# 💻 Examen 105 - Shells y Scripts de Shell

### 105.2 Personalizar o escribir scripts simples

**Teoría**

Un script de shell es simplemente un archivo de texto que contiene una secuencia de comandos de shell. Cuando ejecutas el script, la shell lee y ejecuta los comandos línea por línea. Los scripts de shell te permiten combinar comandos, usar variables, estructuras de control (condiciones, bucles) y automatizar tareas que de otro modo tendrías que realizar manualmente paso a paso.

**Conceptos Clave:**

1. **¿Qué es un Script de Shell?**
    - Un archivo de texto plano que contiene comandos de shell (como `ls`, `cd`, `echo`, `grep`, etc.) y construcciones de la shell (variables, `if`, `for`, `while`).
2. **La Línea Shebang (`#!`):**
    - La primera línea de un script de shell (y de muchos otros scripts en Linux) debe ser la línea "shebang". Comienza con `#!` seguido de la ruta al intérprete que debe ejecutar el script.
    - El intérprete más común para scripts de shell en Linux es Bash: `#!/bin/bash`.
    - Cuando ejecutas un archivo que tiene la línea shebang y permisos de ejecución, el kernel lee esta línea y pasa el resto del archivo como argumento al intérprete especificado.
3. **Hacer un Script Ejecutable:**
    - Por defecto, los archivos de texto no tienen permiso de ejecución. Para poder ejecutar un script de shell simplemente escribiendo su nombre (ej: `./mi_script.sh`), debes darle permiso de ejecución usando `chmod +x <nombre_script>`.
    - Si un script no tiene permiso de ejecución, puedes ejecutarlo pasándolo como argumento a la shell (ej: `bash mi_script.sh`). En este caso, la línea shebang es ignorada por el kernel, pero la shell la puede leer y usar (o no, dependiendo de la shell y la línea). La forma preferida es usar `chmod +x` y confiar en la línea shebang.
4. **Comentarios:**
    - Las líneas que comienzan con # (excepto la shebang) son comentarios y son ignoradas por el intérprete. Útiles para documentar tu script.
5. **Variables en Scripts:**
    - Puedes establecer variables locales al script (similares a las variables de shell locales) usando `NOMBRE=valor`. No se propagan fuera del script.
    - Accedes al valor de una variable con `$NOMBRE`.
    - Puedes usar variables de entorno existentes (como `$PATH`, `$HOME`) dentro de tus scripts.
6. **Parámetros Posicionales:**

- Cuando ejecutas un script, los argumentos que le pasas están disponibles dentro del script como parámetros posicionales.
- `$0`: El nombre del script en sí.
- `$1`, `$2`, `$3`, ...: Los argumentos pasados al script en orden. `$1` es el primer argumento, `$2` el segundo, y así sucesivamente.
- `$#`: El número total de argumentos pasados al script.
- `$*` y `$@`: Representan todos los argumentos pasados al script (desde `$1` en adelante). `$@` es generalmente preferible dentro de comillas dobles (`"$@"`) porque mantiene los argumentos separados individualmente si contienen espacios. `"$*"` los trata como una sola cadena con espacios entre ellos.

7. **Lectura de Entrada (`read`):**
   - El comando `read` lee una línea de texto desde la entrada estándar (generalmente el teclado) y la almacena en una o más variables.
   - `read mi_variable`: Lee una línea y la guarda en `mi_variable`.
   - `read -p "Mensaje: " mi_variable`: Muestra un mensaje antes de leer la entrada.
8. **Comillas en Scripts:**
   - Las mismas reglas de comillas (`'`, `"`, `\`) que vimos en 103.1 aplican en los scripts. Son esenciales para manejar espacios en nombres de archivo o variables.
9. **Estructuras de Control Básicas:**
   - **`if` statements (Condicionales):** Permiten ejecutar código solo si una condición es verdadera.

     Bash

     ```
     if [ condicion ]; then
       # Código si la condición es verdadera
     fi

     # Con else
     if [ condicion ]; then
       # Código si es verdadera
     else
       # Código si es falsa
     fi

     # Con elif (else if)
     if [ condicion1 ]; then
       # Código si condicion1 es verdadera
     elif [ condicion2 ]; then
       # Código si condicion1 es falsa y condicion2 es verdadera
     else
       # Código si ambas son falsas
     fi
     ```

     Las condiciones se evalúan dentro de corchetes `[]` o doble corchetes `[[]]`. `[[]]` es más flexible. Pruebas comunes: `-f archivo` (archivo existe y es regular), `-d directorio` (directorio existe), `-z cadena` (cadena está vacía), `-n cadena`

(cadena no está vacía), `cadena1 = cadena2`, `cadena1 != cadena2`, `num1 -eq num2` (igual, para números), `num1 -ne num2` (no igual), `num1 -gt num2` (mayor que), `num1 -lt num2` (menor que), `num1 -ge num2` (mayor o igual), `num1 -le num2` (menor o igual).

- **`for` loops (Bucles):** Permiten iterar sobre una lista de elementos.
  Bash

```
for variable in lista_de_elementos; do
  # Código a ejecutar para cada elemento
  echo "Procesando: $variable"
done
```

  La lista de elementos puede ser una secuencia (`{1..5}`), una lista de archivos (`*.txt`), o los parámetros posicionales (`"$@"`).

10. **Comandos de Salida (`exit`) y Código de Retorno:**
    - `exit [código_salida]`: Termina la ejecución del script. El `código_salida` es un número entero (0-255).
    - Un código de salida `0` (cero) generalmente indica éxito.
    - Un código de salida distinto de cero (1-255) indica algún tipo de error.
    - Puedes ver el código de salida del *último* comando o script ejecutado en la variable especial `$?`.
11. **Ejecución de Scripts:**
    - `./mi_script.sh`: Ejecuta el script como un programa independiente (la shebang define el intérprete). Es la forma recomendada. `./` es necesario si el directorio actual no está en `PATH`.
    - `bash mi_script.sh`: Ejecuta el script usando explícitamente `bash` como intérprete. La shebang puede ser ignorada.
    - `source mi_script.sh` o `. mi_script.sh`: Ejecuta el script en la **shell actual**. Los cambios (variables, aliases) persisten después de que el script termina. La shebang es ignorada. Esto se usa a menudo para archivos de configuración como `~/.bashrc`.

**Consideraciones Debian vs. Red Hat:**

Para la escritura de scripts Bash *simples*, no hay diferencias significativas entre distribuciones, ya que el intérprete `/bin/bash` y sus comandos integrados (`echo`, `read`, `if`, `for`, `exit`, `export`, etc.) funcionan igual. Las diferencias surgen si el script llama a comandos externos que tienen sintaxis o ubicación diferente (ej: usar `apt` en lugar de `dnf` para gestión de paquetes), pero eso es específico de la tarea que el script intenta automatizar, no del lenguaje de scripting en sí.