CHAPTER 5 : FLOW CONTROL IN BASH

BY HANSIKA BHAMBHANEY

OBJECTIVES:

1.Understand how if/else constructs manage decision-making in scripts.

2.Learn to evaluate exit statuses and test conditions.

3.Implement for, while, until, and select loops for iterative operations.

4.Use case statements for multi-conditional branching.

5.Apply control structures to automate and manage flow in shell scripts.

5.1 `if/else` in Bash

The if statement in Bash is a fundamental flow control structure that allows a script to make decisions based on conditions. It determines whether to execute certain commands depending on whether a condition evaluates to true or false.

Syntax of if statement

if condition; then

Statements

elif condition; then

statements

else statements

fi

- **if condition: If the condition is true, the statements inside then will execute.**
- **elif condition: (Optional) If the first condition is false, it checks the next condition.**
- **else: (Optional) If no conditions are true, the else block runs.**
- **fi: Ends the if block.**

EXAMPLE :

```
#!/bin/bash

num=10

if [ $num -gt 5 ]; then
    echo "Number is greater than 5"
elif [ $num -eq 5 ]; then
    echo "Number is exactly 5"
else
    echo "Number is less than 5"
fi
```

OUTPUT:

```
ansika_34@DESKTOP-TQ8I4PM:~$ chmod +x ifelse.sh
ansika_34@DESKTOP-TQ8I4PM:~$ ./ifelse.sh
Number is greater than 5
ansika_34@DESKTOP-TQ8I4PM:~$
```

Explanation:

- num=10 assigns the value 10 to the variable num.
- The if statement checks if num is greater than 5 using -gt.
- elif checks if it's equal to 5 using -eq.
- else handles all other cases (i.e., less than 5).
- fi ends the if block (note: it must be lowercase, not Fi).

Common Conditions

Conditions can check:

- Numeric comparisons (-eq, -ne, -gt, -lt, -ge, -le)
- String comparisons (==, !=)
- File properties (-f for files, -d for directories, -r, -w, -x for permissions)

EXAMPLE 2 :

```
#!/bin/bash



if [ -f "data.txt" ]; then

    echo "File exists!"

else

    echo "File does not exist!"

fi
```

Explanation:

- `-f "data.txt"` checks if a regular file named `data.txt` exists in the current directory.
- If true, it prints `"File exists!"`.
- Otherwise, it prints `"File does not exist!"`.

5.2 EXIT COMMAND

1. The `exit` command in Bash is used to **terminate a script**.
   Its syntax is simple:
   `exit [n]`
2.
   - `n` is an optional **exit status number**.

3. 🔹 The **exit status** is returned to the **parent shell or calling script**.

4. 🔹 It acts as a signal to indicate whether the script ended **successfully** or **with an error**.

5. 🔹 If you **do not provide a value** (i.e., just write `exit`), Bash will return the **exit status of the last command executed**.

6. 🔹 **Exit status values**:

   - `0` → Success

   - `1` or higher → Error or custom code

Example:

```
#!/bin/bash

echo 'Hello, World!'

exit 0
```

7. 🔹 In the above example:
    - The script prints **"Hello, World!"** to the console.
    - Then it ends with `exit 0`, which signals **successful completion**.
8. 🔹 `exit` is useful when:
    - If you want to **stop a script early** based on a condition.
    - You need to let another program or user know if the script **succeeded or failed**.
9. 🔹 The returned exit status can be checked using:`echo $?`
10. 🔹 `exit` plays a key role in **error handling**, **script chaining**, and **system-level automation**.

#!/bin/bash

```
echo 'Starting the script'
exit 0
echo 'This will not be printed'
# Output:
# 'Starting the script'
```

In this script, we first print "Starting the script", and then execute the `exit 0` command.
This command immediately terminates the script, so the line `echo 'This will not be printed'` is never executed.

The `exit 0` command signals a successful execution of the script.
In Bash and other Unix-like systems, the number `0` represents success, while values from `1` to `255` are typically used to indicate different types of errors. This is part of the exit status concept in Unix systems, which we'll explore further in the upcoming sections.

This feature of the `exit` command is extremely useful for error handling in scripts.
By checking the exit status of commands, we can make our scripts more robust, reliable, and responsive to different conditions and failures.

