**Perl**

**Introduction** :

Perl , stands for Practical Extraction and Report Language, created by Larry Wall in late 1980’s.

1. Text Processing – if you need to manipulate , search , or transform large amount of text, Perl is incredibly efficient at it.
2. System Administration - Used to automate tasks, manage servers , and interact with OS.
3. Web Development – Perl it was a major player in early web development, especially with CGI( Common Gateway Interface) scripts.
4. Bioinformatics – Because of its strong text processing capabilities, Perl is also used in biological research for analysing genomic data.
5. Perl is a Interpreted language that supports Object Oriented Programming.(We can write perl code In oops way like classes , objects, inheritance and polymorphism )

Few Key Points :

1. **Powerful Text Processing**: This is one of Perl's superpowers. It's incredibly good at handling and manipulating text, making it a favourite for working with log files, data cleaning, and reporting. [ **Regular Expression** allow u to give flexible and precise patterns to search for, match and manipulate text]
2. **Flexibility & Versatility**: As we discussed, it's used for a wide range of tasks, from small scripts to complex applications. It allows you to approach problems in many different ways.
3. **Rich Ecosystem (CPAN):** The Comprehensive Perl Archive Network (CPAN) is a huge repository of modules (pre-written code) that can save you a lot of time and effort. Need to connect to a database? There's a CPAN module for that. Need to work with dates? There's a CPAN module for that!
4. **Conciseness**: You can often achieve a lot with very little code in Perl, which can make it very efficient for quick scripting tasks.
5. **Open Source & Community**: Perl is free to use and has a vibrant global community that contributes to its development and supports new learners.

**First Program :**

**#!/usr/bin/perl –** This is called a “shebang” or “hash-bang” line. In linux/unix it tells Operating System which interpreter to use to run the script. In this case, it’s telling it to use Perl interpreter located at /usr/bin/perl.

Commands to compile and run the file :

chmod +x test.pl

./test.pl

**Basic Syntax and Data Types:  
 Scalar variable** ($) –

1. holds single piece of data , can be number , floationg point number, string , Boolean , anything.
2. It begins with $ sign.
3. $name= “Sanjai”;
4. $age= 20;
5. $pi=3.14 ;
6. $message =’hello everyone’; (can use single quotes also for string)
7. $is\_active = 1; (1 – true , 0 – false)
8. Scalar variable is like a box. We a put one thing at a time. If put something new old will get replaced.

**Arrays** (@) –

1. [@name=(“Alice”,”Bob”,”Charlie”)](mailto:1.@name=(); #declaration
2. It has index like normal arrays only.
3. $name[0] returns “Alice”.
4. When accessing single element from array use $ sigil. Because it’s a scalar.
5. Stores in ordered list.
6. my $arr\_count=@name; # will give output as size of array
7. my $val=scalar(@name); # will give output as size of array

**Hashes** (% sigil ) –

1. Dictionaries – unordered collection of key-value pairs.
2. %ages=(“Alice” => 30,”Bob” => 25); #declaration
3. $ages{“Alice”} return 30.
4. $ is used for fetching single value.
5. $ages{“Jon”}=40;
6. $ages{$key}=$value; #user input
7. my $hash\_count = %ages; # will give output as size of hash
8. my $hash\_count = scalar(%ages); # will give output as size of hash

**Operators and Control Structures :**

1. **Arithmetic Operators** –   
     1. + (addition)  
     2. - (subtraction)  
      3. \* (multiplication)  
     4. / (division)  
      5. \*\* (exponentiation, e.g., 2 \*\* 3 is 2 to the power of 3, which is 8)  
       6. % (modulus, gives the remainder of a division)  
   Example:  
   $a = 10;  
   $b = 3;  
   $sum = $a + $b;     # $sum will be 13  
   $remainder = $a % $b; # $remainder will be 1 (10 divided by 3 is 3 with 1 left over)
2. **String Operators** :  
       1) . (concatenation, joins two strings together)  
       2) x (repetition, repeats a string multiple times)  
      Example:  
      $greeting = "Hello";  
    $name = "World";  
    $full\_message = $greeting . " " . $name . "!"; # $full\_message will be "Hello World!"  
    $stars = "\*" x 5; # $stars will be "\*\*\*\*\*"
3. **Comparison Operators:** These are super important for if-else statements because they compare two values and return true or false.  
       **For Numbers:**  
         == (equal to)  
         != (not equal to)  
        < (less than)  
       > (greater than)  
         <= (less than or equal to)  
         >= (greater than or equal to)

<=> (Spaceship ) [Return -1 , 0 ,1 ]  
   **For Strings (important distinction!):**  
      eq (equal to)  
      ne (not equal to)  
      lt (less than - alphabetically)  
      gt (greater than - alphabetically)  
      le (less than or equal to - alphabetically)  
      ge (greater than or equal to - alphabetically)

cmp (compare 2 strings) [Returns -1,0, 1 (lexically)]

**Example (Numerical):**  
$x = 10;  
$y = 5;  
$is\_equal = ($x == $y); # $is\_equal will be false (or 0 in Perl's numeric context)  
$is\_greater = ($x > $y); # $is\_greater will be true (or 1 in Perl's numeric context)

**Example (String):**  
$city1 = "Delhi";  
$city2 = "Mumbai";  
$are\_same = ($city1 eq $city2); # $are\_same will be false  
$is\_before = ($city1 lt $city2); # $is\_before will be true (alphabetically "Delhi" comes before "Mumbai")

* The reason Perl has separate operators (== vs eq) is because the way computers store and compare numbers is fundamentally different from how they store and compare text. Perl needs to know which type of comparison to perform.
* Perl will treat the string as **zero** when compared or used with number or using it with ==.
* == for numbers
* eq for strigs
* In Perl ,true is typically represented by any non-zero number or non-empty string.
* False is typically represented by 0,undef (undefined value), or empty string “”.
* , (comma) used to separate in print statements.
* ; semicolon end of statement

1. **Ternary operator** ?: :

* The ternary operator is a concise form of an if-else condition used to evaluate a condition and return one of two values.
* Syntax - CONDITION ? TRUE\_RESULT : FALSE\_RESULT;

**Precedence and Parentheses**

* Ternary has lower precedence than most operators.
* Always use parentheses around ternary expressions to avoid unexpected behaviour.

**Truthiness in Perl**

* False: 0, '0', '', undef, 0.0
* True: Everything else

**Nested Ternary**

* Nested ternary is allowed but reduces readability.
* Example:
* my $grade = 85; my $result = ($grade >= 90) ? "A" :  
   ($grade >= 80) ? "B" :  
   ($grade >= 70) ? "C" :  
   ($grade >= 60) ? "D" : "F";

**Using do {} Blocks**

* Use do {} when multiple statements are needed in either branch.
* Example:
* my $val = ($x > 0) ? do {  
   print "Positive\n"; $x;   
  } :  
   do {   
  print "Non-positive\n"; -$x;   
  };

**Real-World Examples**

* my $max = ($a > $b) ? $a : $b;
* print ($logged\_in ? 'Welcome' : 'Log in');
* my $user = ($input ne '') ? $input : 'guest';

**Operator Precedence**

* ?: is lower than most operators.
* Use parentheses to control evaluation order.

**Alternatives**

* **Logical OR**: my $val = $x || 'default';
* This assigns $val to the value of $x **if $x is truthy**.
* If $x is **false** (like 0, '', undef, '0'), then 'default' is used instead.
* **Defined OR**: my $val = $x // 'default'
* This assigns $val to $x **if $x is defined**, even if it's a false value like 0 or ''.
* Only if $x is **undef**, the 'default' is used.

**Summary**

* Yes Returns a value
* No Cannot assign to ternary result
* Yes Supports nesting with care
* Yes Works well in print/return
* Yes Accepts do {} blocks

1. **Logical Operators :** These Operators combine or negate Boolean(true/false) expressions.

* and
* or
* not
* && (Short-circuit AND) Same as and but with different precedence.
* || (Short-circuit OR)
* ! ( not )

1. Both and , or then &&, || perform the same operation means also there is some difference. That is precedence && , || have higher precedence compared to and , or.

