Module 1: Introduction to Tcl and SCL

Welcome to the world of Surpac scripting! This first module will introduce you to the fundamental concepts of the Tcl language and the Surpac Command Language (SCL) extensions that allow you to automate your work.

1.1 What is Tcl?

Tcl, which stands for "Tool Command Language", is the scripting language that Surpac uses under the hood. You don't need to be a software developer to use it. Think of it as a set of simple, English-like commands that you can write in a text file to make Surpac perform a sequence of tasks automatically.

The most important concept in Tcl is that **everything is a command, and everything can be treated as a string (text)**. This makes the language syntax very simple and consistent.

1.2 What is SCL?

If Tcl is the language, then the **Surpac Command Language (SCL)** is the specialized vocabulary for that language. SCL is a library of custom commands that GEOVIA has created specifically to give you control over Surpac's functions and data from a Tcl script.

You can think of it like this:

- **Tcl:** Provides the basic grammar and structure (loops, variables, logic).
- **SCL:** Provides the powerful "action words" to interact with Surpac objects like string files, DTMs, block models, and the graphics viewport.

Most SCL commands begin with the prefix `Scl` (e.g., `SclSwaOpenFile`) or `Guido` for creating user interfaces.

1.3 Your First Script: "Hello, Surpac!"

The best way to learn is by doing. Let's write a simple script that displays a message in the Surpac message window. This is a fundamental way to give feedback to the user of your script.

Step 1: Create the Script File

Open a plain text editor (like Notepad, Notepad++, or VS Code) and type the following command into a new file:

```
 # My first Surpac script
# This line is a comment and will be ignored by the computer.

SclMessage "INFO" "Hello, Surpac! The script is running."
```

Step 2: Save the File

Save the file in your current working directory with the name `hello_surpac.tcl`. The `.tcl` extension is important as it tells Surpac that this is a Tcl script file.

Step 3: Run the Script in Surpac

Inside Surpac, find the menu option for running a script (often under a "Macros" or "Automation" menu) and select the `hello_surpac.tcl` file you just saved.

Step 4: See the Result

After running the script, you should see the following message appear in your Surpac message window:

> **Hello, Surpac! The script is running.**

Code Breakdown:

- `#` : The hash symbol indicates a comment. Surpac ignores these lines. They are for you to leave notes for yourself and others.
- `SclMessage` : This is the SCL command used to show a message. For more details, you can view its official documentation here: SclMessage Documentation.
- `"INFO"` : This is the first argument to the command. It tells Surpac what type of message it is (other options might include "ERROR" or "WARNING").
- `"Hello, Surpac! ..."` : This is the second argument, which is the actual text you want to display.

Congratulations! You have successfully written and executed your first SCL script. In the next module, we will dive deeper into the Tcl language fundamentals that form the building blocks of more complex and powerful scripts.

Module 2: Tcl Language Fundamentals

Now that you understand the relationship between Tcl and SCL, it's time to learn the basic building blocks of the Tcl language itself. Mastering these fundamentals is essential for writing scripts that are powerful, flexible, and easy to read.

2.1 Variables: Storing Information

Variables are containers for storing information. You can store numbers, text, or lists of data in a variable to use later in your script. The command to create a variable is `set`.

```
# Create a variable named 'pit_name' and store the text "St. Ives" in it
set pit_name "St. Ives"

# Create a variable named 'bench_height' and store the number 15 in it
set bench_height 15
```

To use the value stored in a variable, you must put a dollar sign (`$`) in front of its name. This tells Tcl to substitute the variable with its stored value.

```
# Use the SclMessage command to display the value of our variables
SclMessage "INFO" "Processing data for pit: $pit_name"
SclMessage "INFO" "The current bench height is: $bench_height metres"
```

2.2 Lists: The Most Important Data Structure

A list is a collection of items. In Surpac scripting, lists are used everywhere, especially for handling coordinates (X, Y, Z) and sets of data. You can create a list with the `list` command or by simply grouping items with spaces.

```
# Create a list of coordinates for a single point
set point_xyz [list 5500 2100 450.5]

# To get the Y coordinate (the second item), we use the lindex command
# Note: Lists are zero-indexed, so the first item is at index 0.
set y_coord [lindex $point_xyz 1]

SclMessage "INFO" "The Y coordinate is: $y_coord"
```

2.3 Expressions and Math

To perform mathematical calculations, you use the `expr` command. This is useful for everything from simple addition to complex engineering formulas.

```
set length 100.5
set width 50.2

# Calculate the area
set area [expr {$length * $width}]

SclMessage "INFO" "The calculated area is: $area"

# You can also use SCL's own expression evaluator, which can be more powerful
# See the documentation for more details: SclExpr
set calculated_value [SclExpr 10*sin(1.57)]
SclMessage "INFO" "SclExpr result: $calculated_value"
```

> **Important:** Always wrap your expression in curly braces `{}`. This ensures Tcl handles the calculation correctly, especially when your variables contain special characters.

2.4 Control Flow: Making Decisions

Control flow commands allow your script to make decisions and repeat actions, making it dynamic and intelligent.

`if / else` - Making a Choice

This lets you run code only if a certain condition is true.

```
set ore_grade 2.5

if {$ore_grade > 1.0} {
    SclMessage "INFO" "This is high-grade ore."
} else {
```

```
    SclMessage "INFO" "This is low-grade waste."
}
```

`foreach` - Looping Through Lists

This is perfect for performing an action on every item in a list, like processing multiple files or points.

```
 set string_files [list "pit1.str" "pit2.str" "pit3.str"]

foreach file $string_files {
    SclMessage "INFO" "Now processing file: $file"
    # In a real script, you would add SclSwaOpenFile here
}
```

2.5 Procedures (`proc`): Creating Your Own Commands

As your scripts get more complex, you'll find yourself writing the same lines of code over and over. A procedure (or "proc") lets you bundle a block of code into your own reusable command.

```
 # Define a procedure to calculate the volume of a rectangular block
proc calculate_volume {length width height} {
    set volume [expr {$length * $width * $height}]
    return $volume
}

# Now, call your new command just like any other Tcl command
set block_volume [calculate_volume 50 25 15]

SclMessage "INFO" "The volume of the block is: $block_volume cubic metres"
```

Procedures are the key to writing clean, organized, and maintainable scripts.

Next Steps: You now have a solid grasp of the basic Tcl commands. In Module 3, we will combine these concepts with SCL commands to start working with real Surpac files.

Module 3: Working with Surpac Files (SWA)

This module is where Tcl scripting becomes truly useful for automating Surpac tasks. You will learn how to programmatically open, read, and save the Surpac files (like `.str` and `.dtm` files) that you work with every day.

3.1 The SWA (Surpac Work Area)

Before you can work with a file, you must load it into Surpac's memory. This in-memory storage space is called the **Surpac Work Area (SWA)**.

Think of it like opening a file in the Surpac application itself. The file from your hard drive is loaded into an active session where you can view and manipulate it. SCL commands operate on these in-memory objects, not directly on the disk files. When you are finished, you save the in-memory version back to a file on the disk.

When you load a file into the SWA, Surpac gives you a unique name for it, called a **handle**. You will use this handle in all subsequent commands to tell Surpac which in-memory object you want to work with.

3.2 Opening and Saving Files

Opening a File

The command to open a file and load it into the SWA is `SclSwaOpenFile`. It takes the file path as an argument and returns the handle.

```
# The path to the file you want to open
set input_file "pit_design_v1.str"

# Open the file and store its handle in a variable
set file_handle [SclSwaOpenFile $input_file]

SclMessage "INFO" "Successfully loaded file $input_file with handle: $file_handle"
```

Saving a File

Once you have modified your data (which we'll cover in the next module), you can save it to a new file using `SclSwaSaveStringFile` or `SclSwaSaveDtmFile`.

```
# Assume file_handle contains the handle to a loaded string file
set output_file "pit_design_v2.str"

# Save the current state of the in-memory object to a new disk file
SclSwaSaveStringFile $file_handle $output_file

SclMessage "INFO" "Saved changes to $output_file"
```

3.3 Reading Data from Files

Surpac data has a hierarchy: a **file** contains **strings**, which contain **points**. You use SCL commands to navigate this hierarchy.

1. **Get a list of all strings:** Use `SclGetStrings` on the file handle to get a Tcl list of all string numbers.
2. **Get a specific string:** Use `SclGetItem` on the file handle to get a new handle for a specific string.
3. **Get a specific point:** Use `SclGetItem` again, this time on the string handle, to get a handle for a specific point.
4. **Get a coordinate value:** Use `SclGetValueByName` on a point handle to get its X, Y, Z, or other description field values.

> This may seem complicated, but it provides a very structured way to access any piece of data. The practical example below will make it clear.

3.4 Practical Example: Find Min/Max Z Value

This script will read a string file and find the highest and lowest Z-values among all its points. This is a common quality control check.

Create a file named `find_extents.tcl` with the following code. To run it, you will need a string file named `test_topo.str` in the same directory.

```
# Practical Example: Find Min/Max Z of a String File

# 1. Set up the input file name
set input_file "test_topo.str"

# Check if the file exists before trying to open it
if {![file exists $input_file]} {
    SclMessage "ERROR" "Input file not found: $input_file"
    return
}

# 2. Open the file and get its handle
set file_handle [SclSwaOpenFile $input_file]
```

```
# 3. Initialize min/max variables to be overwritten
set min_z 99999
set max_z -99999
set first_point_found 0

# 4. Get a list of all strings in the file
set string_list [SclGetStrings $file_handle]

# 5. Loop through each string
foreach string_id $string_list {
    # Get the handle for the current string
    set string_handle [SclGetItem $file_handle "string" $string_id]

    # Get the number of points in this string
    set point_count [SclGetItem $string_handle "count"]

    # 6. Loop through each point in the string
    for {set i 1} {$i <= $point_count} {incr i} {
        # Get the handle for the current point
        set point_handle [SclGetItem $string_handle "point" $i]

        # 7. Get the Z value of the point
        set current_z [SclGetValueByName $point_handle "z"]

        # 8. On the very first point, set min/max to its Z value
        if {$first_point_found == 0} {
            set min_z $current_z
            set max_z $current_z
            set first_point_found 1
        }

        # 9. Compare and update min/max Z
        if {$current_z < $min_z} {
            set min_z $current_z
        }
        if {$current_z > $max_z} {
            set max_z $current_z
        }
    }
}

# 10. Report the results
SclMessage "INFO" "Scan complete for file: $input_file"
SclMessage "INFO" "Minimum Z value found: $min_z"
SclMessage "INFO" "Maximum Z value found: $max_z"

# 11. Clean up the memory by destroying the handle
SclDestroyHandle $file_handle
```

Next Steps: You can now read and analyze data from Surpac files. In Module 4, you will learn how to create new geometry and modify existing data, giving you the power to create and transform data automatically.

Module 4: Creating and Modifying Geometry

In the previous module, you learned how to read and analyze data from existing Surpac files. Now, you will learn how to create new geometry from scratch and modify existing data, giving you the power to generate new designs, automate data cleaning, and create custom tools.

4.1 Creating New Geometry

Creating new geometry in SCL follows a logical, hierarchical process. You start by creating an empty container (a SWA object), then you create a string to put inside it, and finally you create points to add to the string.