While `exit` is the standard way to terminate a Bash script, there are alternative methods such as:

- `return` (used inside functions)
- `kill` (used to terminate scripts using their Process ID or PID)
  These alternatives may be more appropriate in specific scenarios, depending on the script's structure and purpose.

5.2.1 CONDITION TESTS

Test Type	Example	Meaning
Integer Comparison	`[ $a -eq $b ]`	a == b
String Comparison	`[ "$a" == "$b" ]`	string equality
File Exists	`[ -f file.txt ]`	true if file.txt is a regular file
File is Directory	`[ -d myfolder ]`	true if myfolder is a directory

Condition tests help you make decisions in Bash scripting.
The main tool for testing conditions is `[ ... ]` or `[[ ... ]]`, which return an exit status to indicate whether a condition is true (exit status 0) or false (non-zero).

- 1. The [...] Construct

This is the traditional test syntax used in older versions of the shell.

It checks a condition and returns:

- 0 if the condition is true

- Non-zero if the condition is false

Syntax:

```
if [ condition ]; then
    statements
fi
```

Example – Checking If a File Exists:

```
if [ -f "data.txt" ]; then
    echo "File exists!"
fi
```

Here, -f checks if "data.txt" is a regular file.
If it exists, the condition is true and the message is displayed.

Example – Comparing Numbers:

```
num=10
if [ $num -gt 5 ]; then
    echo "Number is greater than 5"
fi
```

In this case, -gt means "greater than".
If num is greater than 5, the exit status is 0, and the message is printed.

🔹 2. The [[...]] Construct (Newer Version)

[[...]] is an improved test syntax introduced in Bash.
It helps avoid common issues like word splitting and wildcard expansion.

Syntax:

```
if [[ condition ]]; then
    statements
fi
```

Example – Checking If a String Matches:

```
name="Hansika"
if [[ $name == "Hansika" ]]; then
        echo "Name matches!"
fi
```

String comparisons are safer with [[...]] because:

- It avoids unexpected filename expansion from characters like * or ?
- It prevents accidental word splitting
- It supports string equality using == without extra escaping

EXAMPLE: ELIF

```bash
#!/bin/bash

echo "Enter filename:"
read fname

if [ -f "$fname" ]; then
    echo "It's a regular file."
elif [ -d "$fname" ]; then
    echo "It's a directory."
else
    echo "File does not exist."
fi
```

5.2.2 Using the 'return' Command

The `return` command is used to exit a shell function. Like the `exit` command, it can also provide a return status. However, unlike `exit`, it does not terminate the entire shell or script where it is called. Instead, it only exits the current function.

```
#!/bin/bash
function hello() {
  echo 'Hello, World!'
```

```
  return 0
}
hello
echo 'This will be printed'
# Output:
# 'Hello, World!'
# 'This will be printed'
```

In this script, we define a function called `hello` that prints ‘Hello, World!’ and then returns with a status of 0. After calling the function, we print ‘This will be printed’.

Unlike with `exit`, this line is executed because `return` only exits the function, not the entire script.

5.2.3 Combinations of Exit Statuses

In Bash, every command returns an exit status after execution.

Exit Status in Bash:

- `0` → Success (command ran without errors)
- Non-zero (`1`, `2`, etc.) → Failure (an error occurred)
  These exit statuses are especially powerful when used with logical operators to control the flow of commands.

🔸 **1. Logical AND (`&&`)**

Syntax:

```
command1 && command2
```

Meaning:

- Run `command1`
- If it succeeds (exit status `0`), then run `command2`

- If `command1` fails, `command2` is skipped

Example:

```bash
grep "start" logfile.txt && echo "Keyword found in log."
```

Explanation:

- If `"start"` is found in `logfile.txt`, the message will be printed.
- If not, the second command won't run.

2. Logical OR (`||`)

Syntax:

```
command1 || command2
```

Meaning:

- Run `command1`
- If it fails (non-zero exit status), then run `command2`
- If `command1` succeeds, `command2` is skipped

🔧 **Example:**

```bash
grep "error" logfile.txt || echo "No errors found."
```

Explanation:

- If `"error"` is not found, the echo command runs.
- If it is found, the script proceeds without printing the message.

3. Logical NOT (`!`)

Syntax:

```
! command
```

Meaning:

- Reverses the exit status of a command:
  - If the command fails, `!` makes it a success
  - If the command succeeds, `!` makes it a failure

EXAMPLE:

```
if ! grep "GENE" seq.txt
then
  echo "GENE is NOT in the list."
fi
```

Explanation:

- If `"GENE"` is **not** found, the `grep` fails, and `!` makes it pass the `if` condition, so the message is printed.