2. && , || used for conditional expressions. If ($x > 10 && $y <20)

3. and , or used for control flow . ( open my $fh, ‘<’ , ‘file.txt’ or die “cannot open file : $!”;

4. The difference between and, or, not AND &&, || , ! is only the precedence.

1. **Assignment Operator:**

* +=
* -=
* \*=
* /=
* ++
* --
* . (Concatenate and assign $x= $x . $y; ) [ .=]
* x= (Repetition and assign $x= $x x $y ; ) [ x= ]

1. **Binary operators :**

* & (Bitwise and)
* | (Bitwise or)
* ^ (Bitwise xor)
* ~ (bitwise not)
* << (Left shift )
* >> (Right shift

1. **File Test Operators :** Used for checking properties of files. They are unary operators.

* -e ( Does file exists ? )
* -f ( Is it a plain File ? )
* -d ( Is it a directory ?)
* -r (Is it readable ?)
* -w (Is it writeable ?)
* -x (Is it executable ? )
* -s (What is the size of the file ?)

**Example :**

my $file = “my\_script.pl”;

if (-e $file){ print “$file Exists \n” ; }

**Conditional Statements :**

**If-else :**

If(){}

elsif(){}

else{}

**given/when** – same like switch :

use strict;  
use warnings;  
use feature 'switch'; # Enables 'given'/'when'  
my $fruit = "banana";  
given ($fruit) {  
    when ("apple") {  
        print "It's an apple.\n";  
    }  
    when ("banana") {  
        print "It's a banana.\n";  
    }  
    when ("cherry") {  
        print "It's a cherry.\n";  
    }  
    default { # Similar to 'default' in switch  
        print "Unknown fruit.\n";  
    }  
}  
# Output: It's a banana.

* Automatic brake – no need to explicitly give break statement unlike c/java.
* Same like switch only. Used in Perl after version 5.10.
* Must add use feature ‘switch’; to use given/when.

**Loops :**

1. **For loop** –

for(initialization; condition ; increment/decrement) {

# code to be executed in each iteration

}

for (@arr){

print $\_,”\n”;

}

Output :  
prints everything from arr. If we did not give variable name it will default use $\_.

1. **While loop** – runs until condition becomes false

while(condition){ #code }

1. **Until loop** – It runs until the condition becomes true.

until(condition){ #code }

Example:

my $guess = 0;  
my $secret\_number = 7;  
until ($guess == $secret\_number) {  
    print "Guess the number (1-10): ";  
    $guess = <STDIN>;  
    chomp $guess;  
    print "Nope! Try again.\n" unless $guess == $secret\_number; # A neat Perl trick!  
}  
print "You guessed it! The number was $secret\_number!\n" ;

1. **do….while Loop :**

do{

#code

} while(condition) ;

1. **For Each loop :**
2. foreach my $item (@list){ #code }
3. for my $item (@list){ #code }
4. for (@arr){

print $\_,”\n”;

}

**Access Modifiers :**

**my Keyword :**

1. In Perl, the my keyword controls the scope (visibility and lifetime) of variables, making them accessible only within the specific block of code where they are declared. It's about preventing variable name clashes and keeping variables localized to where they are needed.
2. Like private in java. Available only within the class.
3. It helps to separate a variable from different scopes.
4. If we did not use my then it is a global variable.

**our Keyword** : (Package Variables - Explicitly "public" for variables)

1. The our keyword declares a package variable that is explicitly visible throughout the current package (which can span multiple files in a module) and can be accessed from other packages by using its fully qualified name (e.g., $MyPackage::my\_variable).
2. Effectively: These are like static public fields in Java, meant for global access within a specific logical grouping.
3. When to use: Rarely needed for most script-level variables. More often used for configuration variables or when you intentionally want a truly global variable that can be accessed by name.

**User Input :**

**<STDIN> -**  to read the user input.

my $name=<STDIN>;

**Note** :  
 1. use **chomp $name;** after getting user input because if we did not give it then it will take the newline also as input.

2. If we want to access single value use $ sigil, from arrays, hash.

3.chomp is smart it only removes the newline

3. chop() – which simply removes the last character of a string, regardless of what it is.

4. **chop $name;**

**Note:**

**use strict; :** Catches typos and variable misuse directly, preventing silent bugs. It makes Perl less forgiving, which is good!. If u use a undefined name some where it will silently create a new variable in global space to prevent that use strict. **use warnings; :** Points out suspicious code that might cause problems, helping you debug faster and write cleaner code.

**Arrays** (@) –

1. [@name=(“Alice”,”Bob”,”Charlie”)](mailto:1.@name=(); #declaration
2. It has index like normal arrays only.
3. $name[0] returns “Alice”.
4. When accessing single element from array use $ sigil. Because it’s a scalar.
5. Stores in ordered list.
6. my $arr\_count=@name; # will give output as size of array
7. my $val=scalar(@name); # will give output as size of array

**Insert user input into array** :

1. **push** :

* syntax – push @array,\_name,elemet1,element2…..;
* stores in end of the array

1. **unshift** :

* syntax – unshift @array,\_name,elemet1,element2…..;
* stores in the beginning.

1. **Direct Assignment to an index** :

* Syntax - $array\_name[index]=$value;
* If index already exists, old value replaced with new value.
* If the index is beyond current size of the array, Perl will automatically extend the array to that size. Any “gaps” between the previous and last element and then new element’s index will be filled with undef (perl’s undefined value).

1. **Array Slicing** [Accessing / Assigning to multiple elements at once ] :

* Syntax - @array\_name[index1,index2]
* my @new\_arr=@array\_name[index1,index2] #fetching from one array and storing it in another array
* @arr[1,3]=(25,35); # assigning this value to index 1 and 3.
* print @arr[1,3]; # 25,35 will be the output
* @arr[1..8]=(1,2,3,4,5,6,7,8); #assigning for a range.
* If there a already some value and if u give index more than the size then perl will increase the size till the index2.
* .. (double dot ) means range operator in perl.

1. **Splice [Insert at specific position]-**

* **Syntax to insert** - @array\_name , OFFSET , LENGTH ,LIST;
* OFFSET – the starting index where we need to perform the operation
* LENGTH – how many existing elements you want to remove from starting from OFFSET.
* For insertion set Length to 0 .this tells the splice not to remove elements, only to insert.
* LIST – the new elements to insert at offset.
* The List doesn’t have to be the same size as Length -Perl automatically expand or shrink the array as needed.
* **Syntax to remove elements** - splice @array\_name , offset , length;
* Remove specific elements from array by specifying index and number of elements to delete.
* **Syntax to remove elements from index to end** – splice @array\_name , offset ;
* Remove element from index till end.
* **Syntax to replace element** – splice @array\_name , offset , length ,list;
* In replacing if we give offset as 3 and length as 2, then 2 element will be removed from index 3 and new values will be added.

**Hashes** (% sigil ) –

1. Dictionaries – unordered collection of key-value pairs.
2. %ages=(“Alice” => 30,”Bob” => 25); #declaration
3. $ages{“Alice”} return 30.
4. $ is used for fetching single value.
5. $ages{“Jon”}=40; # Add / Update
6. $ages{$key}=$value; #user input
7. my $hash\_count = %ages; # will give output as size of hash
8. my $hash\_count = scalar(%ages); # will give output as **size of hash**
9. Unordered key-value pair
10. **Delete key** -value : delete $ages{“bob”};
11. **Checking for key existence**:
12. If(exists $ages{“Alice”}){ print “Exists” ;}
13. Update : $ages{“Alice”}=40;

**Sort function** :

**Arrays** :

1. **Alphabetical** :

* **Ascending order.**
* Syntax- my @new\_array = sort @array\_name ;
* It sorts in lexical ordering.
* It does not change the original array.
* It creates a new array and store the sorted values.
* For **descending order** :
* Syntax – my @new\_array=sort { $b cmp $a } @array\_name;

1. **Numbers** :

* We have separate sorting for numbers and strings because perl sorts in lexical ordering.
* Inside code block , Perl provides two special variables for comparison:
* $a : The first element being compared.
* $b : The second element being compared.
* For ascending use : $a <=> $b;
* For descending use : $b <=> $a;
* Syntax – my @asc\_Arr= sort {$a <=> $b} @number\_array;
* Syntax – my @desc\_Arr= sort {$b <=> $a} @number\_array;
* <=> it is “spaceship” operator in perl. It returns ,
* -1 if $a is less than $b
* 0 if $a is equal to $b
* 1 if $a is greater than $b

Hash :

1. We can’t sort hash because it’s unordered. But we can sort keys and values of hash.

2. We can sort and store it in new array by using that array we can fetch the values.

3. foreach my $keys (sort keys %hash\_name){}; # normal sorting using keys fetch directly.