1. **`SclCreateSwa`** : Creates a new, empty SWA object in memory and returns a handle to it. This is your blank canvas.
2. **`SclCreateString`** : Creates a new, empty string object with a specified string number.
3. **`SclCreatePoint`** : Creates a new point object with specified X, Y, and Z coordinates.
4. **`SclAdd`** : This crucial command adds one object to another. For example, you use it to add a point object to a string object, and then add that string object to the main SWA file object.

This process gives you precise control over the structure of the file you are building.

4.2 Modifying Existing Geometry

Modifying an existing object is often simpler than creating a new one. The most common modification is changing the coordinates or description fields of a point. The primary command for this is `SclSetValueByName` .

To use it, you first need to get the handle to the specific point you want to change (as you learned in Module 3), and then you can set its `x`, `y`, `z`, or description field values.

```
 # Assume point_handle is a handle to a specific point

# Get the current Y coordinate
set current_y [SclGetValueByName $point_handle "y"]

# Calculate the new Y coordinate
set new_y [expr {$current_y + 10.0}]

# Set the point's Y coordinate to the new value
SclSetValueByName $point_handle "y" $new_y

SclMessage "INFO" "Point Y coordinate updated from $current_y to $new_y"
```

4.3 Practical Example: Offset a String File

This script combines reading, creating, and modifying concepts. It will read an existing string file, create a brand new in-memory copy, and offset every point in the copy by a user-defined amount before saving it to a new file.

Create a file named `offset_string.tcl` . You will also need an input file, for example, the `test_topo.str` file from the previous module.

```
 # Practical Example: Offset all points in a string file

# 1. Define input/output files and the offset distance
set input_file "pit_design_v1.str"
set output_file "pit_design_v2.str"
set y_offset 10.0

# 2. Open the source file for reading
set in_file_handle [SclSwaOpenFile $input_file]

# 3. Create a new, empty SWA object for our output
set out_file_handle [SclCreateSwa]

# 4. Get all string numbers from the source file
set string_list [SclGetStrings $in_file_handle]

# 5. Loop through each source string
foreach string_id $string_list {
    set in_string_handle [SclGetItem $in_file_handle "string" $string_id]

    # Create a corresponding new, empty string in our output SWA
    set out_string_handle [SclCreateString $string_id]

    set point_count [SclGetItem $in_string_handle "count"]

    # 6. Loop through each point in the source string
    for {set i 1} {$i <= $point_count} {incr i} {
```

```
        set in_point_handle [SclGetItem $in_string_handle "point" $i]

        # 7. Read the original coordinates
        set x [SclGetValueByName $in_point_handle "x"]
        set y [SclGetValueByName $in_point_handle "y"]
        set z [SclGetValueByName $in_point_handle "z"]

        # 8. Calculate the new Y coordinate
        set new_y [expr {$y + $y_offset}]

        # 9. Create a new point with the new coordinates
        set out_point_handle [SclCreatePoint [list $x $new_y $z]]

        # 10. Add the new point to our new string
        SclAdd $out_string_handle $out_point_handle
    }

    # 11. Add the completed new string to our new SWA file
    SclAdd $out_file_handle $out_string_handle
}

# 12. Save the new SWA object to a disk file
SclSwaSaveStringFile $out_file_handle $output_file

SclMessage "INFO" "Successfully created offset file: $output_file"

# 13. Clean up memory
SclDestroyHandle $in_file_handle
SclDestroyHandle $out_file_handle
```

Next Steps: You are now able to perform powerful data manipulation. In Module 5, we will move from processing raw data to visualizing it by drawing markers, lines, and text directly in the Surpac graphics viewport.

Module 5: Graphics and Visualization

While processing data is powerful, visualizing the results is often essential. SCL provides a rich set of commands to draw temporary graphics directly into the Surpac viewport. This allows you to highlight specific points, label areas of interest, or create custom visual feedback for your scripts.

5.1 The Graphics Context: Your Drawing Canvas

Before you can draw anything, you must first open a graphics "node" or "context". Think of this as creating a transparent layer over your main viewport where your temporary graphics will live. You open it with `SclGraphicsOpen` and, crucially, you must close it with `SclGraphicsClose` when you are finished.

```
# Open a graphics node named 'my_results'
SclGraphicsOpen "my_results"

# ... all your drawing commands go here ...

# Close the node to finalize the drawing
SclGraphicsClose
```

> If you don't close the graphics node, your drawings might not appear correctly. Always pair an `Open` with a `Close`.

5.2 Setting Visual Styles

Before you draw an object, you can set its appearance. These settings will apply to all subsequent drawing commands until you change them again.

- **`SclGraphicsSetColour`** : Sets the color for lines, markers, or text. (e.g., `"lines=red"`, `"markers=blue"`).
- **`SclGraphicsSetLineWeight`** : Sets the thickness of lines.
- **`SclGraphicsSetMarkerSymbol`** : Sets the shape of markers (e.g., "dot", "cross", "diamond").
- **`SclGraphicsSetMarkerSize`** : **Sets the size of the markers.**

5.3 Drawing in the Viewport

Once your styles are set, you can use the insertion commands to draw objects at specific coordinates.

- **`SclGraphicsInsertMarker`** : **Places a single marker at a given XYZ coordinate.**
- **`SclGraphicsInsertPolyline`** : **Draws a line connecting a list of XYZ coordinates.**
- **`SclGraphicsInsertText`** : **Draws a text label at a given XYZ coordinate.**

Finally, after closing the graphics node, you must call `SclGraphicsUpdateDisplay` to make your changes visible in the viewport.

5.4 Practical Example: Highlight Steep Triangles

This script will read a DTM file, check the dip of each triangle, and draw a large red cross on the center of any triangle that is steeper than 60 degrees.

Create a file named `highlight_steep.tcl`. You will need a DTM file (e.g., `pit_surface.dtm`) in the same directory to run it.

```
# Practical Example: Highlight steep triangles in a DTM

# 1. Setup
set input_dtm "pit_surface.dtm"
set dip_threshold 60.0

# 2. Open the DTM file
set file_handle [SclSwaOpenFile $input_dtm]

# 3. Open a graphics node for our results
SclGraphicsOpen "steep_triangles_check"

# 4. Set the desired marker style
SclGraphicsSetColour "markers=red"
SclGraphicsSetMarkerSymbol "cross"
SclGraphicsSetMarkerSize 5

# 5. Get all triobjects (usually just one in a DTM)
set triobject_list [SclGetTriobjects $file_handle]

foreach triobject_handle $triobject_list {
    set triangle_count [SclGetItem $triobject_handle "count"]

    # 6. Loop through every triangle
    for {set i 1} {$i <= $point_count} {incr i} {
```

```
        set triangle_handle [SclGetItem $triobject_handle "triangle" $i]

        # 7. Get the dip of the triangle
        set dip [SclGetValueByName $triangle_handle "dip"]

        # 8. Check if the dip exceeds our threshold
        if {$dip > $dip_threshold} {
            # 9. If it's steep, get its three corner points
            set p1_handle [SclGetValueByName $triangle_handle "p1"]
            set p2_handle [SclGetValueByName $triangle_handle "p2"]
            set p3_handle [SclGetValueByName $triangle_handle "p3"]

            # 10. Calculate the centroid (average XYZ) of the triangle
            set x1 [SclGetValueByName $p1_handle "x"]
            set y1 [SclGetValueByName $p1_handle "y"]
            set z1 [SclGetValueByName $p1_handle "z"]
            set x2 [SclGetValueByName $p2_handle "x"]
            set y2 [SclGetValueByName $p2_handle "y"]
            set z2 [SclGetValueByName $p2_handle "z"]
            set x3 [SclGetValueByName $p3_handle "x"]
            set y3 [SclGetValueByName $p3_handle "y"]
            set z3 [SclGetValueByName $p3_handle "z"]

            set centroid_x [expr {($x1 + $x2 + $x3) / 3.0}]
            set centroid_y [expr {($y1 + $y2 + $y3) / 3.0}]
            set centroid_z [expr {($z1 + $z2 + $z3) / 3.0}]

            # 11. Insert a marker at the centroid
            SclGraphicsInsertMarker [list $centroid_x $centroid_y $centroid_z]
        }
    }
}

# 12. Close the graphics node and update the display
SclGraphicsClose
SclGraphicsUpdateDisplay

SclMessage "INFO" "Steep triangle analysis complete."

# 13. Clean up memory
SclDestroyHandle $file_handle
```

Next Steps: You can now visualize your script's results. In Module 6, we will tackle one of the most powerful features of SCL: reading from and writing to Surpac Block Models.

Module 6: Interacting with Block Models

Automating block model tasks is one of the most powerful applications of SCL scripting. You can perform calculations, reclassify materials, or run complex validation checks on millions of blocks in a fraction of the time it would take manually. This module will teach you how to read from and write to the attributes of a block model.

> **Prerequisite:** Before running any block model script, you must have a block model loaded and active in your Surpac session.

6.1 Accessing the Current Block Model

Unlike string or DTM files, you don't open a block model from a script. Instead, you get a handle to the model that is already loaded in Surpac. The command for this is `SclBlockModelGetCurrentModel`.

```
# Get the handle for the currently active block model
set bm_handle [SclBlockModelGetCurrentModel]

if {$bm_handle == ""} {
    SclMessage "ERROR" "No block model is currently loaded."
    return
} else {
    SclMessage "INFO" "Successfully connected to the active block model."
}
```

6.2 Reading Block Model Data: The Iteration Loop

To process a block model, you must loop through it block by block. SCL provides a specific three-part structure for this loop:

1. `SclBlockModelIterateStart`: This initializes the loop. You provide it with a list of the attributes you want to access.
2. `SclBlockModelIterateNext`: This command moves to the next block in the model. It returns `1` (true) if it successfully moved to a new block, and `0` (false) when it has reached the end of the model. We use this in a `while` loop.
3. `SclBlockModelIterateEnd`: This cleans up and closes the loop. It is essential to call this at the end.

Inside the loop, you use `SclBlockModelGetValues` to retrieve the values of the attributes for the current block.

6.3 Writing to a Block Model

Writing data is done inside the same iteration loop. After you have calculated a new value, you use the command `SclBlockModelSetValues`. You provide it with a list of attribute names and a corresponding list of the new values you want to write.

6.4 Practical Example: Simple Ore Classification

This script will iterate through a block model. If a block's gold grade (`au_grade`) is above a cutoff, it will classify it as ore by setting a new attribute (`ore_class`) to 1. Otherwise, it will set `ore_class` to 2 (for waste).

