Implementing a Stack

- A **stack** is a **data structure** with **two** operation:
  - void Push(int x): put the value "x" in the stack. The value "x" is put "at the top" of the stack.
  - int Pop(): remove the top value off the stack. The value removed is returned.

Example:

```
   (1) Initially:     +----------+  (empty stack)

   (2) After Push(4): +----------+
                      |    4     |
                      +----------+

   (3) After Push(9): +----------+
                      |    9     |    (new value get put ON TOP)
                      +----------+
                      |    4     |
                      +----------+

   (4) After Push(1): +----------+
                      |    1     |    (new value get put ON TOP again)
                      +----------+
                      |    9     |
                      +----------+
                      |    4     |
                      +----------+

Suppose we have 3 variables "a", "b" and "c" defined....

   (5) After a = Pop():  +----------+
                         |    9     | (top value is REMOVED !)
                         +----------+
                         |    4     |
                         +----------+

         and  a = 1

   (6) After b = Pop():  +----------+
                         |    4     | (top value is REMOVED again !)
                         +----------+

         and  b = 9

   (7) After c = Pop():  +----------+ (Stack is empty)

         and  c = 4
```

- **Stack in a program versus the *System Stack***
  - The **stack** that you learned in **Java** is usually **implemented** ***using*** an **array variable**

    Example: (**CS171** material)

```
public class Stack
{
   int[] A;
   int  stacktop;      // Points to the top of the stack

   public Stack(int size)
   {
      A = new int[size];        // Create the array to hold values in stack
      stacktop = -1;
   }

   public void push( int x )
   {

      A[++stacktop] = x;          // Move stack top and put x on the stack
   }

   public int pop( )
   {
      return A[stacktop--];
   }
}
```

  - When a **program** is **running**, the **computer system** will need to **store** various **information items (variables)** in a ***stack***

    Therefore:

    - All modern **computers** maintain a **program stack** (a.k.a. ***System* stack**) to manage **information**

    Information stored in the ***system* stack** include:

    - **function activation information** (created when a **function** is **called**

      (These **function activation information** will be **removed (popped)** when a function **returns (exits)**)

- The reason we **want** to use a **stack** is the **order** the elements enter and leave the stack is **exactly** the same order of **function activation and deactivation**.
- The **system stack** is (always) stored inside the **memory** (memory stores everything :-))
- When a **program** is **running**, the **memory** is **organized** as follows:

```
 Memory:

    0 +----------+
      |          |  <--- this part contains the "code"
      |    A     |       This part of the memory is "write protected"
      |          |       (can not - and should not - be modified)
      |          |
      +----------+
      |          | <--- this part contains "static variables"
      |    B     |      These variables exist at the start of the
      |          |      program (a.k.a.: compiled-time variables)
      +----------+
      |          | ^
      |          | | <---  this part grows when the program creates
      |          | |       new objects: reserve space for instance
      |    C     | |       variables in the new object
      |          | v
      |          | v <--- Direction of growth is "downwards"
      |          | v
      +==========+
      |          | <---- this part is "free" memory
      |          |       (Free means: unreserved !)
      |          |
      |          |
      +==========+
      |          | ^
      |          | ^ <--- Direction of growth is "upwards"
      |          | ^
      |    D     | |
      |          | | <--- This part grows when the program invokes
      |  System  | |      a function/method, reserve space for:
      |  stack   | |      (1) return address
      |          | |      (2) parameter variables
      |          | v      (1) local variables
      +----------+

 1. Parts A and B exist as soon as the program starts execution
    and will exist throughout the program execution.

 2. Part C will grow when program creates new objects
    (with the new operator in Java or malloc() in C)
    Part C is called the "system heap"

 3. Part D will grow when program invokes a function/method
    Part D is called the "system stack"
```

- **Implementing the System Stack**
    - Just like a **stack** object in **Java**, in order to **implement** a **stack**, we need:
        - An **array**
        - A **StackTop** index

- ***Unlike*** in a **programming language**, we have access to the **entire computer memory** when we program in **assembler** !!!

  So the **system stack** that is **located** at the ***end*** of the **memory area**:

```
Memory:

  0 +----------+
    |          |  <--- this part contains the "code"
    |    A     |       This part of the memory is "write protected"
    |          |       (can not - and should not - be modified)
    |          |
    +----------+
    |          | <--- this part contains "static variables"
    |    B     |      These variables exist at the start of the
    |          |      program (a.k.a.: compiled-time variables)
    +----------+
    |          | ^
    |          | | <---  this part grows when the program creates
    |          | |       new objects: reserve space for instance
    |    C     | |       variables in the new object
    |          | v
    |          | v <--- Direction of growth is "downwards"
    |          | v
    +==========+
    |          | <---- this part is "free" memory
    |          |       (Free means: unreserved !)
    |          |
    |          |
    +==========+
    |          | ^
    |          | ^ <--- Direction of growth is "upwards"
    |          | ^
    |    D     | |
    |          | | <--- This part grows when the program invokes
    |  System  | |      a function/method, reserve space for:
    |  stack   | |      (1) return address
    |          | |      (2) parameter variables
    |          | v      (1) local variables
    +----------+
```

  do **not** need to be ***defined*** (as an **array** as in **Java**)