1. **Sort by keys (alphabetical) :**

* Syntax - my @new\_keys = sort keys %hash\_name;
* sorts keys in lexical way. (**Ascending order**)
* For **descending** order :
* Syntax – my @new\_keys = sort { $b cmp $a } keys %hash\_name;

1. **Sort by values (numbers) :**

* To sort values, you still work with the keys of the hash , but your comparison block will access the hash values associated with those keys.
* **After sorting using values the new array will contain the keys sorted based on values.**
* Syntax - my @new\_values = sort {$hash\_name{$b} <=> $hash\_name{$a}} keys %hash\_name;
* sorted the keys in **descending** based on values.
* **For ascending :**
* Syntax - my @new\_values = sort {$hash\_name{$a} <=> $hash\_name{$b}} keys %hash\_name;

**Note for Sort hash :**

* When the values are equal, Perl’s sort is stable, meaning it falls back on a secondary comparison (i.e. The alphabetical order of the keys themselves).
* If you want to handle it , you should handle this explicitly :
* Sort by descending ,but the values are same, sort by name ascending.
* Syntax – my @new\_Array=sort { $score{$b} <=> $score{$a} || $a cmp $b} keys %hash\_name;

**Date and Time :**

There are two fundamental methods for getting time and date .Below functions are Perl’s built-in functions.

1. time() –

* This function is a simple workhorse. It returns a single, large integer :

the number of seconds that have passed since a specific moment in time called the Epoch.

* The Epoch is January 1, 1970 , 00:00:00 Universal Time.
* It’s not very useful for human to read directly, but it’s perfect for comparing timestamps or for calculations.
* Syntax – my $times=time();
* 1754142800.

1. localtime() –

* This function is much more human-friendly. It takes that big number from time() and breaks it down into a list of numbers you can understand.
* localtime() behaves differently depending on whether you use it in a scalar or a list context.
* **In Scalar Context :**

It returns a single human-readable string. For example, if you run it , output will be like ---

Syntax = my $times=localtime();

“Sat Aug 2 20:21:21 2025”

* + - * + **In List Context :**

It returns a list of 9 numbers .

($sec , $min , $hour , $mday , $mon , $year , $wday , $yday , $isdst )

$sec – Seconds (0 – 59 )

$min – Minutes ( 0 -59 )

$hour – Hours (0 – 23 )

$mday – Day of the month (1 – 31 )

$mon – Month ( 0-11 ). Jan -0 , Feb -1 ….

$year – Year since 1900. To get the current year , you need to add 1900 to this number.

$wday – Day of the week (0 – 6 , With 0 begin Sunday )

$yday – Day of the year (0-365 )

$isdst – Is Daylight Savings Time is effect ? (0 or 1)

Syntax – my ($sec , $min , $hour , $mday , $mon , $year) =localtime();

**Time::Piece :**

* + - Time::Piece is a built-in module that comes with most modern perl installation(version 5.9.5 and above )
    - Time::Piece gives an object oriented way to handle dates and times.
    - It already knows the correct year, month and day , and it provides easy-to-use methods to access them.
    - **Syntax** – **use Time::Piece;**
    - It should be on the top of the while, it overloads the Perl’s built-in localtime() function so that it returns a Time::Piece object instead of a list of numbers.

use Time::Piece;

my $t=localtime;

print “1 . The object’s raw value : $t\n”;

print “2. Current year : “,$t->year, “\n”;

print “3.Current month (corrected): “,$t->mon,”\n”;

print “4.Current day : “, $t->mday, “\n”;

print "5.Formatted date (YYYY-MM-DD) : “,$t->ymd, “\n”;

$t->mon Returns value of month from 1 to 12.

$t->year Year will give current year like 2025.

**Common Methods in Time::Piece :**

* + - $t->datetime --- Gives full date time.
    - $t->year --- Gives year
    - $t->mon --- Gives month (1-12)
    - $t->mday--- Gives day of month (1-31)
    - $t->wday --- Gives day of the week in numbers (0=sun)
    - $t->day ---- Gives day of the week in String (Sun , Sat..)
    - $t->yday ---Gives day of the year
    - $t->hour ---Gives hour
    - $t->min --- Gives minutes
    - $t->sec --- Gives seconds

**String Format Time : strftime()**

* This method takes a format string as its argument , which is a blueprint for how the date and time should be displayed.
* This format string is made up of regular text and special format codes that begins with a percentage sign (%).
* Use in top : **use Time::Piece;**
* Common format codes :
* %Y – Full year with century (e.g., 2025)
* %m – Month as 2 digit number ( e.g.,08)
* %d – Day of the month as 2 digit number (e.g., 02)
* %H – Hour in 24-hour format ( e.g., 21 for 9 PM)
* %M – Minute as a 2 digit number (e.g.,51)
* %S – Seconds as 2 digit number (e.g., 25)
* %A – Full Weekday name (e.g., Saturday )
* %B – Full month name (e.g., August )
* %a – Abbreviated weekday name (e.g., Sat)
* %b – Abbreviated month name (e.g., Aug)

Example :

use Time::Piece;

my $t= localtime ;

my $file\_name\_format=$t->strftime(‘%Y-%m-%d\_%H%M%S’);

#2025-08-02\_215147

my $report\_date\_format=$t->strftime(‘%A, %B %d, %Y’);

#Saturday, August 02, 2025

**Manipulating Dates and Times :**

* Trying to add a month to a date with the basic list of numbers would be very complicated (e.g., Is it a leap year ? does the month have 31 or 30 days ?) Time::Piece handles this for u.
* The key idea is that you can use standard arithmetic operators (+ and - ) to add or subtract time interval from a Time::Piece object.
* **Syntax** : use Time::Piece;
* **Syntax** : use Time::Seconds ; # This is where the intervals like “days” and “months” come from.
* Methods in this :
* days( #interval) – Creates interval of N days
* months( #interval ) – Creates interval of N months
* years(#interval) – Creates interval of N years
* seconds( #interval ) – Creates interval of N seconds
* minutes( #interval ) – Creates interval of N minutes
* hours( #interval ) – Creates interval of N hours
* weeks( #interval ) – Creates interval of N weeks

**Example** :

use strict;

use warnings;

use Time::Piece;

use Time::Seconds; # This is where the intervals like "days" and "months" come from

my $t = localtime; # Get the current time: Sunday, August 3, 2025

print "Current Date: ", $t->strftime('%a %b %d, %Y'), "\n";

# **Scenario 1: Add 7 days**

# The 'days' function converts a number into a Time::Seconds object

my $a\_week\_later = $t + days(7);

print "A week later: ", $a\_week\_later->strftime('%a %b %d, %Y'), "\n";

# **Scenario 2: Subtract 3 days**

my $three\_days\_ago = $t - days(3);

print "Three days ago: ", $three\_days\_ago->strftime('%a %b %d, %Y'), "\n";

# **Scenario 3: Add 1 month (handles month-end and leap years automatically!)**

my $a\_month\_later = $t + months(1);

print "A month later: ", $a\_month\_later->strftime('%a %b %d, %Y'), "\n";

# **Scenario 4: Add 1 year**

my $a\_year\_later = $t + years(1);

print "A year later: ", $a\_year\_later->strftime('%a %b %d, %Y'), "\n";

**Comparing Date Objects :**

* **Direct Comparison :**

Use operators like ==, != , > , < ,>= ,<= to compare two Time::Piece objects.

Perl Automatically decides which date comes before or after the other.

* **Finding the Difference :**

When u subtract 2 Time::Piece objects , Perl returns a Time::Seconds object representing the difference between them.

This is used to get difference in a human readable form like days and months.