Create a file named `classify_ore.tcl`. Before running, ensure you have a block model loaded with attributes named `au_grade` and `ore_class`.

```
# Practical Example: Classify ore based on grade

# 1. Setup
set grade_cutoff 0.8
set attributes_to_access [list "au_grade" "ore_class"]

# 2. Get the active block model
set bm_handle [SclBlockModelGetCurrentModel]
if {$bm_handle == ""} {
    SclMessage "ERROR" "No block model loaded."
    return
}

# 3. Start the iteration
SclBlockModelIterateStart $bm_handle $attributes_to_access

SclMessage "INFO" "Starting block model classification..."

# 4. Loop while there are more blocks
while {[SclBlockModelIterateNext $bm_handle]} {
    # 5. Get the values for the current block
    set values [SclBlockModelGetValues $bm_handle $attributes_to_access]

    # Extract the grade from the list of values (it's the first item)
    set current_grade [lindex $values 0]
```

```
    # 6. Make the classification decision
    set new_ore_class 0
    if {$current_grade >= $grade_cutoff} {
        # This is ore
        set new_ore_class 1
    } else {
        # This is waste
        set new_ore_class 2
    }

    # 7. Set the value for the 'ore_class' attribute
    SclBlockModelSetValues $bm_handle [list "ore_class"] [list $new_ore_class]
}

# 8. End the iteration - this is very important!
SclBlockModelIterateEnd $bm_handle

SclMessage "INFO" "Block model classification complete."
```

Next Steps: You have now mastered the core data processing skills for files and block models. In Module 7, we will learn how to make your scripts more user-friendly by building simple graphical user interfaces (GUIs) with the GUIDO toolkit.

Module 7: Building a User Interface (GUIDO)

While scripts are powerful, they can be hard to use if you have to edit the code every time you want to change an input file or a parameter. The GUIDO toolkit solves this by allowing you to create simple, professional-looking forms for your scripts, all using Tcl commands.

7.1 Introduction to GUIDO

GUIDO works by defining UI elements as objects. You create a main form, add widgets (like buttons and text boxes) to it, and then tell Surpac to display it. The command to display a form is `SclRun` .

The structure of a GUIDO script is typically:

1. **Define the main form.**
2. **Define all the widgets (buttons, fields, etc.) and add them to the form.**
3. **Define the actions (callbacks) that happen when a user interacts with a widget (e.g., clicks a button).**
4. **Use `SclRun` to show the form to the user.**

7.2 Core UI Elements

Here are some of the most common widgets you will use:

- **`GuidoForm` : The main window that contains all other widgets.**
- **`GuidoFileBrowserField` : A text box with a "Browse..." button, perfect for selecting input and output files.**
- **`GuidoField` : A simple text box for user input, like numbers or text.**
- **`GuidoButton` : A clickable button that can trigger an action.**

7.3 Handling User Actions with Callbacks

A UI is useless unless it does something when you interact with it. In GUIDO, you use the `-action` option on a widget to specify a callback. A callback is simply the name of a Tcl procedure that you want to run when the action occurs.

For example, you can link an "OK" button to a procedure named `do_processing` . When the user clicks "OK", your `do_processing` procedure will automatically run.

7.4 Practical Example: UI for the Offset Script

Let's take the offset script from Module 4 and build a user-friendly interface for it. This script will present a form where the user can browse for the input and output files and enter the desired offset.

Create a file named `offset_string_ui.tcl` .

```
 # Practical Example: A GUIDO UI for the string offset tool

# ------------------------------------------------------------------------------
#  PROCEDURE: Main processing logic
#  This is the callback that runs when the user clicks "OK".
# ------------------------------------------------------------------------------
proc do_processing {} {
    # Get the values from the UI fields
    set input_file [$::guido(input_file) get]
    set output_file [$::guido(output_file) get]
    set y_offset [$::guido(y_offset) get]

    # --- This is the same logic from Module 4 ---
    set in_file_handle [SclSwaOpenFile $input_file]
    set out_file_handle [SclCreateSwa]
    set string_list [SclGetStrings $in_file_handle]

    foreach string_id $string_list {
        set in_string_handle [SclGetItem $in_file_handle "string" $string_id]
        set out_string_handle [SclCreateString $string_id]
        set point_count [SclGetItem $in_string_handle "count"]

        for {set i 1} {$i <= $point_count} {incr i} {
            set in_point_handle [SclGetItem $in_string_handle "point" $i]
            set x [SclGetValueByName $in_point_handle "x"]
            set y [SclGetValueByName $in_point_handle "y"]
            set z [SclGetValueByName $in_point_handle "z"]
            set new_y [expr {$y + $y_offset}]
            set out_point_handle [SclCreatePoint [list $x $new_y $z]]
            SclAdd $out_string_handle $out_point_handle
```

```
        }
        SclAdd $out_file_handle $out_string_handle
    }

    SclSwaSaveStringFile $out_file_handle $output_file
    SclMessage "INFO" "Successfully created offset file: $output_file"
    SclDestroyHandle $in_file_handle
    SclDestroyHandle $out_file_handle
}

# ------------------------------------------------------------------------
#  UI DEFINITION
#  Here we define the form and all its widgets.
# ------------------------------------------------------------------------

# 1. Create the main form
GuidoForm $::guido(form) -title "Offset String File"

# 2. Create the widgets
GuidoFileBrowserField $::guido(input_file) -parent $::guido(form) -label "Input String File" -type "open" -filetypes {{Surpac
Files} {.str .dtm}}
GuidoFileBrowserField $::guido(output_file) -parent $::guido(form) -label "Output String File" -type "save" -filetypes
{{Surpac Files} {.str .dtm}}
GuidoField $::guido(y_offset) -parent $::guido(form) -label "Y Offset" -value 10.0

GuidoPanel $::guido(button_panel) -parent $::guido(form) -layout "horizontal"
GuidoButton $::guido(ok_button) -parent $::guido(button_panel) -label "OK" -action { do_processing; $::guido(form) close }
GuidoButton $::guido(cancel_button) -parent $::guido(button_panel) -label "Cancel" -action { $::guido(form) close }

# 3. Run the form
SclRun $::guido(form)
```

> The `$::guido()` syntax is a special Tcl convention for naming GUIDO widgets so they can be easily accessed from anywhere in the script.

Next Steps: Congratulations on building your first UI! You now have all the skills necessary to build powerful, user-friendly tools. In the final module, we will combine everything you have learned to create a complete, practical tool for a real-world mining task.

Module 8: Capstone Project - Pit Crest Compliance Checker

Welcome to the final module! This project will combine all the skills you have developed to create a complete, practical, and user-friendly tool. You will build a script that checks if a pit crest design lies within a specified lease boundary, providing both visual and text-based feedback.

Project Objective

To create a tool with a user interface that allows a user to:

1. **Select a string file representing a pit crest design.**
2. **Select a string file representing a closed lease boundary polygon.**
3. **Run a check to see which points of the pit crest fall outside the lease boundary.**
4. **Display the non-compliant points visually in the viewport with large red markers.**
5. **Provide a summary report in the message window.**

Key SCL Commands Used

This project heavily relies on one new, powerful command:

- `SclInside` **: This command determines if a given XY coordinate is inside a closed 2D polygon. It is the core of our compliance check.**

It also uses everything we have learned so far: `GuidoForm` **for the UI,** `SclSwaOpenFile` **for file access, loops for processing, and** `SclGraphicsInsertMarker` **for visualization.**

The Final Script

Create a file named `compliance_checker.tcl` **and paste the following code into it. The code is fully commented to explain each step of the process.**

```
# ------------------------------------------------------------------------------
#  Capstone Project: Pit Crest Compliance Checker
#  This script checks which points of a crest string fall outside a boundary.
# ------------------------------------------------------------------------------

# ------------------------------------------------------------------------------
#  PROCEDURE: run_compliance_check
#  This is the main logic, executed when the user clicks "OK".
# ------------------------------------------------------------------------------
proc run_compliance_check {} {
    # --- Module 7: Get values from the GUIDO form ---
    set crest_file [$::guido(crest_file) get]
    set boundary_file [$::guido(boundary_file) get]

    # --- Module 2: Basic validation ---
    if {![file exists $crest_file] || ![file exists $boundary_file]} {
        SclMessage "ERROR" "One or both input files could not be found."
        return
    }

    # --- Module 3: Open files ---
    set crest_handle [SclSwaOpenFile $crest_file]
    set boundary_handle [SclSwaOpenFile $boundary_file]

    # Assume the boundary is the first string in the file
    set boundary_string_handle [SclGetItem $boundary_handle "string" 1]

    # --- Module 5: Setup graphics ---
    SclGraphicsOpen "compliance_check_results"
    SclGraphicsSetColour "markers=red"
    SclGraphicsSetMarkerSymbol "cross"
    SclGraphicsSetMarkerSize 8

    set non_compliant_count 0
    SclMessage "INFO" "Starting compliance check..."

    # --- Module 3 & 4: Loop through all geometry ---
    set string_list [SclGetStrings $crest_handle]
    foreach string_id $string_list {
        set string_handle [SclGetItem $crest_handle "string" $string_id]
        set point_count [SclGetItem $string_handle "count"]

        for {set i 1} {$i <= $point_count} {incr i} {
            set point_handle [SclGetItem $string_handle "point" $i]
```

```
            set x [SclGetValueByName $point_handle "x"]
            set y [SclGetValueByName $point_handle "y"]
            set z [SclGetValueByName $point_handle "z"]

            # --- Core Logic: Check if the point is inside the boundary ---
            set is_inside [SclInside $boundary_string_handle [list $x $y]]

            if {$is_inside == 0} {
                # This point is OUTSIDE the boundary
                incr non_compliant_count

                # --- Module 5: Draw a marker ---
                SclGraphicsInsertMarker [list $x $y $z]
                SclMessage "WARNING" "Point $i in string $string_id is outside the boundary."
            }
        }
    }

    # --- Finalize and Report ---
    SclGraphicsClose
    SclGraphicsUpdateDisplay

    SclMessage "INFO" "-------------------------------------------------"
    SclMessage "INFO" "Compliance Check Complete."
    SclMessage "INFO" "Found $non_compliant_count non-compliant points."
    SclMessage "INFO" "-------------------------------------------------"

    # --- Module 3: Clean up memory ---
    SclDestroyHandle $crest_handle
    SclDestroyHandle $boundary_handle
}

# ------------------------------------------------------------------------
#  UI DEFINITION (Module 7)
# ------------------------------------------------------------------------
GuidoForm $::guido(form) -title "Pit Crest Compliance Checker"

GuidoFileBrowserField $::guido(crest_file) -parent $::guido(form) -label "Pit Crest Design File" -type "open" -filetypes
{{String Files} {.str}}
GuidoFileBrowserField $::guido(boundary_file) -parent $::guido(form) -label "Lease Boundary File" -type "open" -filetypes
{{String Files} {.str}}

GuidoPanel $::guido(button_panel) -parent $::guido(form) -layout "horizontal"
GuidoButton $::guido(ok_button) -parent $::guido(button_panel) -label "Run Check" -action { run_compliance_check;
$::guido(form) close }
GuidoButton $::guido(cancel_button) -parent $::guido(button_panel) -label "Cancel" -action { $::guido(form) close }

# Run the form
SclRun $::guido(form)
```

Tutorial Complete!

Congratulations! You have completed the Surpac Tcl/SCL Scripting Tutorial. You have progressed from basic commands to building a complete, user-friendly tool that solves a practical problem.

You now have the foundational skills to explore the SCL documentation on your own, automate your repetitive tasks, and build custom tools to enhance your productivity and extend the power of Surpac.

Module 9: Integrating Python with Tcl/SCL

Surpac's integration with Python opens up a vast new world of possibilities for your scripting. Python's extensive libraries for data analysis, scientific computing, machine learning, and web interaction can now be leveraged directly from your Tcl/SCL scripts, allowing you to tackle complex problems that might be difficult or impossible with Tcl/SCL alone.

