

Uso de Shell

≡ Laboratorio

Laboratorio 1

+ Añadir una propiedad

Comentarios

 Añadir un comentario...

► Comandos básicos Shell



▼ Uso del Shell Guia 2

▼ Parte 1: ¿Qué es el shell?

- El shell es un programa que permite interactuar con el sistema operativo.
- Es un intérprete de comandos: lee lo que escribes, lo interpreta y ejecuta.
- También es un lenguaje de programación: tiene condicionales, bucles, funciones y variables.
- Hay varios tipos de shell: `bash`, `sh`, `zsh`, `ksh`, etc.
- Puedes verificar cuál usas con `echo $SHELL` o `ps -p $$`.

▼ Parte 2: ¿Qué es un shell script?

- Un shell script es un archivo de texto plano que contiene comandos de shell.
- Se ejecuta línea por línea de forma interpretada.
- Se puede escribir en editores como `nano`, `vim`, `emacs`, etc.
- Para crear uno:

```
nano mi_script.sh
```

- Para ejecutarlo:

1. Dar permisos: `chmod +x mi_script.sh`
2. Ejecutar: `./mi_script.sh`

- El `./` se usa porque el directorio actual (`.`) no está en el `$PATH` por seguridad.

▼ Parte 3: Shebang (#!) y Comentarios

- El shebang es la primera línea del script y le dice al sistema qué intérprete usar.

```
#!/bin/bash
```

- Sin el shebang, el script se ejecutará con el shell por defecto (que puede no ser Bash).

- Es muy importante para garantizar que el script se interprete correctamente.

✓ Comentarios en Bash:

- Cualquier línea que comienza con `#` (excepto el shebang) es un **comentario**.
- Los comentarios **no se ejecutan**, sirven para documentar.

```
# Esto es un comentario  
echo "Hola mundo" # También puede ir al final de una línea
```

▼ █ Parte 4: Variables en Shell

✓ Variables escalares (simples)

- Se declaran así (sin espacios alrededor del `=`):

```
nombre=Max
```

- Para acceder a su valor:

```
echo $nombre # Muestra: Max
```

✓ Leer datos del usuario con `read`

```
echo "Ingresa tu nombre:"  
read nombre  
echo "Hola, $nombre"
```

⚠ Reglas para nombres de variables:

- Pueden contener letras, números y guiones bajos (`_`)
- Deben **comenzar con una letra o guion bajo**
- Ejemplos válidos: `edad`, `_usuario1`, `nombre_completo`

▼ █ Parte 5: Arreglos, variables de solo lectura y `unset`

✓ Arreglos en Bash

Un arreglo almacena **múltiples valores** con un solo nombre:

```
frutas=(manzana uva piña)
```

O por índice:

```
frutas[0]=manzana  
frutas[1]=uva
```

⭐ Acceder a los valores:

- Un valor:

```
echo ${frutas[1]} # uva
```

- Todos los valores:

```
echo ${frutas[*]}
```

🔒 Variables de solo lectura

Una vez definidas, no pueden ser modificadas:

```
readonly PI=3.1416
```

🗑 Eliminar variables con `unset`

- Para borrar una variable (si no es de solo lectura):

```
unset frutas
```

▼ █ Parte 6: Variables del entorno y del shell

✓ Variables locales

- Definidas normalmente:

```
nombre=Max
```

- Solo existen en el shell actual.
- No se heredan por otros procesos.

✓ Variables de entorno

- Se usan para pasar valores a otros programas o shells hijos.
- Se crean así:

```
export nombre=Max
```

- Ver todas las variables del entorno:

```
env
```

✓ Variables del shell

Son predefinidas por el shell para su funcionamiento.

Variable	Significado
\$PWD	Directorio actual

	DIRECTORIO actual
\$PATH	Rutas donde buscar comandos
\$HOME	Carpeta personal del usuario

- Puedes ver todas las variables del shell con:

```
set
```

▼ Parte 7: Sustitución de variables

Permite usar valores por defecto, hacer asignaciones condicionales, y manejar errores cuando las variables no existen.