5.2.4 File Attribute Checking in Bash

In Bash, we can check if a file or directory exists, and examine its type, permissions, or ownership using file test operators inside conditional statements (if [...]).
These checks are helpful in writing robust scripts that don't break when files are missing, unreadable, or unwritable.

Operator	True if...
-a *file*	*file* exists
-d *file*	*file* exists and is a directory
-e *file*	*file* exists; same as - **a**
-f *file*	*file* exists and is a regular file (i.e., not a directory or other special type of file)

Operator	True if...
-r *file*	You have read permission on *file*
-s *file*	*file* exists and is not empty
-w *file*	You have write permission on *file*
-x *file*	You have execute permission on *file*, or directory search permission if it is a directory
-N *file*	*file* was modified since it was last read
-O *file*	You own *file*
-G *file*	*file* 's group ID matches yours (or one of yours, if you are in multiple groups)
file1 **-nt** *file2*	*file1* is newer than *file2* [6]

Example 1: Basic check if a file exists

```
if [ -e "report.txt" ]; then
    echo "The file exists."
 else
    echo "The file does not exist."
 fi
```

In this example, -e checks whether the file named "report.txt" exists in the current directory.

- Example 2: Check if a directory is accessible

```
if [ -d "$1" ] && [ -x "$1" ]; then
    echo "$1 is a directory and you have permission to access it."
else
    echo "$1 is either not a directory or not accessible."
fi
```

This script checks if the first argument provided is a directory and whether the user has execute (search) permission.

- Combining Expressions
  - Use && to combine conditions with AND
  - Use || to combine conditions with OR
  - Use ! to negate a condition
  - Use \(and \) to group conditions (must be escaped)

Example:

```
if [ -n "$folder" ] && [ -d "$folder" -a -x "$folder" ]; then
    echo "The directory is valid and accessible."
fi
```

This checks if the variable folder is not empty, and whether it's a directory that the user can search.

- Real-Life Example: File Details Checker (filecheck.sh)

```
if [ ! -e "$1" ]; then
    echo "The file $1 was not found."
    exit 1
 fi

if [ -d "$1" ]; then
    echo -n "$1 is a directory, and you may "
    if [ ! -x "$1" ]; then
       echo -n "not "
    fi
    echo "enter it."
 elif [ -f "$1" ]; then
    echo "$1 is a regular file."
 else
    echo "$1 is a special type of file."
 fi

if [ -O "$1" ]; then
    echo "You are the owner of the file."
 else
    echo "You do not own the file."
 fi

if [ -r "$1" ]; then
    echo "You can read this file."
 fi

if [ -w "$1" ]; then
    echo "You can write to this file."
 fi
```

```
if [ -x "$1" ] && [ ! -d "$1" ]; then
    echo "You can execute this file."
 Fi
```

5.2.5 Integer Conditionals in Bash

The shell also provides a set of arithmetic tests. These are different from character string comparisons like < and >, which compare lexicographic values of strings,[8] not numeric values. For example, "6" is greater than "57" lexicographically, just as "p" is greater than "ox," but of course the opposite is true when they're compared as integers.

Test	Comparison
-lt	Less than
-le	Less than or equal
-eq	Equal

Test	Comparison
-ge	Greater than or equal
-gt	Greater than
-ne	Not equal

1. String vs Integer Comparison

In Bash, comparisons can be either **string-based** or **integer-based**.

- **String comparisons** (using `<`, `>`) compare values based on **lexicographic (dictionary) order**.
- **Integer comparisons** compare values **numerically**.

Example – String (Lexicographic) Comparison:

```
if [ "6" > "57" ]; then
    echo "6 is greater than 57 (string)"
 else
    echo "6 is not greater (string)"
 fi
```

Output: 6 is greater than 57 (string)
Explanation: Lexicographically, "6" comes after "5", so it's considered greater.

Example – Integer Comparison:

```
if [ 6 -gt 57 ]; then
    echo "6 is greater (integer)"
 else
    echo "6 is not greater (integer)"
 fi
```

Output: 6 is not greater (integer)
Explanation: Numerically, 6 is less than 57.