**Example :**

use strict;

use warnings;

use Time::Piece;

my $today = localtime; # Our current date: Sun Aug 03 11:08:02 2025

my $anniversary = Time::Piece->strptime('2025-10-15', '%Y-%m-%d'); # Wed Oct 15, 2025

print "Today: ", $today->ymd, "\n";

print "Anniversary: ", $anniversary->ymd, "\n";

# -----------------

# Direct Comparison

# -----------------

if ($anniversary > $today) {

    print "\n1. The anniversary is in the future.\n";

} else {

    print "\n1. The anniversary has already passed.\n";

}

# -----------------

# Finding the Difference

# -----------------

# Subtracting returns a Time::Seconds object

my $time\_difference = $anniversary - $today; # gives result in seconds

# We can then get the difference in specific units

my $days\_to\_go = $time\_difference->days; #convert the seconds to days.

my $months\_to\_go =int($days\_to\_go/30);

print "\n2. Time difference in seconds: $time\_difference\n";

print "   Days until anniversary: $days\_to\_go\n";

print "   Months until anniversary: $months\_to\_go\n";

| **Method** | **Returns** | **Description** |
| --- | --- | --- |
| $diff->seconds | Total seconds (as a number) | Returns the full difference in **seconds**. |
| $diff->minutes | Total minutes (floating-point) | Returns the full difference in **minutes**. |
| $diff->hours | Total hours (floating-point) | Returns the full difference in **hours**. |
| $diff->days | Total days (floating-point) | Returns the full difference in **days**. |
| $diff->weeks | Total weeks (floating-point) | Returns the full difference in **weeks**. |

If you want to calculate:

* **exact months** or
* **human-friendly duration** like 2 months, 3 days

You'll need to use **other modules** like:

* Date::Calc
* DateTime
* Time::Duration

**DateTime :**

Before we get to the methods, it's important to understand a few key principles of how DateTime works: (We need to install DateTime)

 1. **Time Zones are Explicit**: Every DateTime object must have a time zone. This is its most fundamental concept.

 2. **Immutability**: DateTime objects are immutable. This means that methods like ->add() or ->set\_time\_zone() don't change the original object. Instead, they return a new DateTime object with the updated time. This is a best practice in object-oriented programming to prevent unexpected side effects.

 3. **Precise Arithmetic**: Date arithmetic is handled by separate DateTime::Duration objects, which makes calculations very explicit and reliable.

**Key Methods and Concepts**

1. **Creating DateTime Objects**

You create objects using ->now(), ->new(), or ->from\_epoch():

 1. DateTime->now(): Creates an object for the current date and time. It will default to your system's time zone.

 2. DateTime->new(): This is the primary way to create a specific date. You use named parameters for clarity.

 3. DateTime->from\_epoch(): Creates a date object from a raw timestamp (seconds since the Epoch).

Example:

use DateTime;

my $now = DateTime->now;

my $dt\_ist = DateTime->new(

    year      => 2025,

    month     => 8,

    day       => 3,

    hour      => 14,

    minute    => 0,

    time\_zone => 'Asia/Kolkata',

);

**Accessing Information (Accessor Methods) :**

These methods allow you to retrieve parts of the date object. They are very similar to Time::Piece but with some key additions

Methods :

* year(), month() , day() ---get year month or day ---- $dt->year()
* hour() , minute() , second() --- get the hour , minute or second.---- $dt->minute()
* day\_of\_week() --- get day of the week (1-7) ---$dt->day\_of\_week()
* ymd() , hms() --- get formatted parts as strings ---$dt->ymd()
* is\_dst() --- Return true/false if Daylight Saving is active;
* time\_zone() --- get the DateTime::TimeZone object.--- $dt->time\_zone()->name()

**Manipulation and Arithmetic Methods :**

These methods return a new DateTime object with the updated value.

Method –

* plus(),add() --- add a duration. Takes named parameters. ---- $dt->plus( days => 5 )
* minus(),subtract() ----Subtract a duration . Takes named parameters. ------- $dt->minus( months => 2)
* set\_time\_zone() -------Return a new object with a different time zone.---------$dt->set\_time\_zone(‘America/New\_York’)
* set\_ymd(), set\_hms() -------- Return a new object with updated date/time parts.---$dt->set\_ymd(2026,1 , 1)

**Comparison and Difference Methods :**

You can compare DateTime objects directly with operators like > , < and ==, but for explicit checks and finding the difference, specific methods are often preferred.

Methods :

* is\_lt() , is\_gt() , is\_equal() – Explicitly compare 2 DateTime objects.------if($today->is\_lt($deadline)){}
* subtract\_datetime() --- Returns a DateTime::Duration object(like Time::Seconds) --- my $duration =$future\_dt->subtract\_datetime($now);
* print $duration->years(); print $duration->months()….
* in\_units() – A method on DateTime::Duration to get a value.------ $duration->in\_units(‘days’)

**Formatting :**

**DateTime** has a powerful->strftime() method similar to Time::Piece, but it also has a more advanced ->format\_cldr() method for internationalization.

Example :

my $formatted\_date =$dt->strftime(‘%Y-%m-%d’);

**Subroutine :**

* In Perl , a subroutine is a named block of code that performs a specific task.
* They are often called as functions or methods .

**Defining a Subroutine :**

* You define a subroutine using a **sub** keyword , followed by the name you want to give it, and then a block of code enclosed in curly braces {}.
* Example :
* sub subroutine\_name { #body }

**Calling a subroutine :**

* subroutine\_name();
* subroutine\_name;

**Passing Arguments :**

* You pass arguments to a subroutine by listing them in a comma-separated list after the subroutine name.
* Subroutine\_name(arg1, arg2, arg3);
* Subroutine\_name arg1, arg2, arg3;

**Accessing Arguments : The @\_ Array**

* When u calla subroutine with arguments, Perl automatically puts all of those arguments into a special, built-in array called @\_ .This array is local to subroutine.
* U can access the individual arguments from @\_ using their index, starting from 0.
* Syntax : $\_[0]
* Syntax : $\_[1]
* Example:
* sub greet\_person{

my $name=$\_[0];

my $age=$\_[1];

# my ($name , $age)= @\_; standard way of Perl

}

* my ($name , $age)= @\_; Using this is the best practice.
* If you pass a list of items to a subroutine, they are always flattened into a single, combined list in the @\_ array.
* Even if u pass array or hash subroutine takes it a single flattened list.
* **For array of values passed to subroutine** :Perl flattens all the arguments into a single, flat list inside the @\_array.
* This means if u pass scalar and an array to subroutine ,you won’t get a scalar at $\_[0] and a n array in $\_[1]. Instead ,you’ll get the scalar followed by a single element of the array, all as one long list.
* **For Hash of values to subroutine** : The exact same thing happens with hashes. The key-value pairs of the hash are flattened into an alternating list of keys and values.

**Return Keyword :**

* return keyword
* return keyword does 2 things ,
* 1. Immediately Stops the execution of the subroutine.
* 2. It send the value back to the line where it was called.
* Example : return $variable;
* The key thing to remember is that the return keyword, when given an array or a hash, return a flat list of all the elements. The calling code then captures that list into the appropriate type of variable.
* **Returning an array :**
* Return list of fruit.
* The calling code then captures that list into a array variable. @
* **Returning Hash :**
* The same principle applies to hashes. The return keyword returns a flat list of alternating key-value pairs.
* The calling code then captures that list into a hash variable , and Perl correctly rebuild hash. %
* **Notes on Context**
* **What it is**: Context is the situation in which a subroutine is called. The two most common contexts are scalar (expecting one value) and list (expecting many values).
* **The Rule**: A subroutine's return value can change based on the context.
* **Example**:
* my @list = my\_sub(); (List context) will capture all returned items.
* my $count = my\_sub(); (Scalar context) will capture the count of items returned.
* **Key takeaway**: Perl is smart, and it's important to know which context you are using when calling a subroutine.
* **Notes on Prototypes**
* **What it is**: A prototype is a special syntax (($$), (@)) added to a subroutine definition to enforce its argument signature at compile time. To inform how many arguments should be passed
* **The Syntax**:
* sub my\_sub ($$): Expects two scalars.
* sub my\_sub (@): Expects a single array (and doesn't flatten it).
* **Why it's Controversial**: Prototypes don't always behave in an intuitive way and can lead to confusing errors.
* **Best Practice**: The modern, recommended approach is to avoid prototypes and instead perform a manual check on the @\_ array inside your subroutine. This is more explicit and much safer.