1. Valor por defecto (solo se muestra)

```
echo ${nombre:-Invitado}
```

Si `nombre` no está definida → imprime "Invitado"

(No cambia el valor de `nombre`)

2. Asignar valor si está vacía

```
echo ${nombre:=Invitado}
```

Si `nombre` está vacía o no existe → le asigna "Invitado"

3. Error si no está definida

```
: ${nombre:?Debes definir tu nombre}
```

Si `nombre` no está definida → muestra mensaje y termina el script

4. Sustituir solo si está definida

```
echo ${DEBUG:+Modo Depuración Activado}
```

Solo imprime el texto si `DEBUG` está definida

▼ Parte 8: Sustitución con patrones (#, ##, %, %%)

Permite modificar cadenas quitando texto desde el inicio o final, según patrones.

a) Substring (extraer parte de una cadena)

```
 ${param:offset}
 ${param:offset:len}
```

- `offset` → posición inicial (empieza en 0).

- `len` → longitud opcional a extraer.

Ejemplo:

```
msg="abcdef"
echo ${msg:2}      # cdef
echo ${msg:2:3}    # cde
```



b) Longitud de la cadena

```
#${#param}
```

Devuelve la longitud de `param`.

Ejemplo:

```
msg="hola"
echo ${#msg}    # 4
```



Eliminar desde el inicio (`#` y `##`)

```
mensaje="holafeomundo"
```



Eliminar desde el final (`%` y `%%`)

```
archivo="backup_2025.txt"
```



* es un comodín:

- = "cualquier cosa"
- Si no usas , solo coincide exactamente con texto literal

▼ Parte 9: Sustitución de órdenes y aritmética

✓ 1. Sustitución de órdenes

Permite ejecutar un comando y **reemplazar** la expresión por su salida.

```
echo "Hoy es: $(date)"
```

- ◆ Ejecuta `date` y muestra su salida.



Sintaxis válida:

- Moderna: `$(comando)`
- Antigua (menos recomendable): ``comando``

✓ 2. Sustitución aritmética

Permite hacer operaciones matemáticas con enteros:

```
echo $((2 + 3))      # Resultado: 5
```

- ◆ Se usa para sumar, restar, multiplicar, etc.



Operadores comunes:

Operador	Significado
<code>+ - * / %</code>	Básicos
<code>**</code>	Potencia
<code>++ --</code>	Incrementos
<code>== != > < >= <= &&</code>	Comparaciones

💡 `$((expresión)) ≠ $(comando)`

- `$(())` → hace cuentas
- `$()` → ejecuta comandos



🧪 ¿Cómo probarlos?

Puedes usar líneas como estas en la terminal:

```
x=8
echo $((x ** 2))      # Potencia: 64

x=15
((x %= 4))            # Módulo y asignar: x = 3
echo $x

a=3; b=2
echo $((a > b ? 1 : 0)) # Operador ternario: imprime 1
```



▼ █ Parte 10: Entrecomillado (Quoting)

El shell interpreta caracteres especiales como `$`, `*`, `;`, `&`, etc.

El quoting sirve para protegerlos y evitar errores.

I. Backslash (\) → Protege un solo carácter

```
echo Hola\;Mundo
```

- ◆ El ; no se interpreta como "fin de comando"

2. Comillas simples ('') → Protege todo el contenido

```
echo '<-$1250.*>; (update?) [y|n]'
```

- ◆ Nada se interpreta: ni \$, ni *, ni ;, etc.

3. Comillas dobles (") → Protege todo excepto \$, \ y \$(...)

```
echo "$USER tiene \$100 [$(date +%Y-%m-%d)]"
```

- ◆ El \$USER y \$(date ...) sí se interpretan
- ◆ El resto queda protegido

💡 Regla de oro:

Quoting	¿Evalúa \$, \$(...) ?	¿Protege texto?
'texto'	✗ No	✓ Totalmente
"texto"	✓ Sí	✓ Parcialmente
\carácter	✗ No	✓ Solo ese carácter

▼ █ Parte 11: Estructuras de control – if, test, []

Permiten evaluar condiciones y tomar decisiones en los scripts.

