
  stack_push(self→stack, self→numbers[pos]);
}

void jvm_execute_instruction_iadd(jvm_t *self) {
  stack_push(self→stack, stack_pop(self→stack) + stack_pop(self→stack));
}

void jvm_execute_instruction_isub(jvm_t *self) {
  int a = stack_pop(self→stack);
  int b = stack_pop(self→stack);
  stack_push(self→stack, b − a);
}

void jvm_execute_instruction_imul(jvm_t *self) {
  stack_push(self→stack, stack_pop(self→stack) * stack_pop(self→stack));
}

void jvm_execute_instruction_idiv(jvm_t *self) {
  int a = stack_pop(self→stack);
  int b = stack_pop(self→stack);
  stack_push(self→stack, b / a);
}

void jvm_execute_instruction_irem(jvm_t *self) {
  int a = stack_pop(self→stack);
  int b = stack_pop(self→stack);
  stack_push(self→stack, b % a);
}

void jvm_execute_instruction_ineg(jvm_t *self) {
  stack_push(self→stack, −stack_pop(self→stack));
```

sep 24, 18 22:35 **java_virtual_machine.c** Page 2/2

```
67  }
68
69  void jvm_execute_instruction_iand(jvm_t *self) {
70    stack_push(self→stack, stack_pop(self→stack) & stack_pop(self→stack));
71  }
72
73  void jvm_execute_instruction_ior(jvm_t *self) {
74    stack_push(self→stack, stack_pop(self→stack) | stack_pop(self→stack));
75  }
76
77  void jvm_execute_instruction_ixor(jvm_t *self) {
78    stack_push(self→stack, stack_pop(self→stack) ^ stack_pop(self→stack));
79  }
80
81  int* jvm_get_variables_array(jvm_t *self) {
82    return self→numbers;
83  }
84
85  void jvm_destroy(jvm_t *self) {
86    stack_destroy(self→stack);
87    free(self→stack);
88    free(self→numbers);
89  }
```

sep 24, 18 22:35 **client.h** Page 1/1

```
1  #ifndef __CLIENT_H__
2  #define __CLIENT_H__
3
4  // Funcion encargada de realizar las tareas del cliente.
5  int client(char *argv[]);
6
7  #endif
```

sep 24, 18 22:35 **client.c** Page 1/1

```c
#include <stdlib.h>
#include <stdio.h>
#include "socket.h"

#define ERROR -1
#define SUCCESS 0

int client(char *argv[]) {
  // argv[2] = Hostname
  // argv[3] = Port
  // argv[4] = N
  // argv[5] = Namefile

  if(argv[2] == NULL || argv[3] == NULL || argv[4] == NULL) return ERROR;

  FILE* input;

  if (argv[5] == NULL) {
    input = stdin;
  } else {
    input = fopen(argv[5], "rb");
  }

  if (input == NULL) return ERROR;

  socket_t skt_client;

  short unsigned int port = (short unsigned int) strtol(argv[3], NULL, 10);
  int N = (int) strtol(argv[4], NULL, 10);

  if (socket_create(&skt_client) == ERROR) return ERROR;
  if (socket_connect(&skt_client, argv[2], port) == ERROR) return ERROR;

  if (socket_send_int(&skt_client, N) == ERROR) return ERROR;

  do {
    char c = fgetc(input);
    if(socket_send_char(&skt_client, c) == ERROR) {
      if(input != stdin) fclose(input);
      return ERROR;
    }
  } while (!feof(input));

  if(input != stdin) fclose(input);

  socket_shutdown_write(&skt_client);

  printf("%s\n", "Variables dump");

  for(int i = 0; i<N; ++i) {
    int number;
    if (socket_recv_int(&skt_client, &number) == ERROR) return ERROR;
    printf("%08x\n", number);
  }

  socket_shutdown_read(&skt_client);

  socket_destroy(&skt_client);

  return SUCCESS;
}
```

sep 24, 18 22:35 **bytecodes.h** Page 1/1

```c
#define ISTORE 0x36
#define ILOAD  0x15
#define BIPUSH 0x10
#define DUP    0x59
#define IAND   0x7E
#define IXOR   0x82
#define IOR    0x80
#define IREM   0x70
#define INEG   0x74
#define IDIV   0x6C
#define IADD   0x60
#define IMUL   0x68
#define ISUB   0x64
```

sep 24, 18 22:35 **Table of Content** Page 1/1

Table of Contents