9.1 The `SclPythonExecutor` Command

The bridge between Tcl/SCL and Python is the `SclPythonExecutor` command. This command allows you to execute Python code directly from your Tcl script. It has two main variants:

Variant 1: Executing an External Python File (`file`)

This is the recommended approach for more complex or reusable Python scripts. You provide the path to a `.py` file, and you can optionally pass arguments to it.

```
 # Tcl Syntax:
SclPythonExecutor file <python_file_path> [<arg1> <arg2> ...]

# Example:
set python_script "my_analysis.py"
set input_param "some_value"
set output_path "results.csv"
SclPythonExecutor file $python_script $input_param $output_path
```

In your Python script: Arguments are received via the standard `sys.argv` list. Remember that `sys.argv[0]` is the script name itself, and subsequent elements are the arguments passed from Tcl.

```
 # Python (my_analysis.py):
import sys

if __name__ == "__main__":
    # sys.argv[0] is script name, arguments start from index 1
    input_param = sys.argv[1] if len(sys.argv) > 1 else "default"
    output_path = sys.argv[2] if len(sys.argv) > 2 else "default_results.csv"

    print(f"Python received input: {input_param}")
    # ... perform complex analysis ...
    with open(output_path, "w") as f:
        f.write(f"Processed data for {input_param}\n")
        f.write("Result,123.45\n")
    print(f"Python wrote results to {output_path}")
```

Variant 2: Executing a Python Code String (`script`)

This variant is useful for executing short, dynamic snippets of Python code directly within your Tcl script. The Python code is provided as a string.

```
 # Tcl Syntax:
SclPythonExecutor script "<python_code_string>"

# Example:
set tcl_variable "Hello from Tcl!"
set python_code "print(f'Python says: {tcl_variable}')"
SclPythonExecutor script $python_code

# For multi-line Python code, use Tcl's curly braces for the string:
set multi_line_python {
import math
radius = 10
area = math.pi * radius**2
print(f"Calculated area: {area:.2f}")
}
SclPythonExecutor script $multi_line_python
```

> **Important:** When embedding Tcl variables into the Python code string, be extremely careful with quoting. Using Tcl's curly braces `{}` for multi-line Python strings is generally safer as it prevents Tcl from performing substitutions prematurely.

9.2 Communicating Between Tcl and Python

This is a critical aspect of integration. Surpac's Tcl environment (Tcl 8.5) does not have direct support for complex data structures like JSON. Therefore, the recommended and most robust method for exchanging structured data is via **temporary files, typically CSV (Comma Separated Values)**.

- **Tcl to Python:** Pass data as command-line arguments (for the `file` variant) or embed simple values directly into the Python script string (for the `script` variant). For larger datasets, Tcl can write data to a temporary CSV file, and Python can read it.
- **Python to Tcl:** Python's `print()` statements will output directly to the Surpac message window. For structured results, Python should write data to a temporary CSV file, which your Tcl script can then read and parse.

> The `SclPythonExecutor` command is **synchronous**. This means your Tcl script will pause and wait for the Python script to complete before continuing. This simplifies error handling and file management.

9.3 Practical Example: Python-Enhanced Data Processing

This example demonstrates a common workflow: Tcl prepares some data, calls a Python script to perform a calculation, and then reads the results back from a CSV file generated by Python.

First, create a Python script named `calculate_average.py` in the same directory as your Tcl script:

```
 # calculate_average.py
import sys
import csv

# Get input data and output file path from command-line arguments
input_numbers_str = sys.argv[1] if len(sys.argv) > 1 else ""
output_csv_path = sys.argv[2] if len(sys.argv) > 2 else "python_output.csv"

# Convert input string of numbers to a list of floats
try:
    numbers = [float(x) for x in input_numbers_str.split(',') if x.strip()]
except ValueError:
    print("Error: Invalid numbers provided.", file=sys.stderr)
    sys.exit(1)

# Perform calculation
if numbers:
    average = sum(numbers) / len(numbers)
else:
    average = 0.0

# Write results to CSV
with open(output_csv_path, 'w', newline='') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(["Input_Count", "Average_Value"])
    writer.writerow([len(numbers), average])

print(f"Python: Calculated average {average:.2f} from {len(numbers)} numbers.")
```

Now, create a Tcl script named `tcl_python_integration.tcl`:

```
 # tcl_python_integration.tcl

# 1. Define input data for Python
set data_for_python "10.5,20.0,15.3,22.1,18.7"

# 2. Define paths
set python_script_path "calculate_average.py"
set output_csv_path "python_results.csv"

# 3. Clean up previous output file if it exists
if {[file exists $output_csv_path]} {
    file delete $output_csv_path
}

SclMessage "INFO" "Tcl: Preparing to call Python script..."

# 4. Execute the Python script
# Tcl will pause here until Python finishes
if {[catch { SclPythonExecutor file $python_script_path $data_for_python $output_csv_path } result]} {
    SclMessage "ERROR" "Tcl: Failed to execute Python script. Error: $result"
    return
} else {
    SclMessage "INFO" "Tcl: Python script execution completed."
}

# 5. Read results from the CSV file generated by Python
if {[file exists $output_csv_path]} {
```

```
    set fh [open $output_csv_path r]
    set header [gets $fh) ; # Read header line
    set data_line [gets $fh] ; # Read data line
    close $fh

    # Parse the data line
    set columns [split $data_line ","]
    set input_count [lindex $columns 0]
    set average_value [lindex $columns 1]

    SclMessage "INFO" "Tcl: Python reported: $input_count numbers with an average of $average_value."

    # Optional: Delete the temporary CSV file
    file delete $output_csv_path
} else {
    SclMessage "WARNING" "Tcl: Python script completed, but output CSV not found."
}

SclMessage "INFO" "Tcl: Script finished."
```

Conclusion: You now have the knowledge to integrate Python's vast capabilities into your Surpac workflows. This allows for highly specialized data processing and analysis, extending the power of your automation scripts significantly.

Module 10: Advanced Script Control and Surpac Functions

Beyond simply executing commands, Tcl/SCL allows you to exert fine-grained control over script execution flow and directly interact with Surpac's internal functions. This module explores commands that enhance user interaction, manage script pauses, and call any Surpac menu function.

10.1 Controlling Script Flow: `SclPause`

The `SclPause` command is used to introduce a delay or pause script execution, either for a specified duration or until user interaction occurs.

- `SclPause <seconds>` : Pauses the script for the specified number of seconds.
- `SclPause` **(no arguments):** Pauses the script indefinitely until the user clicks in the graphics window. This is useful for allowing the user to inspect the display before the script continues.

```
# Pause for 3 seconds
SclMessage "INFO" "Script pausing for 3 seconds..."
SclPause 3
SclMessage "INFO" "Script resumed."

# Pause until user clicks in graphics window
SclMessage "INFO" "Click in the graphics window to continue..."
SclPause
SclMessage "INFO" "User clicked. Script resumed."
```

10.2 Form Behavior: `_action` and `_error` Clauses

When defining GUIDO forms, you can use special clauses to control how the form behaves upon display or if an error occurs. These are typically used with `SclFunction` or `SclRun`.

- `_action="display"` : This is the default behavior for forms. It displays the form and waits for user input (e.g., clicking OK or Cancel).
- `_action="apply"` : This automatically applies the form without displaying it to the user. This is useful if you want to programmatically set values and immediately apply them without user intervention.
- `_error="abort"` : If an error is encountered within the form (e.g., invalid input), the script will stop execution.
- `_error="display"` : If an error is encountered, the form will be displayed, allowing the user to correct the input. This is generally preferred for user-facing scripts.

```
# Example of _action="apply" (assuming a form named my_form is defined elsewhere)
# This would set values and apply them without showing the form
# SclFunction "my_form" "_action=apply" "field1=value1" "field2=value2"

# Example of _error="display" (common for user input forms)
# GuidoForm $::guido(my_form) -title "My Form" -error "display"
# ... add fields ...
# SclRun $::guido(my_form)
```

10.3 Calling Surpac Functions: `SclFunction`

The `SclFunction` command is incredibly versatile. It allows you to call almost any function available in Surpac's menu system directly from your Tcl script. This is how you automate complex Surpac operations that don't have a direct SCL command.

The syntax typically involves the function name (as it appears in Surpac's command line or macro recorder) followed by any required parameters.

```
# Example: Recalling a string file using SclFunction
# This is equivalent to File -> Recall -> String
SclFunction "RECALL STRING" "file=my_data.str"

# Example: Running a triangulation function
# SclFunction "TRIANGULATE DTM" "input_dtm=surface.dtm" "output_dtm=new_surface.dtm"
```

> To find the exact function name and parameters for `SclFunction`, use Surpac's macro recorder. Record yourself performing the desired action, then open the generated macro file (usually a `.tcl` file) in a text editor. The recorded commands will show you the correct syntax.

10.4 Integrating Scripts into Surpac Menus and Toolbars

Once your scripts are robust and user-friendly, you can integrate them directly into the Surpac interface. This makes them easily accessible to users without requiring them to manually run the script file.

The process typically involves:

1. **Customizing Menus/Toolbars:** In Surpac, go to `Customise -> Customise menus/toolbars`.
2. **Creating New Items:** Create a new menu item or toolbar button.

3. **Linking to Script:** Point the new item to your `.tcl` script file. You can also specify an icon for toolbar buttons.
4. **Saving Profile:** Save your customized profile so the changes persist.

This allows you to create a tailored Surpac environment with your custom automation tools readily available.

Next Steps: You now have powerful tools for controlling script execution and integrating with Surpac's core functions. In Module 11, we will explore Tcl's advanced data structures, specifically arrays, which are crucial for managing complex, non-sequential data.

Module 11: Tcl Arrays and Advanced Data Structures

While Tcl lists are incredibly versatile, sometimes you need a way to store and retrieve data using descriptive names rather than numerical indices. This is where Tcl's associative arrays (often called hash maps or dictionaries in other languages) become invaluable. This module will introduce you to arrays and their practical applications in Surpac scripting.