Sintaxis básica:

```
if [ condición ]; then
    # instrucciones si la condición es verdadera
else
    # si es falsa
fi
```

Uso de test o []:

Son equivalentes:

```
test "$a" = "$b"
[ "$a" = "$b" ]
```

⚠️ Espacios son obligatorios:

```
[ "$a" = "$b" ] ✓
```

```
["$a"="$b"] X
```

✓ Tipos de condiciones:

◆ a) Archivos

Expresión	Significado
<code>-e archivo</code>	¿Existe el archivo o carpeta?
<code>-f archivo</code>	¿Es un archivo regular?
<code>-d carpeta</code>	¿Es un directorio?
<code>-r archivo</code>	¿Tiene permisos de lectura?
<code>-x archivo</code>	¿Es ejecutable?

◆ b) Cadenas (texto)

Expresión	Significado
<code>-z "\$var"</code>	¿Cadena vacía?
<code>-n "\$var"</code>	¿Cadena NO vacía?
<code>"\$a" = "\$b"</code>	¿Son iguales?
<code>"\$a" != "\$b"</code>	¿Son diferentes?

◆ c) Números

Expresión	Significado
<code>-eq</code>	Igual a
<code>-ne</code>	No igual
<code>-lt , -le</code>	Menor, menor o igual
<code>-gt , -ge</code>	Mayor, mayor o igual

■ 1. Expresiones compuestas: complemento, AND y OR

En Bash puedes combinar condiciones en una misma expresión con:

✓ a) Complemento (negación): `!`

```
if [ ! -z "$nombre" ]; then
    echo "La variable NO está vacía"
fi
```

! niega el resultado de la condición.

✓ b) AND lógico: `&&`

```
if [ "$edad" -ge 18 ] && [ "$pais" = "Peru" ]; then
    echo "Eres mayor de edad y vives en Perú"
fi
```

Ambas condiciones deben ser verdaderas para que se ejecute el bloque.

✓ c) OR lógico: ||

```
if [ "$dia" = "sábado" ] || [ "$dia" = "domingo" ]; then
    echo "Es fin de semana"
fi
```



Se ejecuta si al menos una condición es verdadera.

💡 Alternativa avanzada (dobles paréntesis):

También puedes usar `[[...]]` en lugar de `[...]`, lo cual permite más flexibilidad:

```
if [[ -n "$nombre" && "$edad" -ge 18 ]]; then
    echo "Nombre válido y mayor de edad"
fi
```



2. Sentencia case

Es una forma **ordenada** de hacer muchas comparaciones de valores (como un `switch` en otros lenguajes).

✓ Sintaxis:

```
case $variable in
    valor1)
        # comandos
        ;;
    valor2)
        # comandos
        ;;
    *)
        # comandos si no coincide ningún caso
        ;;
esac
```



📝 Ejemplo:

```
read -p "Ingrese una opción [a/b/c]: " opcion

case $opcion in
    a)
        echo "Elegiste A"
        ;;
    b)
        echo "Elegiste B"
        ;;
    c)
        echo "Elegiste C"
        ;;
    *)
        echo "Opción no válida"
        ;;
esac
```



💡 Notas:

- Los `;;` marcan el fin de cada caso.
- El `El` actúa como "default" (si no coincide con ninguno).

▼ 🟢 Parte 12: Lazos (`while`, `until`, `for`)

Los lazos permiten repetir instrucciones automáticamente.

✓ 1. `while` → mientras la condición sea verdadera

```
contador=1
while [ $contador -le 3 ]; do
    echo "Contador: $contador"
    contador=$((contador + 1))
done
```

- ◆ Repite mientras `$contador` sea menor o igual que 3.

✓ 2. `until` → mientras la condición sea falsa

```
contador=1
until [ $contador -gt 3 ]; do
    echo "Contador: $contador"
    contador=$((contador + 1))
done
```

- ◆ Funciona igual que el anterior, pero con lógica inversa.

✓ 3. `for` → para cada elemento en una lista

```
for fruta in manzana uva pera; do
    echo "Fruta: $fruta"
done
```

- ◆ Imprime cada elemento de la lista.