3. Important Notes

- The `<` and `>` operators used for **string comparisons** must be **escaped** using a backslash (e.g., `\<`, `\>`) when using `[ ... ]` to avoid shell misinterpretation.

Example: ["$a" < "$b"]

- The `test` command is **equivalent** to `[ ... ]` in functionality.

Example: test "$a" -eq "$b" is the same as ["$a" -eq "$b"]

EXAMPLE:

Compare two numbers:

```
a=10
 b=20

if [ $a -lt $b ]; then
    echo "$a is less than $b"
 fi
```

Efficient version using double parentheses:

```
if (( a < b )); then
    echo "$a is less than $b (efficient check)"
 Fi
```

5.3 FOR LOOP IN BASH

In Bash scripting, the `for` loop is one of the most powerful and frequently used constructs.

It allows you to repeat a block of code multiple times, making automation, iteration over lists, arrays, and numeric ranges easy and efficient.

1. Simple for Loop (Fixed List)

This loop iterates over a fixed list of values such as letters, words, or numbers.

Example:

```
for n in a b c
do
   echo $n
done
```

Explanation:

- The loop will run 3 times.
- Each time, the variable `n` takes a new value from the list: a, then b, then c.
- The `echo` statement prints the value of `n` in each iteration.

🔹 **2. Range-Based for Loop**

You can use **curly braces** `{}` to loop over a range of numbers.

Example:

```
for i in {1..5}
do
   echo $i
done
```

Explanation:

- This loop runs from 1 to 5.
- The loop automatically increments by 1 in each cycle.

To change the step size, you can use:
for i in {1..10..2} (from 1 to 10, step 2)

🔹 **3. Array Iteration with for Loop**

You can loop through an array using the `[@]` symbol to access all elements.

Example:

```
sports=("football" "cricket" "hockey")
```

```
for game in "${sports[@]}"
do
   echo $game
done
```

Explanation:

- The loop goes through each item in the `sports` array.
- Each element (e.g., football, cricket, hockey) is assigned to the variable `game` during its turn.

🔹 **4. C-Style for Loop**

Bash also supports a C-like syntax, useful when you need **numeric control** like initialization, condition checking, and incrementing in one line.

Example:

```
n=7
for (( i=1; i<=n; i++ ))
do
   echo $i
done
```

Explanation:

- `i` starts at 1.
- The loop continues as long as `i` is less than or equal to `n`.
- `i` increases by 1 in each cycle.
- Prints numbers from 1 to 7.

This style is efficient for numeric operations and gives you more control than fixed lists.

5.3.1 CASE Statement in Bash

The case statement in Bash is used for multi-way decision-making.
It allows you to compare a variable or expression against multiple possible values or patterns and execute different blocks of code based on the match.

This construct is cleaner and easier to manage than writing multiple if-else statements.

🔹 **Key Features of Case Statements**

- Works like `switch-case` in other languages (like C or Java), but simpler.
- No need for a `break` statement — Bash automatically stops after the first match.
- Supports multiple patterns using the pipe symbol `|`.
- Includes a default case using `*`, which runs if no pattern matches.
- Ends with the keyword `esac` (which is `case` spelled backward).

🔹 **Syntax of a Case Statement**

```
case EXPRESSION in
    pattern1)
        statements
        ;;
    pattern2)
        statements
        ;;
    ...
    *)
        default statements
        ;;
esac
```

🔹 **How It Works**

1. The `EXPRESSION` is evaluated once.
2. The result is compared against each pattern.
3. When a match is found, the corresponding block of code runs.
4. After the match, Bash exits the case block — no further checking.
5. If no patterns match, the default block marked by `*` is executed.

🔹 **Example 1: Department Description**

```
DEPARTMENT="Computer Science"
 case $DEPARTMENT in
    "Computer Science")
       echo "Computer Science"
       ;;
    "Electrical Engineering" | "EEE")
       echo "Electrical Engineering"
       ;;
    "Information Technology" | "Electronics and Communication")
       echo "IT or E&C"
       ;;
    *)
       echo "Invalid Department"
       ;;
 esac
```

Explanation:

- Matches the variable DEPARTMENT with multiple possible department names.
- If it matches "Computer Science", it prints that.
- If none match, it prints "Invalid Department".

