TCL and OTCL programming

- **OTcl**
  - **Tcl and OTcl**

    - TCL is pronounced "Tickle"
    - TCL is a ***scripting*** **programming language** - similar to PERL or Python

      (Scripting languages are usually **interpreted** and have **dynamic typing**

      Very good for **quick and dirty jobs**)

    - **Object TCL (OTCL)** is an **object oriented extension** of **TCL**

      OTcl is written in **C++**
    - **OTCL's library of objects** can be **extended** by **adding object definitions** in the **C++ source code**

  - **NS2 and OTcl**

    - The **NS2 network simulator** consists mainly of **object extensions written in C++** that have been **added to OTcl**
    - **Built-in objects** include:

      - **Node** (is a **router**)
      - **Links** (output buffers)
      - **TCP Agents** (simulate all kinds of TCP protocols)
      - ...
    - **Simulator object**

      - A **special simulator object** is defined in **NS2** that perform **process-based simulation** on the **network objects**

    - To **write a network simulation program** in **NS2**, the programmer simply **write an OTcl program** that **creates** **network objects** and **one (special) simulator object**

  - **Presentation order:**

    - We will learn to **program in TCL** and
    - Then we will look at **OTcl** very **briefly**

    - how to **create objects**
    - how to **invoke methods in objects**
  - Lastly, learn how to use the **Network objects** defined in **NS2**

    (they are **OTcl objects** !!!)

- External resources to learn TCL:
  - Univ. of Chicago: click here

- **Variables**
  - **Defining variable:**
    - You **do not need to define** a **variable before you use it**
    - When a **new name** is encountered, **TCL** will assume that it is a **new variable**

  - **Name of a variable:**
    - **Variable names** in **Tcl** can contain **any characters** and be of **any length**.
    - To get **exotic characters (like whitespace)** into a **variable name** you may need to **quote it** (i.e., use **{....}**);

      Using exotic names (with **characters other than letters, digits and _**) are generally a **bad idea**.

    More on **variable names** when we learn to **use variables** below....

- **Variable Type**
  - All **variable** are **auto-typed** and **dynamically typed**

  - **Auto-typing:**
    - **Auto-typing** means that ***type*** **of the variable depends** on the ***value*** **stored in the variable**
  - **Dynamic typing:**
    - **Dynamically typed** means that **type of the variable** can **change over time**...

      (this can be confusing at times)

- **TCL statements**
  - A **TCL statement always** consists of:

    1. A **command name**
    2. **Optionally** followed by **one or more arguments (parameters)**

       - **unlike conventional programming language**, the **parameters** are ***not*** **given between brackets** !

    Example:

    ```
    set  x  "Hello World"
    ```

  - **Number of parameters:**

    - Most Tcl command has a **fixed number of arguments**
    - There are a few Tcl command (such as **expr**) that has a **variable number of arguments**
    - If you specify an **incorrect number of arguments** to a command, you will receive an **error** and the Tcl program will terminate.

Assignment Command/Statement

- **Assignent Statement: `set` command**
  - The `set` command is used to assign values to variables
  - The `set` command has **2 forms** , **depending on** the **number of arguments given**:

    ```
    set varName  value        # Assign value to variable varName

    set varName               # Return value in variable varName
    ```

  - **Examples:**

    ```
    set   b   1234       // Associate b with 1234

    set   b              // return 1234 (value associated with b)
    ```

  - **Note:**

- When using **"set b"**, an **error** occurs if there is **no variable named "b"**.

- **Obtaining the content of a variable**
  - You can **obtain the value** of a **variable** using the **set** command:

```
set  x  Hello    // assigned Hello to x

set  x           // Obtain the value of variable x
```

  - There is a **more common way** to **obtain the value** of a **variable:**
    - By using **Tcl's *substitution* operator "$"**
    - **Syntax:**

```
$variableName
```

      will **replace "variableName"** with the ***content*** **of that variable**

Examples:

```
set   b    56      // Asscoiate b with 56

puts  b            // Output: b

puts  [set b]      // Output: 56

puts  $b           // Output: 56
```

- **Variable names *revisited***
  - Unlike most conventional programming languages, you can use **any character** in a **variable name**.... **but**...
  - **However**, you would be **wise** to stick to using only the following set of characters:
    - **letters**
    - **digits**
    - **underscore**

That is because the **substitution operator ($)** **assumes** that **variables names consists only of these characters**.