11.1 Understanding Tcl Arrays

Unlike lists where elements are accessed by their numerical position (0, 1, 2, ...), array elements are accessed by a key, which can be any string. This allows you to create more readable and self-documenting code.

```
# Creating and setting values in an array
set mine_info(name) "Super Pit"
set mine_info(location) "Kalgoorlie"
set mine_info(ore_type) "Gold"
set mine_info(annual_production_tonnes) 15000000

# Accessing values from an array
SclMessage "INFO" "Mine Name: $mine_info(name)"
SclMessage "INFO" "Location: $mine_info(location)"

# You can also use variables as keys
set attribute "ore_type"
SclMessage "INFO" "Ore Type: $mine_info($attribute)"
```

> **Notice the syntax:** `array_name(key)` **. The key is enclosed in parentheses.**

11.2 Iterating Through Arrays

You can loop through all the keys or values in an array using commands like `array names` **and** `foreach` **.**

```
# Get all keys (attribute names) in the array
set all_keys [array names mine_info]
SclMessage "INFO" "All attributes: $all_keys"

# Loop through each key and print its value
foreach key $all_keys {
    SclMessage "INFO" "  $key: $mine_info($key)"
}

# You can also check if a key exists
if {[array exists mine_info(manager)]} {
    SclMessage "INFO" "Manager: $mine_info(manager)"
} else {
    SclMessage "INFO" "Manager information not available."
}
```

11.3 Practical Application: Storing Configuration Data

Arrays are excellent for storing configuration settings or parameters for your scripts, making them easy to manage and update.

```
# Define script configuration using an array
array set config {
    input_folder "C:/Surpac_Projects/Data"
    output_folder "C:/Surpac_Projects/Results"
    default_grade_cutoff 0.5
    log_file_enabled true
}

# Use configuration values in your script
SclMessage "INFO" "Input data from: $config(input_folder)"

if {$config(log_file_enabled)} {
    SclMessage "INFO" "Logging is enabled."
} else {
    SclMessage "INFO" "Logging is disabled."
}

# You can easily update configuration values
```

```
set config(default_grade_cutoff) 0.7
SclMessage "INFO" "New cutoff: $config(default_grade_cutoff)"
```

11.4 Arrays for Data Aggregation

Arrays can also be used to aggregate data, for example, counting occurrences or summing values based on categories.

```
 # Example: Counting rock types from a list of samples
set samples [list "GRANITE" "BASALT" "GRANITE" "ANDESITE" "BASALT" "GRANITE"]

array set rock_counts {}

foreach rock_type $samples {
    # Increment the count for each rock type
    # If the key doesn't exist, it's treated as 0 initially
    incr rock_counts($rock_type)
}

SclMessage "INFO" "Rock Type Counts:"
foreach type [array names rock_counts] {
    SclMessage "INFO" "  $type: $rock_counts($type)"
}
```

Next Steps: You now understand how to use Tcl arrays for flexible data storage and retrieval. In Module 12, we will dive deeper into the GUIDO toolkit, exploring more comprehensive widget properties and advanced layout techniques for building sophisticated user interfaces.

Module 12: Advanced GUIDO: Comprehensive Widget Properties and Layouts

Module 7 introduced the basics of creating user interfaces with GUIDO. This module dives deeper into the extensive properties (or "switches") available for various GUIDO widgets, allowing you to create more sophisticated, validated, and visually appealing forms. We'll also explore advanced layout techniques, including grouping elements and using tabs.

12.1 Common Widget Properties (Switches)

Many GUIDO widgets share common properties that control their appearance and behavior:

- `-dependency <widget_name> <value>` : Makes the visibility or enabled state of the current widget dependent on the value of another widget.
- `-height <characters>` : Specifies the height of the object in characters.
- `-label "<text>"` : Specifies the text label displayed next to the widget.
- `-icon "<path_to_icon.gif>"` : Displays an icon next to the label (for some widgets).
- `-tip "<tooltip_text>"` : Provides tooltip text that appears when the user hovers over the widget.
- `-width <characters>` : Specifies the width of the field in characters.
- `-default <value>` : Sets a default value for the widget when the form opens.
- `-display_length <characters>` : Specifies the visible width of the input field box.
- `-format <type>` : Applies data type validation. Common types include: `none` , `double` , `float` , `integer` , `short` , `datetime` , `range` , `string_field` , `decimal_angle` , `dms_angle` , `db_charset` , `colour` , `string` , `real_exp` , `font` , `boolean` .
- `-high_bound <value>` : Specifies the maximum allowed numerical value.
- `-input <true/false>` : Determines if the widget can accept user input.
- `-low_bound <value>` : Specifies the minimum allowed numerical value.
- `-max_length <characters>` : Specifies the maximum number of characters allowed in the input.
- `-null <true/false>` : Specifies if the widget can be left empty.
- `-translate <type>` : Controls character translation (e.g., `none` , `lower` , `mixed`).

12.2 Specific Widget Enhancements

`GuidoField`

Beyond the common properties, `GuidoField` is often used with `-format` for data validation.

```
GuidoField $::guido(my_number_field) -parent $::guido(form) \
    -label "Enter a number" \
    -format "double" \
    -low_bound 0.0 \
    -high_bound 100.0 \
    -default 50.0
```

`GuidoComboBox`

Used for dropdown lists. The `-exclusive` switch is important:

- `-exclusive true` **(default):** User can only select from the provided list.
- `-exclusive false` : User can select from the list OR type in a new value.

```
GuidoComboBox $::guido(rock_type) -parent $::guido(form) \
    -label "Rock Type" \
    -list {Granite Basalt Andesite} \
    -exclusive false
```

`GuidoCheckBox`

For boolean (true/false) options.

```
GuidoCheckBox $::guido(enable_logging) -parent $::guido(form) \
    -label "Enable Logging" \
    -value true
```

`GuidoRadioButton` and `GuidoButtonGroupPanel`

Used for mutually exclusive options. Radio buttons must be grouped within a `GuidoButtonGroupPanel` .

```
GuidoButtonGroupPanel $::guido(output_format_group) -parent $::guido(form) \
    -label "Output Format" \
    -layout "vertical"

GuidoRadioButton $::guido(output_format_csv) -parent $::guido(output_format_group) \
    -label "CSV" -value "csv" -group "output_format"
```

```
GuidoRadioButton $::guido(output_format_txt) -parent $::guido(output_format_group) \
    -label "Text" -value "txt" -group "output_format" -selected true
```

GuidoFileBrowserField

Enhanced file selection.

- `-start_dir "<path>"` : Specifies the initial directory for the browser.
- `-extension <true/false>` : If `true` , the file extension is trimmed from the returned name.
- `-multiple_selection <true/false>` : Allows selecting multiple files.

GuidoLabel

For displaying static text. Can be used for instructions or titles.

```
GuidoLabel $::guido(instructions) -parent $::guido(form) \
    -label "Please fill in the required fields below:"
```

GuidoTable

For displaying or entering tabular data. Requires defining columns (`GuidoField` s) within it.

```
GuidoTable $::guido(my_table) -parent $::guido(form) \
    -instances 5 5 5 ; # Initial rows, max rows, increment
    -interactive true ; # Allow user to edit

    GuidoField col1 { -label "Name" -width 10 }
    GuidoField col2 { -label "Value" -width 5 -format "double" }
```

GuidoChart

For embedding simple charts directly into your form.

GuidoHTMLPane

Allows you to embed formatted HTML content directly into your form, useful for rich text instructions or displaying complex information.

GuidoColumnValueBrowser

A specialized browser for selecting column values from a database. Can use `-shared_list` to optimize performance if multiple browsers use the same list.

12.3 Advanced Layouts: Grouping and Tabs

Grouping Parameters with `GuidoPanel`

You can use `GuidoPanel` with the `-border etched true` switch to visually group related parameters within a box on your form. This significantly improves form organization and readability.

```
GuidoPanel $::guido(input_panel) -parent $::guido(form) \
    -label "Input Settings" \
    -border etched true \
    -layout "vertical"

    # Add your input fields here, e.g., GuidoFileBrowserField
    GuidoFileBrowserField $::guido(input_file) -parent $::guido(input_panel) -label "File"
```

Using Tabs for Complex Forms

For very complex forms with many parameters, you can organize widgets into different tabs using a `GuidoTabbedPane` . Each tab is essentially a separate panel.

```
GuidoTabbedPane $::guido(main_tabs) -parent $::guido(form)

    # Define the first tab
    GuidoPanel $::guido(general_tab) -parent $::guido(main_tabs) \
        -label "General" \
        -layout "vertical"
        # Add widgets for the General tab here

    # Define the second tab
    GuidoPanel $::guido(advanced_tab) -parent $::guido(main_tabs) \
        -label "Advanced" \
        -layout "vertical"
        # Add widgets for the Advanced tab here
```

Next Steps: You now have a comprehensive understanding of GUIDO's capabilities for building sophisticated user interfaces. In Module 13, we will explore how to read from and to generic text files, which is essential for custom reporting and data exchange beyond Surpac's native file formats.

Module 13: General Text File Input/Output

While SCL provides powerful commands for working with Surpac's native file formats (like `.str` and `.dtm`), you will often need to read from or write to generic text files (e.g., `.txt`, `.csv`, custom log files). Tcl has built-in commands for robust file input/output (I/O) that are essential for custom reporting, data exchange with other software, and logging script activity.

13.1 Opening and Closing Files

To work with a file, you first need to open it using the `open` command. This returns a **file handle**, which you then use for all subsequent operations on that file. When you're finished, it's crucial to close the file using the `close` command to ensure all data is written and resources are released.

The `open` command takes two main arguments: the file path and the mode.

- **Modes:**
  - `r` : Read-only. The file must exist.
  - `w` : Write-only. Creates a new file or truncates (empties) an existing one.
  - `a` : Append-only. Creates a new file or appends to the end of an existing one.
  - `r+` : Read and write. The file must exist.
  - `w+` : Read and write. Creates a new file or truncates an existing one.
  - `a+` : Read and append. Creates a new file or appends to the end of an existing one.

```
 # Open a file for writing (will create or overwrite)
set output_fh [open "my_report.txt" "w"]

# ... write to file ...

# Close the file
close $output_fh

SclMessage "INFO" "Report file created."
```

13.2 Writing to Files: `puts`

The `puts` command is used to write data to a file. You can specify the file handle as the first argument. If no file handle is given, it defaults to writing to the console (or Surpac's message window).

```
 set output_fh [open "my_log.txt" "a"]

puts $output_fh "Script started at [clock format [clock seconds]]"
puts $output_fh "Processing complete."

close $output_fh

SclMessage "INFO" "Log entry added."
```

13.3 Reading from Files: `gets` and Loops

The `gets` command reads a single line from a file. It returns the number of characters read, or `-1` if the end of the file is reached. This makes it perfect for reading files line by line within a `while` loop.

```
 # Open a file for reading
set input_fh [open "data.csv" "r"]

SclMessage "INFO" "Reading data.csv:"

# Loop through each line until end of file
while {[gets $input_fh line] >= 0} {
    SclMessage "INFO" "Read line: $line"
    # You can then use string manipulation (Module 14) to parse the line
}

close $input_fh
SclMessage "INFO" "Finished reading data.csv."
```

13.4 Practical Example: Simple Data Logger

This script will log some information to a text file, appending to it if it already exists. It also demonstrates reading a simple CSV-like file.

Create a file named `simple_logger.tcl` :

```
 # Simple Data Logger

set log_file "script_activity.log"
set data_file "sample_data.txt"

# --- Write to log file ---
set log_fh [open $log_file "a"]
puts $log_fh "[clock format [clock seconds]] - Script started."

# --- Create a sample data file (if it doesn't exist) ---
if {![file exists $data_file]} {
    set data_fh [open $data_file "w"]
    puts $data_fh "PointID,X,Y,Z"
    puts $data_fh "P1,100.0,200.0,50.0"
    puts $data_fh "P2,105.5,201.2,51.5"
    puts $data_fh "P3,110.1,202.8,52.3"
    close $data_fh
    SclMessage "INFO" "Created sample data file: $data_file"
}

# --- Read from data file and log its content ---
set data_fh [open $data_file "r"]
puts $log_fh "--- Content of $data_file ---"

while {[gets $data_fh line] >= 0} {
    puts $log_fh "  $line"
}
close $data_fh

puts $log_fh "[clock format [clock seconds]] - Script finished."
close $log_fh

SclMessage "INFO" "Activity logged to $log_file"
```

Next Steps: You can now manage generic text files. In Module 14, we will delve into advanced string and list manipulation techniques, which are crucial for parsing and formatting the data you read from and write to files.