  (The **memory** is **there** and it is ***reserved*** for us to make a **stack**)

- All we need to **implement** the **System Stack** is:

  - A **stacktop** index

- The **system stack** is **always** implemented by:

  - Using ***one*** of the **registers** in the **CPU** as **stacktop** index

  Example:

- In **M68000**, the **stacktop** index is:
  - The **address register A7**

- In **Intel**'s CPU (Pentium, Core Duo...), the **stacktop** index is stored in:
  - The **special purpose register (named) sp** (= stack pointer)

- So in **M68000**, the **system stack** is given **as follows:**

```
  Memory:

     0 +----------+
       |          |  <--- this part contains the "code"
       |    A     |       This part of the memory is "write protected"
       |          |       (can not - and should not - be modified)
       |          |
       +----------+
       |          | <--- this part contains "static variables"
       |    B     |      These variables exist at the start of the
       |          |      program (a.k.a.: compiled-time variables)
       +----------+
       |          | ^
       |          | | <---  this part grows when the program creates
       |          | |       new objects: reserve space for instance
       |    C     | |       variables in the new object
       |          | v
       |          | v <--- Direction of growth is "downwards"
       |          | v
       +==========+
       |          | <---- this part is "free" memory
       |          |       (Free means: unreserved !)
       |          |
       |          |
 A7 -->+==========+
       |          | ^
       |          | ^ <--- Direction of growth is "upwards"
       |          | ^
       |    D     | |
       |          | | <--- This part grows when the program invokes
       |  System  | |      a function/method, reserve space for:
       |  stack   | |      (1) return address
       |          | |      (2) parameter variables
       |          | v      (1) local variables
       +----------+
```

- **Pushing values onto the System Stack**
  - **Suppose** the **System Stack** is **as follows before** we **push** a new value:

```
         +----------+
         |          |
         +----------+
         |          |  (available, i.e., unused)
         +----------+
  A7 -->|     Z    |
         +----------+ ^
         |     Y    | ^ <--- Direction of growth is "upwards"
         +----------+ ^
         |          | |
         +----------+ |
         |          | |
         |  System  | |
         |  stack   | |
         |          | |
         |          | v
         +----------+
```

- The **Suppose** the **System Stack** will be **like this** ***after*** we **push** a new value ***X***:

```
         +----------+
         |          |
         +----------+
  A7 -->|     X    |  (Memory space no longer available)
         +----------+
         |     Z    |
         +----------+ ^
         |     Y    | ^ <--- Direction of growth is "upwards"
         +----------+ ^
         |          | |
         +----------+ |
         |          | |
         |  System  | |
         |  stack   | |
         |          | |
         |          | v
         +----------+
```

(The **memory space** is **no longer available** because **all memory spaces** "at and below" the **stack pointer A7** are ***used*** !!!!)

- The following **assembler instructions** will **push** a **integer (4 byte)** value ***X*** onto the **System Stack**:

```
suba.l    #4, A7          // Move system stacktop up 4 bytes
move.l    X, (A7)         // Store X in the top of the stack
```

- **De-allocating variables from the System Stack**
  - The following **assembler instructions** will **destroy (= unreserve memory))** a **integer (4 byte)** value from the **System Stack**:

```
adda.l    #4, A7             // Move system stacktop down 4 bytes
```

- **The system stack pointer**
  - **Stack pointer**:
    - The **stacktop index** (**A7**) is **more commonly known** as:
      - **(System) *Stack pointer***

    - It is **usually abbreviated** as: `sp` (for **stack pointer**)

- **Important functions of the system stack pointer A7**
  - **Function** of the **stack pointer A7**:
    - **A7** indicate the ***top*** of the **stack**; as such:
      - **A7** points to the **location** in the **stack** that the `push` and `pop` operations will **operate**

    - **Another function** of **A7** is:
      - **A7** ***marks*** the **memory locations** that are ***reserved*** **(= currently used !!!)**

      Specifically:

```
              +----------+
              |          |
              |          | AVAILABLE !!!! (not used)
              |          |
              +----------+
       A7 -->|          | ^
              |          | | USED !!!!
              |          | |
              |  System  | | (reserved !!!!)
              |  stack   | |
              |          | |
              |          | v
              +----------+
```

- **Therefore:**
  - When you perform a **push** operation (= **increase** the **stack size**), you will:
    - **create** one or more **variables** (on the **stack** !!!)
  - When you perform a **pop** operation (= **decrease** the **stack size**), you will:
    - **destroy** one or more **variables** (on the **stack** !!!)