**References :**

* It is nothing but like pointer in c/c++.
* It is a scalar variable which holds the address of another variable like scalar variable, hash variable , array variable.
* A reference (e.g., my $age\_ref = \$age ).
* Use backslash (\) operator for creating a reference .
* A reference is always scalar. We will store it scalar variable only.
* Useful for 2 main reasons :
* 1. **Building Comple Data Structures**: You can’t directly put an array inside another array, but you can put a reference to an array inside another array. This allows you to build 2D arrays or hashes of hashes.
* **2. Passing Arguments without Flattening** : When u pass an array to a subroutine it’s flattened. By passing a reference to an array instead, you only pass a single scalar variable(the address ), and the subroutine can then use that address to access the original ,un-flattened array.

**Creating a Reference for scalar :**

* my $scalar=”Hello”;
* my $scalar\_ref=\$scalar;

**Creating a Reference for Arrays :**

* my @array= qw(red green blue );
* my $array\_ref=\@array;

**Creating a Reference for Hash :**

* my %hash=(name => “Sanjai” , age=> 21);
* my $hash\_ref=\%hash;

**Creating Anonymous References** :

* We can createa reference to an unnamed(anonymous ) data structure on the fly.
* An anonymous array reference is created with […]
* An anonymous hash reference is created with {…}
* my $anonymous\_array\_ref=[“apple”, “banana”];
* my $anonymous\_hash\_ref={type=> “fruit” , color => “red”}

**Dereferencing :**

* Getting the value back from address.
* Syntax : Combine the appropriate sail ($,@,%) with the reference variable in curly braces {},

**Dereferencing a Scalar Reference** :

* To get scalar value back, use the scalar sigil ($)
* Syntax - ${$name\_ref}

**Dereferencing an Array Reference** :

* To get the entire array back, use the array sigil (@)
* Syntax - @{ $tasks\_ref}
* Important : A much more common and readable way to access a single element from a dereferenced array is to use -> operator .It automatically dereferences the array for u.
* Syntax - $task\_ref->[0]

**Dereferencing a Hash Reference** :

* To get the entire hash back , you use the hash sigil ( %).
* Syntax - %{@hash\_ref}{‘name’}
* Important much more simple way to access a single element from a dereferenced hash is to use -> operator.
* Syntax - $hash\_ref -> {name}

**Complex Data Structure :**

* By using reference we ca build a complex data structures, like array that contains other arrays, or hash that contain other arrays.
* We can do this with the help of anonymous reference, using[..] for array and {..} for hash.

**Creating an Array of Arrays ( A 2D Array):**

* Syntax – my @matrix=( [1,2,3],[4,5,6],[7,8,9]);
* The @matrix array contains 3 scalar references, one for each row.

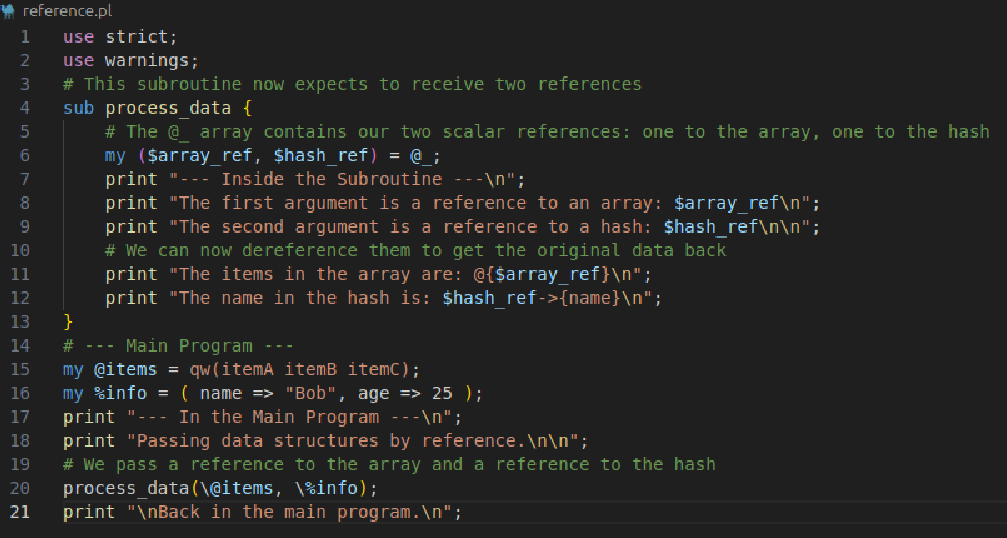
**Creating a Hash of Hashes:**

* Syntax – my %personal1 = ( “Alice “ =>{position => “Manager”, dept =>”Sales”},”Bob” => { position=> “Analyst” , dept =>”Finance” } );
* Alice info is stored at $personal1{Alice}

**Passing Reference to Subroutine :**

* If we pass an array ,the subroutine receives a single flattened list in the @\_ array.
* Solution : You pass a reference to the data structure instead. Since reference is a single scalar, it’s passed as a single item in the @\_ array, and the original structure is preserved.
* Example :

use strict;  
use warnings;  
# This subroutine now expects to receive two references  
sub process\_data {  
    # The @\_ array contains our two scalar references: one to the array, one to the hash  
    my ($array\_ref, $hash\_ref) = @\_;  
    print "--- Inside the Subroutine ---\n";  
    print "The first argument is a reference to an array: $array\_ref\n";  
    print "The second argument is a reference to a hash: $hash\_ref\n\n";  
    # We can now dereference them to get the original data back  
    print "The items in the array are: @{$array\_ref}\n";  
    print "The name in the hash is: $hash\_ref->{name}\n";  
}  
# --- Main Program ---  
my @items = qw(itemA itemB itemC);  
my %info = ( name => "Bob", age => 25 );  
print "--- In the Main Program ---\n";  
print "Passing data structures by reference.\n\n";  
# We pass a reference to the array and a reference to the hash  
process\_data(\@items, \%info);  
print "\nBack in the main program.\n";



**List :**

* A List is a actual scalar values. Where as Array is a variable that stores the list.
* A List is a temporary, literal, ordered collection of values. It doesn’t have name or sigil($ , %, @ ).It exists only part of a large expression.
* An array is a variable that stores a list. It has name and a sigil (@)

**Creating and using Lists :**

* The most common way to create a list by enclosing a comma-seperated sequence of values in parentheses.
* Syntax – (value1, value2 , value3, …)
* **List Assignment** :
* my ($day , $month , $year) = (27,08,2025);
* assign **list to array**.
* my @arr=(“apple” , “banana” , “cherry” );

**List in Different Contexts:**

* A list’s behaviour depends on the context it is evaluated in.
* List Context : when perl expects a list, it returns the entire list.
* Example : my @list=(1,2,3);
* Scalar Context : when perl expects a single value, a list will return the number of elements it contains.
* Example : my $count=(1,2,3);

**List and Subroutines** :

* The entire concepts of passing arguments to a subroutine is built on lists.

**Passing Arguments** :

* When we call a subroutine with arguments, the arguments are first collected into a single, flat list.
* Example :
* my @arr=qw(a b c)
* my\_sub(1, @arr , 2); # @\_ will be (1, ‘a’ , ‘b’ , ‘c’ ,2)

**Returning values**:

* When subroutines return a list, the calling code a choose to capture the list in either an array or scalar variable, which the dictates the context.
* Example:
* sub get\_arr{ return (“A” , “B”); }  
  my @item=get\_arr(); # List context, @item is (‘A’, ‘B’)  
  my $count=get\_arr(); # Scalar context , $count is 2  
  my $count=scalar(()=get\_arr()); # if did not give the output try this

**Quote Word:**

* qw() operator- It is a common short-hand operator for creating list of strings without quotes and commas.
* qw(apple orange) is equal to (“apple” , “orange”).