Module 14: Advanced String and List Manipulation

Tcl is fundamentally a string-based language, and lists are its most common data structure. Mastering advanced string and list manipulation techniques is crucial for parsing data from files, formatting output, and generally handling textual information efficiently within your scripts.

14.1 String Manipulation

Tcl provides a powerful `string` command with various subcommands for common string operations:

- `string length <string>` **: Returns the number of characters in a string.**
- `string index <string> <index>` **: Returns the character at a specific position (0-indexed).**
- `string range <string> <first> <last>` **: Extracts a substring.**
- `string tolower <string>` **: Converts a string to lowercase.**
- `string toupper <string>` **: Converts a string to uppercase.**
- `string trim <string> [<chars>]` **: Removes leading/trailing whitespace or specified characters.**
- `string first <substring> <string>` **: Finds the first occurrence of a substring.**
- `string last <substring> <string>` **: Finds the last occurrence of a substring.**
- `string replace <string> <first> <last> <newstring>` **: Replaces a portion of a string.**

```
set my_text "  Hello World!  "
SclMessage "INFO" "Original: '$my_text'"
SclMessage "INFO" "Length: [string length $my_text]"
SclMessage "INFO" "First char: [string index $my_text 2]"
SclMessage "INFO" "Substring: [string range $my_text 2 6]"
SclMessage "INFO" "Lowercase: [string tolower $my_text]"
SclMessage "INFO" "Trimmed: '[string trim $my_text]'"
```

14.2 List Manipulation

Lists are fundamental in Tcl. Here are some key commands for working with them:

- `list <item1> <item2> ...` **: Creates a new list.**
- `lindex <list> <index>` **: Extracts an element from a list by its 0-based index.**
- `lrange <list> <first> <last>` **: Extracts a sub-list.**
- `llength <list>` **: Returns the number of elements in a list.**
- `lappend <listVar> <item1> ...` **: Appends elements to a list variable.**
- `linsert <list> <index> <item1> ...` **: Inserts elements into a list at a specific position.**
- `lreplace <list> <first> <last> [<item1> ...]` **: Replaces elements in a list.**
- `lsearch <list> <pattern>` **: Searches for elements matching a pattern.**
- `lsort <list>` **: Sorts the elements of a list.**
- `split <string> <delimiter>` **: Splits a string into a list based on a delimiter.**
- `join <list> <delimiter>` **: Joins list elements into a string using a delimiter.**

```
set my_list [list "apple" "banana" "cherry"]
SclMessage "INFO" "Original list: $my_list"
SclMessage "INFO" "Length: [llength $my_list]"
SclMessage "INFO" "Second item: [lindex $my_list 1]"

lappend my_list "date"
SclMessage "INFO" "Appended list: $my_list"

set csv_line "10.5,20.1,30.7,gold"
set parsed_data [split $csv_line ","]
SclMessage "INFO" "Parsed data: $parsed_data"

set joined_string [join $parsed_list " | "]
SclMessage "INFO" "Joined string: $joined_string"
```

14.3 Practical Example: Parsing and Formatting Drillhole Data

This script will simulate reading a simple drillhole data line (e.g., from a text file), parse it, and then format it for display or further processing.

```
# Practical Example: Parsing and Formatting Drillhole Data

set drillhole_data_line "DH001,1234.56,5678.90,150.20,Au,1.25,Cu,0.50"
SclMessage "INFO" "Original data line: $drillhole_data_line"

# 1. Split the line into a list using comma as delimiter
set fields [split $drillhole_data_line ","]
```

```
# 2. Extract specific fields using lindex
set drillhole_id [lindex $fields 0]
set x_coord [lindex $fields 1]
set y_coord [lindex $fields 2]
set z_coord [lindex $fields 3]
set primary_element [lindex $fields 4]
set primary_grade [lindex $fields 5]

SclMessage "INFO" "--- Parsed Data ---"
SclMessage "INFO" "ID: $drillhole_id"
SclMessage "INFO" "Coordinates: ($x_coord, $y_coord, $z_coord)"
SclMessage "INFO" "Primary Element: $primary_element Grade: $primary_grade"

# 3. Format a summary string
set summary_message [format "Drillhole %s at %.2f, %.2f, %.2f has %s grade of %.2f" \
    $drillhole_id $x_coord $y_coord $z_coord $primary_element $primary_grade]

SclMessage "INFO" "--- Summary ---"
SclMessage "INFO" "$summary_message"

# 4. Example of modifying a list (adding a new element)
set assay_list [lrange $fields 4 end] ; # Get elements and grades
lappend assay_list "Ag" "15.0"
SclMessage "INFO" "Updated assay list: $assay_list"

# 5. Example of converting element name to uppercase
set upper_element [string toupper $primary_element]
SclMessage "INFO" "Uppercase element: $upper_element"
```

Conclusion: With these advanced string and list manipulation techniques, you can effectively process and format almost any textual data within your Tcl/SCL scripts, making them even more powerful for data integration and reporting.

Module 15: Surpac Logicals, Command Aliases, and Hotkeys

To make your Surpac scripts truly powerful and user-friendly, it's essential to understand how to integrate them seamlessly into the Surpac environment. This module covers three key customization features: Logicals, Command Aliases, and Hotkeys.

15.1 Surpac Logicals: Abstracting File Paths

A Logical in Surpac is a named reference that maps to a physical directory on your hard drive at runtime. This system insulates your scripts and configurations from hard-coded paths, making them portable across different Surpac installations or network drives.

Why use Logicals?

- **Portability: Scripts don't break if the physical location of files changes.**
- **Shorter Paths: Use short, memorable names instead of long directory paths.**
- **Standardization: Centralize data and macro storage for teams.**

Logicals come in three types:

- **System Logicals: Defined by the Surpac installation (e.g., `SSI_ETC:` , `SSI_STYLES:`). Stored in `translate.ssi` . You generally should not modify these.**
- **User Logicals: Defined by users for shared resources. Stored in `logicals.ssi` (located in the `SSI_ETC:` directory).**
- **Personal Logicals: Defined by individual users for their own files. Can be stored in any file (e.g., `my_macros.txt`) and specified via Surpac's `Customise > Default Preferences > Alias` tab.**

Example of a Logical Definition (in `logicals.ssi` or a personal logicals file):

```
MY_MACROS: C:/Surpac_Projects/MyScripts/
```

> **Important: Logical names must end with a colon (`:`), and the physical path must end with a forward or backward slash. Forward slashes are generally recommended in Tcl scripts.**

15.2 Command Aliases: Creating Shortcuts

A Command Alias allows you to create a short, memorable name for any Surpac command or Tcl/SCL script. This is incredibly useful for frequently used functions.

Aliases are defined in a text file (e.g., `my_aliases.txt`) with a specific syntax:

```
"<ALIAS_NAME>" "<COMMAND_OR_SCRIPT_PATH>"
```

To link an alias to a Tcl/SCL script, you must use the special `MACRO:` keyword, followed by the logical name (or full path) to your script.

```
# Example: Alias for a Tcl script
"RUN_MY_SCRIPT" "MACRO:MY_MACROS:my_script.tcl"

# Example: Alias for a Surpac internal function
"ZOOM_ALL" "ZOOM ALL"
```

To make Surpac recognize your alias file, you must specify its full path (or logical) in `Customise > Default Preferences > Alias` tab.

15.3 Hotkeys: Keyboard Shortcuts for Scripts

Hotkeys allow you to assign your Tcl/SCL scripts or Surpac functions to keyboard keys (like F1-F12) for instant execution. This is defined in a keymaps file (e.g., `keymaps.ssi`).

The syntax is similar to command aliases:

```
"<KEY_NAME>" FUNCTION "<COMMAND_OR_SCRIPT_PATH>"
```

Key names must be lowercase (e.g., `f1`, `f11`).

```
# Example: Assigning a script to F11
"f11" FUNCTION "MACRO:MY_MACROS:hello_surpac.tcl"
```

The keymaps file is also specified in `Customise > Default Preferences > Alias` tab.

Conclusion: By effectively using Logicals, Command Aliases, and Hotkeys, you can create a highly customized and efficient Surpac environment, making your scripts easily accessible and your workflow much smoother.

Module 16: Interactive Graphical Selection (`SclSelectPoint`)

While you can draw graphics to provide visual feedback (Module 5), sometimes you need your script to interact with the user by allowing them to select objects directly in the Surpac graphics window. The `SclSelectPoint` command is a powerful tool for this, enabling your scripts to retrieve detailed information about graphically selected points.

16.1 The `SclSelectPoint` Command

The `SclSelectPoint` function allows you to prompt the user to select a point in the graphics window. Upon selection, it returns a wealth of information about that point into specified Tcl variables.

Syntax

```
SclSelectPoint <handle> <prompt> <layer> <stringNo> <segNo> <pointNo> <x> <y> <z> <desc>
```

Arguments Explained:

- **`<handle>` : A variable that will receive a reference to the selected point object. This handle can be used with other SCL functions (e.g., `SclGetValueByName`) to get more details about the point.**
- **`<prompt>` : A text string displayed in the Surpac message window, instructing the user what to select (e.g., "Select the setup point").**
- **`<layer>` : A variable that will receive the name of the graphics layer the selected point belongs to.**
- **`<stringNo>` : A variable that will receive the string number of the selected point.**
- **`<segNo>` : A variable that will receive the segment number of the selected point.**
- **`<pointNo>` : A variable that will receive the point number within its segment.**
- **`<x>`, `<y>`, `<z>` : Variables that will receive the X, Y, and Z coordinates of the selected point.**
- **`<desc>` : A variable that will receive the entire description field of the selected point.**

Return Value: The command returns `SCL_OK` on successful selection or `SCL_ERROR` if the user cancels the selection (e.g., by pressing Escape).

16.2 Practical Example: Bearing and Distance Tool

This script will prompt the user to select two points in the graphics window and then calculate and display the bearing and distance between them. This demonstrates how to use `SclSelectPoint` to get coordinates and then use them in calculations.