- **Fact:**

  > - If a **variables name** contains **characters with *special meaning* (e.g., "-" means *subtract*)** then you **need** to use **braces {...}** to **"quote"** the name of the variable
  >
  > - This practice is a pretty ugly business....

- **Examples:**

```
set  foo-bar  1234     // Associate foo-bar with 1234

puts  $foo-bar         // Error
                       // Error message: "foo": no such variable
                       // (- character interpreted as minus operation)

puts  ${foo-bar}       // Output: 1234
```

- **The `unset` command**
  - The `unset` command **deletes the named variables** and **all storage** associated with them.
  - After an `unset`, any attempt to refer to the value of those variables results in an error.

- **Querying information: `info`**
  - The **info** command can be used to **obtain information** on various things

---

- **Check for the existance of a variable: The `info exists` command**
  - The `info exists varName` command returns:

    > - **1** if the **variable `varName` exists**
    > - **0** if it **does not exist**

- **List all variables: The `info vars` command**
  - The `info vars` command **returns** a **list of all defined variables** (in the current scope).

- **Arrays**

- Just like **simple variables**, you **don't have to define array variables** before **using them**.
- **Array variable syntax:**

```
variableName ( index )
```

Example:

```
set  A(2)  4567      # A(2) = 4567

puts $A(2)           # Print content of A(2)
```

- An **array variable** is **created** when you **first assign a value to it**.

- Find out the size of an array: **The `array size` command**
  - The `array size` command **returns number of elements** in the array
  - **Note:**

    - **Arrays are dynamic**, and the **array size can change** !

  - **Example:**

```
set  A(2)  4567      # A(2) = 4567
puts [array size A]  # Output: 1

set  A(6)  1111      # A(6) = 1111
puts [array size A]  # Output: 2
```

  - **Note:**

    - Each **array element** has an **independent from other array elements**; not like Java.

  - In Java, when you define an array:

```
int A[] = new int[10];
```

    You define the array elements **A[0], A[1], A[2], A[3], A[4], A[5], A[6], A[7], A[8], A[9]**

    In fact, if A[2] exist, we know that A[0] and A[1] must also exist !

  - In TCL, when you create array element A(2):

```
set  A(2)  4567      # A(2) = 4567
```

Only element **A(2) is created** and nothing else.

- **The *associative* array**
  - The **TCL's array** is in fact an ***associative* array**
  - **Arrays in *procedural* languages** (such as **Java, C, C++**, can only be **indexed by integers**

  - **Associative arrays:**

    - **Associateive arrays** are **arrays** than are **indexed** by **strings**
    - **Associative arrays** are also called **dictionaries**

  - **Examples:**

```
set  A(cs455)   Cheung          # A(cs355) = Cheung
set  A( cs455 ) Cheung          # Error: No spaces allowed in array name !!!
```

```
puts $A(cs455)                  # prints Cheung
set  s  cs455                   # s = cs455
puts $A($s)                     # prints Cheung !!!
```

  - The **reason** that **associative arrays** are called **dictionaries** is the fact that you can make a **dictionary** with it:

```
set dict[red]   "Color, associated with fire"
set dict[dead]  "Opposite of alive"
```

- **`Unsetting` an array**
  - You can **unset *individual* array element**:

    Example:

```
unset  A(smith)                 # unsets A(smith)
unset  A(jones)                 # unsets A(jones)
```

  - You can **unset** an **entire array (all elements)**:

    Example:

```
unset  A                # unsets all elements of A
```

- **Multi-dimensional arrays**
  - Tcl does not have multidimensional arrays...

    but associative arrays can simulate them easily:

```
set   A(1,1)   0
set   A(1,2)   0
set   A(1,3)   0
set   A(2,1)   0
set   A(2,2)   0
set   A(2,3)   0
```

  - **NOTE:**
    - The `array names` for the example above will return:

```
2,2   1,3   2,3   1,1   2,1   1,2
```

    So there is no need for nested loops to process multidimensional arrays in TCL