**List of Lists :**

* We cannot have list of list directly.
* ((1,2),(3,4),(5,6)) is not list of list ,it is just single list of (1,2,3,4,5,6).
* This is the reason why we use references to create nested data structures: ([1,2],[3,4]) , it is list of 2 array reference.

**Split and reverse :**

* Split based on symbols**.**
* Example:   
  my $dob=”15-07-2004”;  
  my ($day, $month ,$year)= split /-/ $dob;  
  # $day is 15  
  # $month is 07  
  # $year is 2004

**Reverse :**

* It reverse the list.
* It produces a new list of the elements in a reverse order.
* Example :  
  my @arr=(1,2);  
  my @new =reverse @arr;  
  print @new; # 2 1
* **Array Slicing** [Accessing / Assigning to multiple elements at once ] :
* Syntax - @array\_name[index1,index2]
* my @new\_arr=@array\_name[index1,index2] #fetching from one array and storing it in another array
* @arr[1,3]=(25,35); # assigning this value to index 1 and 3.
* print @arr[1,3]; # 25,35 will be the output
* @arr[1..8]=(1,2,3,4,5,6,7,8); #assigning for a range.
* If there a already some value and if u give index more than the size then perl will increase the size till the index2.
* .. (double dot ) means range operator in perl.

File Handling :

* File handling in Perl allows you to **read from**, **write to**, and **manipulate files** using built-in functions like open, close, print, readline, and others.
* Syntax - open(FILEHANDLE, MODE, FILENAME);
* Recommanded Syntax : open(my $fh, '<', 'file.txt') or die "Can't open file: $!";
* Access Modes :

| **Mode** | **Meaning** |
| --- | --- |
| * < | * Read only (default) |
| * > | * Write only (truncates existing content) |
| * >> | * Append (write at the end) |
| * +< | * Read and write (no truncate) |
| * +> | * Read and write (truncates) |
| * +>> | * Read and write, append mode |

* > , >> ,+> , +< , +>> , will create a new file if it does not exists.

**Reading line by line :**

* Example:  
  while (my $line = <$fh>) {

print $line;

}

**Reading All at Once :**

* my @lines = <$fh>;

**Read Entire file into Scalar (Slurp Mode)** :

* my $content = do {

local $/; # Undefines input record separator

<$fh>;

};

**Writing to a file**:

* print $fh "This is a line\n";

**Redirect STDOUT to a File :**

* Example:

open(STDOUT, '>', 'output.txt') or die $!;

print "This will go to output.txt**";**

close(STDOUT)

**Closing file** :

* close($fh) or warn "Could not close: $!";

**File Tests (Using -X Operators) :**

* Perl has file test operators to check various file attributes.

| **Operator** | **Checks If...** |
| --- | --- |
| * -e | * File exists |
| * -r | * Readable |
| * -w | * Writable |
| * -x | * Executable |
| * -s | * Non-zero size (returns size) |
| * -z | * Zero size (return true if file is empty) |
| * -f | * Regular file |
| * -d | * Directory (return true if it a directory) |
| * -T | * Text file |
| * -B | * Binary file |

-f operator returns **true (1)** **only if**:

* The file **exists**, and
* It is a **regular (plain) file**, i.e., **not**:
  + a directory
  + a symbolic link
  + a pipe
  + a socket
  + a character/block device (e.g., /dev/null)
* Example :  
  if (-e 'log.txt') {

print "File exists\n";

}

**Error Handling :**

* **What Is Error Handling?**

Error handling in Perl is how you manage:

* + - 1. **Failures** (e.g., failed open calls)
      2. **Exceptions** (unexpected errors during execution)
      3. **Warnings** (non-fatal alerts)
* Perl provides simple tools (die, warn), system variables ($@, $!, $?), and advanced modules like autodie and Try::Tiny.

**Using die – Fatal Error**

* die immediately terminates your script with an error message.
* Perl prints the message and exits.
* die is like exit() but shows what went wrong.
* **Syntax** : open(my $fh, '<', 'missing.txt') or die "Could not open file: $!";
* **$!** contains the OS**/**systemerror (like No such file or directory).
* Prints message to STDERR and exits with non-zero status

| **Feature** | **Fatal Error** | **Non-Fatal Error** |
| --- | --- | --- |
| **Effect** | Immediately terminates the script | Script continues execution |
| **Caused by** | die, uncaught exceptions, failed system ops | warn, user-defined warnings |
| **Typical Use** | Serious problems that must stop execution | Informational or recoverable issues |
| **Output** | Prints message and exits with non-zero status | Prints message but script continues |
| **Can be handled?** | Yes (with eval, Try::Tiny) | Yes (can suppress or redirect warnings) |
| **Default behaviour** | Script halts unless caught | Warning shown; execution continues |
| **Error variable** | $@ (after eval) | $! (OS error), $^W (warnings) |

* Fatal Error is like Checked Exception we must handle it.
* Non – Fatal Error is like unchecked Exception we can handle if needed. Execution continues; handle it only if needed

**Use warn – Non-Fatal Warning :**

* Warn prints the error message but continues execution
* **Syntax** - warn "Missing file, continuing anyway\n" unless -e 'file.txt';
* use this when wrong, but not critical.

**How to write all the warnings and errors ,prints into a file ?**

* Example:  
  # Step 1: Open a log file

open my $logfh, '>>', 'output.log' or die "Can't open log file: $!";

# Step 2: Redirect STDOUT and STDERR

open STDOUT, '>&', $logfh or die "Can't redirect STDOUT: $!";

open STDERR, '>&', $logfh or die "Can't redirect STDERR: $!";

# Step 3: Use print, warn, or die as usual

print "This is a normal print statement\n";

warn "This is a warning message\n";

die "This is a fatal error\n";

* **>&** -It duplicates or redirects the output of one filehandle to another existing filehandle or file descriptor.
* There are many ways to write to a file.

**System Error Variable $!**

* $! = last system call error
* Context-sensitive:
* In string context → error message
* In numeric context → errno (error number)
* Example :  
  print "Error: $!\n"; # "No such file or directory"

print "Error number: ", 0 + $!, "\n"; # 2

**Capturing Exceptions with eval {} :**

* eval lets you catch errors from die without crashing the program.
* Example :  
  eval {

open(my $fh, '<', 'missing.txt') or die "Open failed: $!";

};

if ($@) {

print "Caught error: $@\n";

}

* $@ holds the error message from the last eval.
* You must check $@ immediately after eval.

**The $@ Variable**

* Holds the **error string** from eval {}.
* Overwritten by any new eval or even some runtime warnings.

**Using autodie :**

* Automatically makes built-in functions throw exceptions if they fail.
* Use this – use autodie;
* Example :
* use autodie;
* open(my $fh, '<', 'data.txt'); # No need for "or die"
* Supports: open, close, system, unlink, chdir, etc.
* Does not affect user-defined subs unless extended.

**Using Try::Tiny – Safer Exception Handling** :

* eval is prone to edge-case bugs. Use Try::Tiny instead.
* Use the header file – use Try::Tiny;
* Example **:**use Try::Tiny;

try {

die "Critical failure!";

} catch {

warn "Caught exception: $\_";

};

* $\_ holds the exception.
* Clean, localized, no accidental overwriting.