✓ Contar con `for` y `seq`

```
for i in $(seq 1 5); do
    echo "Número: $i"
done
```

▼ 🟢 Parte 13: Lazos anidados, infinitos, `break` y `continue`

✓ 1. Lazos anidados (`while` dentro de `while`)

Útiles para imprimir estructuras o recorrer matrices.

```

i=0
while [ $i -le 3 ]; do
    j=$i
    while [ $j -ge 0 ]; do
        echo -n "$j "
        j=$((j - 1))
    done
    echo
    i=$((i + 1))
done

```

- ◆ Imprime:

```

0
1 0
2 1 0
3 2 1 0

```

✓ 2. Lazo infinito

```

while true; do
    echo "Infinito..."
done

```

✓ 3. `break` → salir del lazo

```

while true; do
    read -p "¿Salir? (s/n): " respuesta
    [ "$respuesta" = "s" ] && break
done

```

✓ 4. `continue` → saltar a la siguiente iteración

```

for i in $(seq 1 5); do
    [ $i -eq 3 ] && continue
    echo "Número: $i"
done

```

- ◆ Salta el número 3.

▼ ■ Parte 14: Parámetros del script (`$0`, `$1`, `$@`, etc.)

Cuando ejecutas un script con argumentos:

```
bash script.sh uno dos tres
```

Bash los guarda en **variables especiales**:

✓ Parámetros posicionales

Variable	Significado
----------	-------------

<code>\$0</code>	Nombre del script (<code>script.sh</code>)
<code>\$1, \$2, \$3 ...</code>	Argumentos recibidos (<code>uno</code> , <code>dos</code> , ...)

✓ Variables especiales

Variable	¿Qué hace?
<code>\$#</code>	Número de argumentos
<code>\$*</code>	Todos los argumentos como una cadena
<code>\$@</code>	Todos los argumentos como lista individual
<code>\$?</code>	Código de salida del último comando
<code>\$\$</code>	ID del proceso del script actual

📝 Ejemplo:

```
echo "Script: $0"
echo "Argumentos: $*"
echo "Número de argumentos: $"
```

Con:

```
bash ejemplo.sh hola mundo
```

Resultado:

```
Script: ejemplo.sh
Argumentos: hola mundo
Número de argumentos: 2
```

✓ Diferencia entre `$*` y `$@`

Cuando los usas entre comillas:

- `"$*"` → todo como una sola cadena
- `"$@"` → cada argumento como unidad separada

💡 Diferencia entre `$*` y `$@`

Ambos representan todos los argumentos pasados al script o función.

La diferencia solo aparece cuando los pones entre comillas dobles (`" "`).

✓ `$*` — todos los argumentos como una sola cadena

```
#!/bin/bash
for arg in "$*"; do
    echo "ARG: $arg"
```

```
done
```

Si ejecutas:

```
bash script.sh uno dos tres
```

◆ Resultado:

```
ARG: uno dos tres
```

✓ `$@` — todos los argumentos como varios elementos independientes

```
#!/bin/bash
for arg in "$@"; do
    echo "ARG: $arg"
done
```

◆ Resultado:

```
ARG: uno
ARG: dos
ARG: tres
```

⭐ En resumen:

Expresión	¿Cómo lo interpreta el <code>for</code> si hay " " ?
<code>"\$*"</code>	Un solo argumento: <code>"uno dos tres"</code>
<code>"\$@"</code>	Tres argumentos separados: <code>"uno"</code> <code>"dos"</code> <code>"tres"</code>

▼ Parte 15: Funciones en Bash

✓ ¿Qué es una función?

Es un bloque de código reutilizable dentro de un script.

Sirve para organizar mejor el código, evitar repeticiones y modularizar tareas.

✓ Sintaxis:

```
nombre_funcion() {
    # instrucciones
}
```

Y para invocarla:

```
nombre_funcion
```

Ejemplo simple:

```
saludar() {  
    echo "Hola, $1"  
}  
  
saludar Max  
saludar Ana
```

- ◆ Usa `$1` como primer argumento de la función.



Retorno de valores

Forma 1: usando variables globales

```
sumar() {  
    resultado=$(( $1 + $2 ))  
}  
sumar 3 4  
echo "Resultado: $resultado"
```

Forma 2: con `return` (solo valores 0–255)

```
sumar() {  
    return=$(( $1 + $2 ))  
}  
sumar 3 4  
echo $? # Muestra: 7
```

- ⚠ No sirve para textos ni para números grandes.



Puedes usar `$1`, `$2`, `$@`, `$#` dentro de funciones igual que en scripts.