🔹 **Example 2: Case Inside a For Loop**

```
DEPARTMENTS=("IT" "EEE" "Mechanical")
 for dept in "${DEPARTMENTS[@]}"
 do
    case $dept in
       "IT") echo "Information Technology" ;;
       "EEE") echo "Electrical Engineering" ;;
       *) echo "Other Department" ;;
    esac
 done
```

Explanation:

- Iterates over an array.
- Matches each item to print the corresponding department.

🔹 **Example 3: User Input for Yes/No**

```
echo "Are you a student? [yes or no]"
 read response
 case $response in
    "yes" | "Yes" | "Y" | "y")
       echo "Yes, I am a student."
       ;;
    "no" | "No" | "N" | "n")
       echo "No, I am not a student."
       ;;
    *)
       echo "Invalid input."
       ;;
 esac
```

Explanation:

- Takes user input and matches different variations of "yes" or "no".
- Responds accordingly.
- Default case handles incorrect input.

🔹 **Example 4: Process Signal Handler**

If the script receives a signal number and a PID (Process ID), it can send specific system signals.

Example:

```
case "$1" in
    1) send SIGHUP to $2 ;;
    2) send SIGINT to $2 ;;
    3) send SIGQUIT to $2 ;;
    *) echo "Unknown signal" ;;
 esac
```

Explanation:

- Based on the signal number (first argument), the script sends a corresponding signal to the process with the given PID (second argument).
- Helps in managing running processes.

Advantages of Using Case Statements

- Cleaner than multiple `if-elif-else` chains.
- Easier to read and maintain.
- Supports pattern matching using wildcards.
  Ideal for handling user inputs, menu options, or grouped choices.

5.4 `select` Statement in Bash

The `select` command in Bash is used to create simple menus in scripts. It allows users to choose from a list of options, making the script more interactive and user-friendly.

🔹 **Key Features of `select`**

- Displays a numbered list of options.
- Waits for the user to enter a number corresponding to their choice.
- Stores the selected value in a variable.
- Commonly used with `case` statements for processing the selection.
- Automatically provides a prompt for selection.
- Runs in a loop by default (use `break` to exit).

🔹 **Syntax**

```
select variable in list
do
    statements
done
```

🔹 **How It Works**

1. The `select` command displays each item in the list with a number.
2. The user types the number of their choice.
3. The selected value is stored in the specified variable.
4. The loop body executes based on the selection.
5. Typically combined with a `case` block to handle choices.
6. The loop continues until you manually `break` it.

🔹 **Example: Simple Menu**

```
options=("Play" "Pause" "Stop" "Exit")
 select choice in "${options[@]}"
 do
    case $choice in
       "Play") echo "Playing music" ;;
       "Pause") echo "Music paused" ;;
       "Stop") echo "Music stopped" ;;
       "Exit") echo "Exiting menu"; break ;;
       *) echo "Invalid choice" ;;
    esac
 done
```

Explanation:

- Displays a menu with Play, Pause, Stop, and Exit.
- User inputs the number corresponding to a choice.
- The script responds accordingly.
- Exits when "Exit" is selected.

✅ **Advantages of `select`**

- Great for creating simple interactive menus.
- No need to manually number or format options.
- Easy to integrate with `case` for clean control flow.
- Useful for scripts where user interaction is required.

EXAMPLE DNA SEQUENCE:

```
#!/bin/bash



# Define the DNA sequence

DNA="ATGCTAGCTAGGCTA"



# Set custom prompt for select menu

PS3="Choose an operation on the DNA sequence: "



# Display menu options

select operation in "Show sequence" "Length of sequence" "Count
nucleotides" "Complement" "Quit"

do

  case $operation in



    "Show sequence")

      echo "DNA sequence: $DNA"

      ;;



    "Length of sequence")

      echo "Length of the sequence: ${#DNA}"

      ;;
```

```
    "Count nucleotides")
      echo "Nucleotide counts:"
      echo "A: $(echo "$DNA" | grep -o "A" | wc -l)"
      echo "T: $(echo "$DNA" | grep -o "T" | wc -l)"
      echo "G: $(echo "$DNA" | grep -o "G" | wc -l)"
      echo "C: $(echo "$DNA" | grep -o "C" | wc -l)"
      ;;

    "Complement")
      complement=$(echo "$DNA" | tr 'ATGC' 'TACG')
      echo "Complement: $complement"
      ;;

    "Quit")
      echo "Exiting the program..."
      break
      ;;

    *)
      echo "Invalid choice. Please select a valid option."
      ;;
  esac
done
```