**Using Carp for Better Messages :**

* Standard warn/die reports from inside a subroutine may point to the sub, not the caller.
* Use Carp to report **to the caller**.
* Syntax – use Carp;
* **Example :**use Carp;

sub dangerous {

confess "Fatal error from inside a sub!";

}

dangerous(); # Trace to where sub was called

| **Function** | **Use case** |
| --- | --- |
| * carp | Like warn, to caller |
| * croak | Like die, to caller |
| * cluck | warn + stack trace |
| * confess | die + stack trace |

* Syntax when u use cluck and confess - use Carp qw(cluck confess);

**System Calls and $?**

* $? contains the exit status of the last system/exec call.
* **Example** :
* system("ls /nonexistent");
* print "System failed with code: $?\n" if $? != 0;
* To extract the exit code:
* my $code = $? >> 8;

**Data::Dumper** :

* Data::Dumper is a Perl core module used to convert Perl data structures into readable strings.
* This is especially useful for debugging complex structures like arrays, hashes, objects, etc.
* To **print the internal structure** of a variable (arrays, hashes, references, objects).
* To **debug deeply nested data**.
* To **serialize** data for storage or transmission.
* To see **real-time values** in scripts or subroutines.
* Syntax - use Data::Dumper;

**Procedural :**

**Dumping Scalar** :

* Example :

use Data::Dumper;

my $x = 42;

print Dumper($x);

* Output : $VAR1=42;

**Dumping Arrays** :

* Example:  
  my @arr = (1, 2, 3);

print Dumper(\@arr);

* Output : $VAR1=[  
   1,  
   2,  
   3  
   ];
* You must **pass references** to complex structures (\@arr, \%hash, etc.)
* Every Dumper() is a separate call so the variable name will be like $VAR1 for all.
* If we give multiple variables a single Dumper() then it will be $VAR1, $VAR2,etc…

**Dumping Hashes** :

* Example :  
  my %h = (name => 'Alice', age => 30);

print Dumper(\%h);

* Output :   
  $VAR1 = {

'name' => 'Alice',

'age' => 30

};

**Multiple Variables :**

* Code :  
  my $a = 1;

my @b = (2, 3);

print Dumper($a, \@b);

* Output :  
  $VAR1 = 1;

$VAR2 = [

2,

3

];

* If we give multiple variables a single Dumper() then it will be $VAR1, $VAR2,etc…
* You can change **global settings** using package variables:
* $Data::Dumper:Indent=1; It changes the global settings.

**Object-Oriented (OO):**

* The Object-Oriented (OO) interface in Data::Dumper provides finer control, method chaining, and safer use in larger programs or subroutines than the simpler procedural style.
* Example code :  
  use Data::Dumper;

my $data = { name => "Alice", age => 30 };

my $dumper = Data::Dumper->new([$data]);

print $dumper->Dump;

**Method Chaining Example :**

* Example :use Data::Dumper;

my $obj = { x => 10, y => [1,2,3] };

print Data::Dumper

->new([$obj], ['object'])

->Indent(2)

->Terse(0)

->Sortkeys(1)

->Useqq(1)

->Dump;

* Each method returns the object itself ($self), so you can chain them.
* It affects only the object $obj not the global setting.

**Notes** :

* Procedural and OO Mixing: Don't mix OO and procedural interface unless you're very sure. OO avoids global side effects.
* Performance: OO interface is slightly slower but preferred for clarity and safety in larger projects.
* Name Clashes: Setting Names() can lead to mismatched names if the number of names doesn't match the number of references.
* Blessed Refs: Dumper will print bless(...) for objects. You can change how it displays with Bless().
* Purity Limitations: Not everything can be serialized 1:1 using Purity(1) — especially code refs or tied variables.
* Thread Safety: OO usage is safe across threads or subroutines as it doesn’t rely on shared global state.
* You can change **global settings** using package variables:
* $Data::Dumper:Indent=1; It changes the global settings.

**Customizing Output :**

* Changing Variable Names :
* Changing the variable names ($VAR1 to \*b)
* Syntax – Data::Dumper->new([\$data] , [ $name ] );
* Prefix \* means you want to name it as an array variable @b
* Prefix $ means would name it as Scalar , % for hashes .
* Example :  
  my @b=(2,3);

my $d = Data::Dumper->new([\@b], ['\*b']);

print $d->Dump;

* Use the second array to provide custom variable name.
* Output :  
  @b = [

2,

3

];

**Debugging Methods :**

* This works only with Object oriented (OO) interface only.

| **Method** | **Purpose** |
| --- | --- |
| Indent($n) | Control indentation level (0, 1, 2) |
| Terse(1) | Remove $VAR1 = from output |
| Names(\@names) | Set custom variable names |
| Sortkeys(1) | Sort hash keys |
| Useqq(1) | Show escaped characters inside strings (e.g., \n, \t) |
| Purity(1) | Ensure dumped output can be eval'd to recreate data |
| Maxdepth($n) | Limit nested levels in deep structures |
| Quotekeys(0) | Do not quote hash keys |
| Bless($pkg) | Change the package name shown for blessed refs |
| Deparse(1) | Deparse code references (if possible) |

| Feature | OO Interface (->new) | Procedural (Dumper) | Global Defaults |

|---------------------|----------------------|----------------------|--------------------------|

| Indent() | Yes | No | Yes (via $Data::Dumper::Indent) |

| Terse() | Yes | No | Yes (via $Data::Dumper::Terse) |

| Sortkeys() | Yes | No | Yes (via $Data::Dumper::Sortkeys) |

| Names() | Yes | No | Yes (via $Data::Dumper::Names) |

| Method chaining | Yes | No | No |

| Safer in subs/libs | Yes | Yes (limited) | No |

**Regular Expression :**

* Match operator =~
* Not Match !=
* Regex should be inside **/ #regex /**

| **Symbol** | **Meaning** |
| --- | --- |
| . | Any single character except newline |
| ^ | Start of string |
| $ | End of string |
| \* | 0 or more of preceding element |
| + | 1 or more of preceding element |
| ? | 0 or 1 of preceding element |
| \d | Digit (0–9) |
| \w | Word character (a-zA-Z0-9\_) |
| \s | Whitespace |
| \b | Word boundary |

**Quantifiers** :

| Symbol | Meaning |
| --- | --- |
| {n} | Exactly n times |
| {n,} | n or more times |
| {n,m} | Between n and m times |

**Grouping :**

* my $name = "Mr. John Doe";

if ($name =~ /(Mr\.|Mrs\.)\s+(\w+)\s+(\w+)/) {

print "Title: $1, First: $2, Last: $3\n";

}

* use () group part of the pattern

**Substitution :**

* Syntax - s/#word/#replace\_word/ (s///)
* my $str = "perl is awesome";

$str =~ s/perl/Perl/;

print $str; # Perl is awesome

**Modifiers** :

| **Modifier** | **Meaning** |
| --- | --- |
| i | Case-insensitive match |
| g | Global match |
| m | Treat string as multi-line |
| s | Allows . to match newline |

* Example :  
  "HELLO" =~ /hello/i # Match

"aaaabbbb" =~ s/a/A/g # Replace all a's

**OOPS** :

* Perl’s OOPS is built on existing ,simple primitives,

**Classes are just packages :**

* In Perl , a package is a namespace, or a container, for a set of related code and data.
* When u want to define a class, you simply create a package with the same name.
* package Car;
* This line tells , Perl that all the subroutines (sub) and variables we’re about to define belong to Car package.
* This package Car; line is equivalent of public class Car{}.
* Extension - .pm
* This extension should be added to the file after it has the package details.

**Constructors :**

* Here we are using method to initialize the object’s data with any arguments provided.
* In Perl, the universal convention is to name this method **new.**
* **Steps :**
* Get the class name: myClass->new()
* Create a Data structure: hash {}.create reference to hold the instance data.
* Bless the Reference
* Return the blessed Reference.

**Constructor initialization Arguments :**

* U can pass arguments to the constructor to initialize.  
  it should be key-value pair.
* my ($class, %args)=@\_;

**Constructors and Inheritance ;**

* What happens when one class inherits from another ?
* The child class’s constructor should typically call the parent class’s constructor to handle the shared attributes.
* This is done by calling new on the special SUPER pseudo-class.
* **Inheritance** we should use – **use parent ‘class\_name’;**
* **Syntax** : my $self=$class->SUPER::new(%args);
* # %args passing parameters.