Create a file named `bearing_distance_tool.tcl` :

```
# Practical Example: Bearing and Distance Tool

SclMessage "INFO" "Please select the FIRST point."
set status1 [SclSelectPoint pnt1 "Select the first point" \
    layer1 str_no1 seg_no1 pnt_no1 x1 y1 z1 desc1]

if {$status1 == "SCL_ERROR"} {
    SclMessage "WARNING" "First point selection cancelled. Aborting."
    return
}

SclMessage "INFO" "Please select the SECOND point."
set status2 [SclSelectPoint pnt2 "Select the second point" \
    layer2 str_no2 seg_no2 pnt_no2 x2 y2 z2 desc2]

if {$status2 == "SCL_ERROR"} {
    SclMessage "WARNING" "Second point selection cancelled. Aborting."
    return
}

# Calculate Bearing and Distance
# Note: Surpac has a built-in SclBearingDistance command for this, but we'll do it manually for demonstration.

set dx [expr {$x2 - $x1}]
set dy [expr {$y2 - $y1}]
set dz [expr {$z2 - $z1}]

set distance_2d [expr {sqrt($dx*$dx + $dy*$dy)}]
set distance_3d [expr {sqrt($dx*$dx + $dy*$dy + $dz*$dz)}]

# Calculate bearing (in radians, then convert to degrees)
set bearing_rad [expr {atan2($dx, $dy)}]
set bearing_deg [expr {($bearing_rad * 180.0 / 3.1415926535) % 360.0}]

# Ensure bearing is positive
```

```
if {$bearing_deg < 0} {
    set bearing_deg [expr {$bearing_deg + 360.0}]
}

SclMessage "INFO" "--- Results ---"
SclMessage "INFO" "Point 1: ($x1, $y1, $z1)"
SclMessage "INFO" "Point 2: ($x2, $y2, $z2)"
SclMessage "INFO" "2D Distance: [format %.2f $distance_2d]"
SclMessage "INFO" "3D Distance: [format %.2f $distance_3d]"
SclMessage "INFO" "Bearing (from North): [format %.2f $bearing_deg] degrees"

# Clean up handles (important for memory management)
SclDestroyHandle $pnt1
SclDestroyHandle $pnt2
```

Next Steps: You can now create scripts that interact directly with the Surpac graphics environment. In Module 17, we will explore Tcl's native file system operations, allowing you to manage files and directories beyond Surpac's SWA.

Module 17: Tcl File System Operations

Beyond reading and writing the content of files (as covered in Module 13), Tcl provides a comprehensive set of commands for interacting directly with the file system. This includes tasks like copying, deleting, renaming files, creating and removing directories, and listing files based on patterns. These operations are crucial for managing your project data and organizing script outputs.

17.1 The `file` Command: Managing Files and Directories

The `file` command is a versatile Tcl command with many subcommands for file system operations. Here are some of the most commonly used ones:

- **`file copy [-force] <source> <target>` : Copies a file. Use `-force` to overwrite an existing target file.**
- **`file delete [-force] <path>` : Deletes a file or an empty directory. Use `-force` to delete read-only files.**
- **`file rename [-force] <source> <target>` : Renames or moves a file or directory. Use `-force` to overwrite an existing target.**
- **`file exists <path>` : Returns `1` (true) if the file or directory exists, `0` (false) otherwise. Useful for conditional logic.**
- **`file mkdir <path>` : Creates a new directory. It will create parent directories if they don't exist.**
- **`file isdirectory <path>` : Returns `1` if the path is a directory, `0` otherwise.**
- **`file isfile <path>` : Returns `1` if the path is a regular file, `0` otherwise.**
- **`file extension <path>` : Returns the file extension (e.g., ".str").**
- **`file rootname <path>` : Returns the file name without the extension.**
- **`file dirname <path>` : Returns the directory name of a path.**
- **`file join <path_elements> ...` : Joins path elements correctly for the operating system.**

```
# Example: Copying and deleting files
file copy "input.str" "backup.str"
SclMessage "INFO" "Copied input.str to backup.str"

if {[file exists "old_report.txt"]} {
    file delete "old_report.txt"
    SclMessage "INFO" "Deleted old_report.txt"
}

# Example: Creating a directory and checking existence
set new_dir "./output_data"
if {![file exists $new_dir]} {
    file mkdir $new_dir
    SclMessage "INFO" "Created directory: $new_dir"
}

# Example: Renaming a file
file rename "temp_results.csv" "final_results.csv"
SclMessage "INFO" "Renamed temp_results.csv to final_results.csv"
```

17.2 The `glob` Command: Listing Files by Pattern

The `glob` command is used to find files and directories that match a specified pattern. This is incredibly useful for processing multiple files that follow a naming convention.

Syntax

```
glob [-nocomplain] <pattern>
```

- **`<pattern>` : Can include wildcards like `*` (matches any sequence of characters), `?` (matches any single character), and `[]` (matches characters within a set).**
- **`-nocomplain` : Prevents an error if no files match the pattern.**

```
# Example: List all .str files in the current directory
set str_files [glob "*.str"]
SclMessage "INFO" "String files found: $str_files"

# Example: Process all CSV files in a subdirectory
set csv_files [glob "data/*.csv"]
foreach csv_file $csv_files {
    SclMessage "INFO" "Processing CSV: $csv_file"
    # ... your processing logic here ...
}

# Example: Find files starting with 'drill' and ending with .txt or .log
set drill_logs [glob "drill*.{txt,log}"]
SclMessage "INFO" "Drill logs: $drill_logs"
```

17.3 Practical Example: Batch File Processing

This script will find all `.str` files in a specified input directory, process each one (e.g., find its Z-extents using the logic from Module 3), and then log the results to a summary file.

Create a file named `batch_processor.tcl` **:**

```
 # Practical Example: Batch File Processor

set input_dir "./input_strings"
set output_log "./processing_summary.log"

# Ensure input directory exists
if {![file isdirectory $input_dir]} {
    SclMessage "ERROR" "Input directory not found: $input_dir"
    return
}

# Open log file for writing (overwrite if exists)
set log_fh [open $output_log "w"]
puts $log_fh "--- Batch Processing Summary - [clock format [clock seconds]] ---"

# Find all .str files in the input directory
set str_files [glob "$input_dir/*.str"]

if {[llength $str_files] == 0} {
    SclMessage "WARNING" "No .str files found in $input_dir."
    puts $log_fh "No .str files found."
} else {
    SclMessage "INFO" "Found [llength $str_files] string files to process."
    foreach file_path $str_files {
        set file_name [file tail $file_path] ; # Get just the filename
        puts $log_fh "\nProcessing file: $file_name"

        # --- Logic to find Z-extents (from Module 3) ---
        set file_handle [SclSwaOpenFile $file_path]
        set min_z 99999
        set max_z -99999
        set first_point_found 0

        set string_list [SclGetStrings $file_handle]
        foreach string_id $string_list {
            set string_handle [SclGetItem $file_handle "string" $string_id]
            set point_count [SclGetItem $string_handle "count"]

            for {set i 1} {$i <= $point_count} {incr i} {
                set point_handle [SclGetItem $string_handle "point" $i]
                set current_z [SclGetValueByName $point_handle "z"]

                if {$first_point_found == 0} {
                    set min_z $current_z
                    set max_z $current_z
                    set first_point_found 1
                }

                if {$current_z < $min_z} {
                    set min_z $current_z
                }
                if {$current_z > $max_z} {
                    set max_z $current_z
                }
            }
        }
        SclDestroyHandle $file_handle
        # --- End of Z-extents logic ---

        puts $log_fh "  Min Z: [format %.2f $min_z]"
        puts $log_fh "  Max Z: [format %.2f $max_z]"
    }
}

close $log_fh
SclMessage "INFO" "Batch processing complete. See $output_log for summary."
```

Conclusion: You can now perform comprehensive file system operations and process files in batches, significantly enhancing your automation capabilities in Surpac.

Module 18: Manipulating Surpac Ranges (`SclRange` Commands)

Surpac frequently uses the concept of "ranges" to define sections, benches, or other intervals of data. These ranges can be simple (e.g., 1000 to 2000 at 100m intervals) or more complex (non-uniform). SCL provides specific commands to expand, query, and iterate through these ranges, which is essential for automating tasks that operate on defined data intervals.

18.1 Expanding a Range: `SclRangeExpand`

The `SclRangeExpand` command takes a Surpac range specification (e.g., `"1000,2000,100"`) and expands it into its individual components in memory. This allows you to then access each value within that range.

Syntax

```
SclRangeExpand <RangeHandle> <RangeExpression>
```

- **`<RangeHandle>` : A variable that will receive a reference (handle) to the expanded range object in memory.**
- **`<RangeExpression>` : The Surpac range string (e.g., `"1000,2000,100"` or `"100,200,25;235;256;300,500,50"`).**

```
# Example: Expand a simple range
set my_range_expression "1000,2000,100"
set range_handle [SclRangeExpand my_range_handle $my_range_expression]

SclMessage "INFO" "Expanded range handle: $range_handle"
```

18.2 Querying a Range: `SclRangeGetCount` and `SclRangeGet`

Once a range has been expanded using `SclRangeExpand` , you can query its properties.

- **`SclRangeGetCount` : Returns the total number of items (values) in the expanded range.**
- **`SclRangeGet` : Retrieves a specific value from the expanded range by its 0-based position (index).**

```
# Assume range_handle is already set from SclRangeExpand

# Get the number of items in the range
set num_items [SclRangeGetCount $range_handle]
SclMessage "INFO" "Number of items in range: $num_items"

# Get the first item (index 0)
set first_value [SclRangeGet $range_handle 0]
SclMessage "INFO" "First value: $first_value"

# Get the last item (index num_items - 1)
set last_value [SclRangeGet $range_handle [expr {$num_items - 1}]]
SclMessage "INFO" "Last value: $last_value"
```

18.3 Practical Example: Iterating Through Sections

This script will take a range expression for sections, expand it, and then iterate through each section, printing its name. This is a common pattern for automating tasks across multiple sections or benches.

Create a file named `iterate_sections.tcl` :

```
# Practical Example: Iterate Through Sections Defined by a Range

set section_range_expression "7000,8000,100" ; # Example: from 7000 to 8000, every 100m

SclMessage "INFO" "Expanding section range: $section_range_expression"

# 1. Expand the range expression
set range_handle [SclRangeExpand my_section_range $section_range_expression]

if {$range_handle == ""} {
    SclMessage "ERROR" "Failed to expand range. Check expression syntax."
    return
}

# 2. Get the total number of items in the expanded range
set num_sections [SclRangeGetCount $range_handle]
SclMessage "INFO" "Found $num_sections sections."
```

```
# 3. Loop through each section
for {set i 0} {$i < $num_sections} {incr i} {
    # Get the value of the current section
    set current_section_value [SclRangeGet $range_handle $i]

    # Format the section name (e.g., for a file name)
    set section_file_name "section_[format %.0f $current_section_value].str"

    SclMessage "INFO" "Processing section: $current_section_value (File: $section_file_name)"
    # In a real script, you would now perform operations on this section,
    # e.g., recall the section file, perform calculations, save results.
}

# 4. Clean up the range handle
SclDestroyHandle $range_handle

SclMessage "INFO" "Section iteration complete."
```

Conclusion: You can now effectively work with Surpac's range definitions, enabling you to automate tasks that require processing data across specific intervals. This completes the expanded set of modules based on the provided training notes.

Module 19: Robust Scripting: Error Handling and Debugging

Even the most carefully written scripts can encounter unexpected situations or errors. Robust scripts anticipate these issues and handle them gracefully, preventing crashes and providing meaningful feedback. This module will introduce you to Tcl's primary error handling mechanism and discuss strategies for debugging your scripts.

19.1 Error Handling with `catch`

The `catch` command is Tcl's fundamental way to trap and manage errors. It executes a script and, if an error occurs, it prevents the error from propagating and crashing the main script. Instead, it returns a status code and captures the error message.