OUTPUT :

```
ansika_34@DESKTOP-TQ8I4PM:~$ chmod +x DNA.sh
ansika_34@DESKTOP-TQ8I4PM:~$ ./DNA.sh
1) Show sequence
2) Length of sequence
3) Count nucleotides
4) Complement
5) Quit
Choose an operation on the DNA sequence: 2
Length of the sequence: 15
Choose an operation on the DNA sequence: 1
DNA sequence: ATGCTAGCTAGGCTA
Choose an operation on the DNA sequence: 3
Nucleotide counts:
A: 4
T: 4
G: 4
C: 3
Choose an operation on the DNA sequence: 4
Complement: TACGATCGATCCGAT
Choose an operation on the DNA sequence: 5
Exiting the program...
```

5.5 `while` and `until` Loops in Bash

- 1. while Loop

The while loop runs as long as the condition is true.
It checks the condition before each iteration. If the condition becomes false, the loop stops.

Syntax:

```
while [ condition ]; do
    commands
done
```

How it works:

- If the condition is true, the loop body executes.
- After each run, the condition is re-checked.
- When the condition becomes false, the loop ends

Example:

```
count=1
 while [ $count -le 5 ]; do
    echo $count
    count=$((count + 1))
 done
```

This loop prints numbers from 1 to 5.

- 2. until Loop

The until loop is the opposite of the while loop.
It runs the loop until the condition becomes true.

Syntax:

```
until [ condition ]; do
    commands
 done
```

How it works:

- If the condition is false, the loop body runs.
- After each iteration, it checks again.
- Once the condition becomes true, the loop ends.

Example:

```
count=1
 until [ $count -gt 5 ]; do
    echo $count
    count=$((count + 1))
 done
```

FEATURE	`WHILE` LOOP	`UNTIL` LOOP
Condition True	Loop continues	Loop stops
Condition False	Loop stops	Loop continues
Use Case	When you want to repeat **while true**	When you want to repeat **until true**

Infinite Loop Example Using `while`

```
while true; do
    echo "Press CTRL+C to stop."
    sleep 1
done
```

This loop runs forever unless manually stopped.

1.Example Using `while` Loop

Read a FASTA file line by line and count sequences

```
#!/bin/bash

count=0

while read line; do
  if [[ $line == ">"* ]]; then
    count=$((count + 1))
  fi
done < sequences.fasta

echo "Number of sequences in the file: $count"
```

OUTPUT :

```
ansika_34@DESKTOP-TQ8I4PM:~$ nano sequences.fasta
ansika_34@DESKTOP-TQ8I4PM:~$ ./count.sh
Number of sequences in the file: 3
```

2.Example Using until loop

```
#!/bin/bash

DNA="ATGCGTACTGAAGCTAGTGA"
codon=""
index=0

until [[ $codon == "TAG" || $codon == "TAA" || $codon == "TGA" ]]; do
  codon=${DNA:$index:3}
  echo "Reading codon: $codon"
  index=$((index + 3))
done

echo "Stop codon encountered: $codon"
```

Explanation:

- The `until` loop keeps reading the DNA string 3 bases (1 codon) at a time.
- It continues **until** it finds a stop codon.
- The loop then ends and reports the stop codon found.

SUMMARY

Chapter 5: Flow Control in Bash explains how to manage the execution path of Bash scripts using various conditional and looping structures. It begins with the `if/else` construct, which allows decisions based on conditions like numeric comparisons, string matching, and file checks. The chapter introduces the `exit` command to terminate scripts and return status codes, which help in signaling success or failure. Condition tests using `[ ... ]` or `[[ ... ]]` are discussed for evaluating expressions. The `return` command is explained for use within functions, along with logical operators like `&&`, `||`, and `!` for combining and negating conditions. File attribute checking using operators like `-f`, `-d`, `-r`, and `-x` is also covered. The

chapter differentiates between string and integer comparisons and introduces arithmetic evaluation using `(( ... ))`. Looping structures such as `for`, `while`, and `until` are presented, along with real-world bioinformatics examples like reading a FASTA file line-by-line or finding a stop codon in a DNA sequence. The `case` statement is explained as a cleaner alternative to multiple if-else branches, allowing pattern matching and multi-way decision-making. Finally, the `select` loop is introduced for creating interactive menus, enhancing user interaction in scripts