**Objects are just blessed references :**

* This is the most unique part of perl’s OOPS.
* Object is just a regular data structure – usually a hash reference ({}) – that has been “blessed”.
* The bless function is a magic that associates a reference with a package.
* The bless function takes 2 arguments: the reference ($self) and the name of the package ($class).
* **Syntax** - bless $self , $class;
* bless is used to create object of the package.
* Before bless : $self is just a hash reference . you can access its keys, but you can’t call any methods on it.
* After bless: $self is now a Object. With this we can call the methods of that class.
* **Attributes are just hash keys** :
* a blessed hash reference, we can simply store all of the object’s data (it’s attributes) as key-value pairs inside that hash.
* $self->{color} = “red”;
* Here we are using our blessed hash reference ($self) to store the attribute color with the value “red”;
* **Shift keyword**
* Syntax : shift
* This is built-in perl function. It’s job is to take an array, remove the first element from it , and return the element.
* The default Array :When shift is used a subroutine without a specific array, it automatically operates on the special @\_ array.
* @\_ is where all the arguments to a subroutine are placed.
* my $class = shift ; is equal to my $class=$\_[0];

**Say function:**

* It is used after 5.10;
* It is same as print , but only one difference that is in print u have to manually add “\n” at last.
* But in say “\n” is added built-in at the end of the line;

**Methods** :

* A method in perl is just a subroutine that is defined inside a package (our class). The key difference is that when u call a method, Perl automatically passes the object itself as the very first argument.
* The first argument is named as $self.
* Two types of methods :
* **Class Methods**: Called on the class name (e.g . Car->new()) . The first argument received is the class itself(“Car”) .
* **Instance Methods** : Called on an object (e.g. $my\_car->start\_engine()). The first object received is object reference ($my\_car).
* **ref($my\_car)** – give the name of the class to which this object belongs.

**Access Modifiers**:

* In perl there no keywords like (public , protected, private).
* By default, all methods and attributes in a perl object are public.
* How does perl handle data privacy ? Through a gentleman’s agreement - a strong, universal followed convention .
* The convention : The Leading **Underscore \_**:
* If you want to mark a method or an attribute as “private” (i.e., accessible only with in a same class ), prefix its name with a single underscore \_.
* $self->{make} # is public
* $self->{\_internal\_id} # is private.
* $self->get\_make() # public method
* $self ->\_get\_internal\_id() #private method
* This underscore does not stop prevent the access with in the same class. It is a strong signal for the programmer that it should used only with in this class.
* Access the private variable is possible ,It works ,but u are breaking the convention.
* This \_ convention is used for both private and protected.

**Encapsulation:**

* Bundling data(Attributes) and the methods that operate on that data into a single unit( the object) , and hiding the internal details from the outside world.
* **Perl achieves this even without private keyword** :
* **Bundling** : The bless function bundles the data (the hash reference ) with the methods (the package’s subroutines).
* **Hiding** : By providing public “accessor” methods (like get\_make()) to interact with the data, you hide the fact that the data is stored in a hash. By creating a public interface.
* **Why it is important**?
* If u wish to change the $self->{year} to $self->{creation\_date}->{year}.
* If the users are directly using $car->{year} all the codes will beak.
* If users were using $self->get\_year() method , you only need to update the get\_year() method inside your class.

**Inheritance** :

* **Inheritance** we should use – **use parent ‘class\_name’;**
* **Syntax** : my $self=$class->SUPER::new(%args);
* # %args passing parameters to the parent class constructor.

**Method Overriding:**

* If u don’t like parent class method implementation u can override it according to ur wish in child class this is called overriding.
* It is normal like other language overloading only.
* Just use the same subroutine name in the child class.
* Calling the Parent’s Method with SUPER:: -
* Sometime’s we don’t want to completely replace the parent’s method, but rather add to it .
* You can do this by calling the parent’s version of the method using the SUPER::subroutine\_name;
* Syntax - $self->SUPER::subroutine\_name;

**Polymorphism**

* Concept: "Many forms". It's the ability to have a single interface (like a function call) work for objects of different classes.
* Perl's relaxed typing makes polymorphism natural. This is often called "duck typing": if it walks like a duck and quacks like a duck, Perl treats it like a duck.  
  Perl doesn't care if an object is-a Vehicle. It only cares if the object can do honk.
* **Example**:  
  # A subroutine that works with ANY object that can honk  
  sub make\_some\_noise {  
      my $noisy\_object = shift;  
      print "Testing noise on a ", ref($noisy\_object), ": ";  
      $noisy\_object->honk(); # This is polymorphism in action!  
  }  
  # --- In main.pl ---  
  my $car   = Vehicle->new();  
  my $truck = Truck->new();  
  make\_some\_noise($car);  
  make\_some\_noise($truck);  
  **Output**:  
  Testing noise on a Vehicle: Beep beep!  
  Testing noise on a Truck: HONK HONK!
* The make\_some\_noise subroutine works on both objects without needing to know their specific types, because they both fulfill the "contract" of having a honk method.

**Method Overloading** (Perl's version)

* This is another area where Perl differs greatly from C++/Java.
* **Classical Overloading**: In Java, you can define the same method name multiple times with different argument lists (e.g., do\_task(int a) and do\_task(int a, String b)). The compiler picks the right one.
* **Perl's Approach**: Perl does not have this kind of overloading. A subroutine has exactly one name and one definition.
* Instead, Perl achieves a similar result with a single subroutine that inspects its arguments and behaves differently based on what it receives.
* **Example**: A find method that can find by ID (a number) or by make (a string).  
  use Scalar::Util 'looks\_like\_number';  
  sub find {  
      my ($class, $arg) = @\_;  
        
      if (looks\_like\_number($arg)) {  
          print "Searching for a vehicle with ID: $arg\n";  
          # ... logic to search by number  
      }  
      else {  
          print "Searching for a vehicle with make: $arg\n";  
          # ... logic to search by string  
      }  
  }  
  # --- In main.pl ---  
  Vehicle->find(12345);      # Calls the numeric part  
  Vehicle->find("Honda");  # Calls the string part

**Abstraction**

* Hiding the unnecessary details and showing only the necessarydetails.
* Think of a car's dashboard. You have a steering wheel, accelerator, and brake pedal.
* This is the abstraction—a simple interface to the car. You don't need to know about the fuel injection, spark plugs, or brake fluid pressure (the hidden implementation details).
* In our code:
* Calling $car->get\_vin() is using an abstraction. You ask for the VIN, you get it. You don't care how it was generated or stored.
* The make\_some\_noise($object) subroutine is an abstraction. It works with any object that can honk, abstracting away the specific type of the object.

**Destruction** (Destructors)

* Concept: A destructor is a special method that is called automatically when an object is no longer being used and is about to be garbage collected by Perl.
* **The Perl Way**: A method named DESTROY.
* Its primary purpose is to clean up any resources the object was holding.
* Common uses include:  
   \* Closing a file handle.  
   \* Disconnecting from a database.  
   \* Releasing a network socket.  
   \* Logging that an object's lifecycle has ended.
* **Example**: Add a destructor to Vehicle.pm  
  # In Vehicle.pm  
  sub DESTROY {  
      my $self = shift;  
      my $make = $self->{make} || 'unnamed';  
      my $year = $self->{year} || 'unknown';  
      # This message will print just before the object is removed from memory  
      print "--> Destructor called: The $year $make vehicle is being scrapped.\n";  
  }
* How to see it work in main.pl:  
  sub run\_test {  
      print "Entering the test subroutine.\n";  
      my $temp\_car = Vehicle->new({ make => 'Tesla', year => 2025 });  
      print "The temporary car exists now.\n";  
      # $temp\_car goes out of scope when the sub ends  
  }  
  print "Starting script...\n";  
  run\_test();  
  print "The test subroutine has finished.\n";
* **Output:**  
  Starting script...  
  Entering the test subroutine.  
  The temporary car exists now.  
  --> Destructor called: The 2025 Tesla vehicle is being scrapped.  
  The test subroutine has finished.
* Notice the destructor was called **automatically** when run\_test finished, because the $temp\_car variable ceased to exist.
* **Database connection:**
* **DBI** (Database Interface)**-**  consider this as a universal Tv Remote control. U have all the functions in it like “power on” ,”adjust volume” **.**  It doesn’t matter which brand of tv u have it works well.
* **DBI –** is the Perl module that gives you a standard set of functions for interacting with any database.
* **cpanm DBI –** to install DBI
* **DBD** (Database Driver) – it is a specific remote that works with particular tv brand. The driver translates the generic “change channel” command from DBI into a specific signal your TV understands.
* **DBD –** is the specific driver module you installed for the database you want to use (e.g.. DBD::mysql for MySQL , or DBD::Pg forPostgreSQL).
* **cpanm DBD::mysql**
* The process of executing a query in Perl with DBI is a bit different from just typing a command into a database client. There 3 steps:
* Step 1-prepare
* Step 2- Execute
* Step 3- fetch