Syntax

```
catch <script> [<resultVar>] [<options>]
```

- **`<script>` : The block of Tcl code you want to execute and monitor for errors.**
- **`<resultVar>` (optional): A variable to store the result of the script (if no error) or the error message (if an error occurs).**
- **Return Value: `catch` returns `0` if the script executed successfully, and `1` if an error occurred. Other return codes indicate different types of non-error exceptions (e.g., `return` , `break` , `continue`).**

```
 # Example: Handling a file that might not exist
set filename "non_existent_file.str"

if {[catch {SclSwaOpenFile $filename} file_handle]} {
    SclMessage "ERROR" "Failed to open file: $file_handle"
    SclMessage "INFO" "Please ensure '$filename' exists and is accessible."
    return ; # Stop script execution gracefully
} else {
    SclMessage "INFO" "Successfully opened file with handle: $file_handle"
    # ... continue with script using $file_handle ...
    SclDestroyHandle $file_handle
}
```

> **Using `catch` is crucial when dealing with operations that might fail, such as file I/O, user input, or complex Surpac functions.**

19.2 Debugging Strategies

When your script doesn't behave as expected, debugging helps you find and fix the problems. Here are some common strategies:

- **`SclMessage "DEBUG" "..."` : Use `SclMessage` with the "DEBUG" type (or "INFO") to print variable values, execution flow messages, and intermediate results to the Surpac message window. This is your primary tool for understanding what your script is doing step-by-step.**
- **Temporary `puts` statements: For quick checks, you can temporarily insert `puts` commands directly into your script to print values to the console where you run the script (if applicable) or to a log file (Module 13).**
- **Simplify and Isolate: If a complex script is failing, comment out sections of code and test smaller parts in isolation until you pinpoint the problematic area.**
- **Check Input Data: Verify that the data your script is receiving (from files, UI, or other functions) is in the expected format and range.**
- **Review Documentation: Double-check the syntax and arguments for any SCL or Tcl command you are using, especially if you are getting unexpected errors.**
- **Surpac Macro Recorder: For SCL functions, record the manual operation in Surpac and compare the generated macro with your script to ensure you are calling the function correctly.**

```
 # Example of using SclMessage for debugging
set my_value 123.45
SclMessage "DEBUG" "Value of my_value before calculation: $my_value"

set result [expr {$my_value * 2}]
SclMessage "DEBUG" "Result after multiplication: $result"

# ... later in the script ...
if {$result > 200} {
    SclMessage "DEBUG" "Condition met: result is greater than 200."
} else {
    SclMessage "DEBUG" "Condition NOT met: result is not greater than 200."
}
```

Next Steps: With robust error handling and debugging skills, you can write more reliable and maintainable scripts. In Module 20, we will explore advanced text processing using Tcl's powerful regular expression capabilities.

Module 20: Advanced Text Processing: Regular Expressions

Regular expressions (often shortened to regex or regexp) are powerful patterns used to match character combinations in strings. Tcl has built-in commands that allow you to perform complex text searching, validation, and substitution, which are invaluable for parsing unstructured data, validating user input, or extracting specific information from text files.

20.1 Introduction to Regular Expressions

A regular expression is a sequence of characters that defines a search pattern. They can be simple (e.g., matching a specific word) or complex (e.g., matching email addresses or phone numbers). Tcl's regex engine is powerful and follows standard regex syntax.

Some common regex metacharacters:

- `.` **: Matches any single character (except newline).**
- `*` **: Matches the preceding element zero or more times.**
- `+` **: Matches the preceding element one or more times.**
- `?` **: Matches the preceding element zero or one time.**
- `[abc]` **: Matches any one of the characters a, b, or c.**
- `[^abc]` **: Matches any character except a, b, or c.**
- `[0-9]` **or** `\d` **: Matches any digit.**
- `[a-zA-Z]` **or** `\w` **: Matches any word character (alphanumeric + underscore).**
- `^` **: Matches the beginning of the string.**
- `$` **: Matches the end of the string.**
- `\s` **: Matches any whitespace character.**
- `(pattern)` **: Groups patterns and captures matches.**

20.2 Searching with `regexp`

The `regexp` **command is used to check if a string matches a regular expression pattern. It can also extract matching substrings.**

Syntax

```
regexp ?switches? <pattern> <string> ?matchVar? ?subMatch1? ?subMatch2? ...
```

- `<pattern>` **: The regular expression to match.**
- `<string>` **: The string to search within.**
- `matchVar` **(optional): A variable to store the entire matched substring.**
- `subMatch1, subMatch2, ...` **(optional): Variables to store substrings captured by parentheses in the pattern.**
- **Return Value: Returns** `1` **if the pattern matches,** `0` **otherwise.**

```
set text "Drillhole DH123 has a grade of 1.5 Au."

# Check if a pattern exists
if {[regexp {DH\d+} $text]} {
    SclMessage "INFO" "Found a drillhole ID."
}

# Extract a drillhole ID and grade
regexp {DH(\d+).*grade of (\d+\.\d+)} $text all_match dh_id grade
SclMessage "INFO" "Extracted: ID=$dh_id, Grade=$grade"

# Using non-greedy matching
set log_line "INFO: Start [timestamp] End [timestamp]"
regexp {INFO: (.*?) End (.*)} $log_line dummy start_time end_time
SclMessage "INFO" "Log times: Start=$start_time, End=$end_time"
```

20.3 Substituting with `regsub`

The `regsub` **command is used to perform substitutions based on regular expression matches. It returns a new string with the replacements.**

Syntax

```
regsub ?switches? <pattern> <string> <subSpec> <varName>
```

- `<pattern>` **: The regular expression to match.**
- `<string>` **: The input string.**
- `<subSpec>` **: The replacement string. Captured groups (e.g.,** `\1` **,** `\2` **) can be used here.**
- `<varName>` **: The name of the variable to store the result.**

- **Switches:** `-all` **(replaces all occurrences),** `-nocase` **(case-insensitive matching).**

```
set filename "report_v1.txt"

# Replace .txt with .csv
regsub {\.txt$} $filename ".csv" new_filename
SclMessage "INFO" "New filename: $new_filename"

set data_line "Au: 1.25, Ag: 15.0, Cu: 0.3"
# Replace all occurrences of a colon followed by space with an equals sign
regsub -all {:\s*} $data_line "=" formatted_line
SclMessage "INFO" "Formatted line: $formatted_line"

# Reorder parts of a string
set date_str "2023-07-27"
regsub {(\d{4})-(\d{2})-(\d{2})} $date_str "\3/\2/\1" formatted_date
SclMessage "INFO" "Formatted date: $formatted_date"
```

20.4 Practical Example: Cleaning and Extracting Data from Log Files

This script simulates reading a log file, extracting specific information (like timestamps and message types), and cleaning up entries using regular expressions.

Create a file named `log_parser.tcl` **:**

```
# Practical Example: Log File Parser

set log_file_content {
[2023-07-27 10:00:05] INFO: Script started.
[2023-07-27 10:00:10] WARNING: File not found: temp.str
[2023-07-27 10:00:15] ERROR: Calculation failed due to invalid input.
[2023-07-27 10:00:20] INFO: Processing complete.
}

SclMessage "INFO" "--- Original Log Content ---"
SclMessage "INFO" "$log_file_content"

set processed_log_lines {}

# Split the log content into individual lines
foreach line [split $log_file_content "\n"] {
    if {[string trim $line] == ""} continue ; # Skip empty lines

    # Use regexp to extract timestamp, type, and message
    if {[regexp {^\[(\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2})\]\s*([A-Z]+):\s*(.*)$} $line all_match timestamp type message]} {
        SclMessage "INFO" "Parsed: [string toupper $type] at $timestamp - $message"

        # Clean up the message: remove leading/trailing spaces and replace multiple spaces with single
        set cleaned_message [string trim $message]
        regsub -all {\s+} $cleaned_message " " cleaned_message

        lappend processed_log_lines "$timestamp | $type | $cleaned_message"
    } else {
        SclMessage "WARNING" "Could not parse line: $line"
    }
}

SclMessage "INFO" "\n--- Processed Log Summary ---"
foreach entry $processed_log_lines {
    SclMessage "INFO" "$entry"
}
```

Conclusion: Regular expressions are a powerful tool in your Tcl/SCL arsenal for handling complex text data. Combined with file I/O and list manipulation, they enable sophisticated data extraction and transformation workflows.

Module 21: Script Modularity and Variable Scope

As your Tcl/SCL scripts grow in complexity, organizing your code becomes crucial for readability, maintainability, and reusability. This module will cover how to break down large scripts into smaller, manageable files (modularity) and how to control the visibility and accessibility of variables across different parts of your code (variable scope).

21.1 Script Modularity: The `source` Command

The `source` command allows you to execute the contents of another Tcl script file within the current script. This is Tcl's primary mechanism for modularity, enabling you to organize your code into logical units (e.g., separate files for utility functions, UI definitions, or specific processing steps).

Syntax

```
source <filename>
```

- **`<filename>` : The path to the Tcl script file to execute. This can be an absolute path or a path relative to the current script's directory.**

```
# main_script.tcl

# Source a file containing common utility procedures
source "./my_utilities.tcl"

# Source a file containing UI definitions
source "./my_forms.tcl"

SclMessage "INFO" "Main script started."

# Call a procedure defined in my_utilities.tcl
my_utility_procedure "some_argument"

# Display a form defined in my_forms.tcl
SclRun $::guido(main_form)
```

> **Using `source` promotes code reuse and makes your scripts easier to navigate and debug.**

21.2 Variable Scope: Local, Global, and `upvar`

Understanding variable scope is fundamental to writing correct and predictable Tcl scripts. Tcl has a relatively simple scoping model:

- **Local Scope: Variables created inside a `proc` (procedure) are local to that procedure. They are not accessible outside of it, and they are destroyed when the procedure finishes.**
- **Global Scope: Variables created outside of any `proc` are in the global scope. They are accessible from anywhere in the script, but procedures need to explicitly declare their intent to use or modify a global variable.**

The `global` Command

Inside a procedure, use the `global` command to declare that a variable refers to a global variable, not a new local one.

```
set global_counter 0 ; # Global variable

proc increment_counter {} {
    global global_counter ; # Declare intent to use the global variable
    incr global_counter
    SclMessage "INFO" "Counter: $global_counter"
}

increment_counter ; # Output: Counter: 1
increment_counter ; # Output: Counter: 2
```

The `upvar` Command

The `upvar` command is more advanced and allows a procedure to access a variable in a calling stack frame (e.g., the procedure that called it) by name, rather than by value. This is particularly useful for modifying variables passed by reference or for working with variables in a parent scope without making them global.

Syntax

```
upvar ?level? <otherVar> <myVar>
```

- `?level?` **(optional): Specifies how many levels up the call stack to look (** `0` **is current,** `1` **is caller, etc.). Defaults to** `1` **.**
- `<otherVar>` **: The name of the variable in the other scope.**
- `<myVar>` **: The local name you want to use for that variable within the current procedure.**

```
proc process_data {data_list_name} {
    upvar 1 $data_list_name my_local_list ; # Link local_list to the caller's list

    SclMessage "INFO" "Processing list with [llength $my_local_list] items."
    lappend my_local_list "new_item" ; # This modifies the caller's list
}

set my_main_list [list "A" "B" "C"]
SclMessage "INFO" "Before call: $my_main_list"

process_data my_main_list

SclMessage "INFO" "After call: $my_main_list" ; # Output: A B C new_item
```

Conclusion: By structuring your scripts with modularity and understanding variable scope, you can write more organized, reusable, and less error-prone Tcl/SCL code, especially for larger and more complex automation tasks.