UNIT-1

CGI AND HANDLING COOKIES

Ruby is a general-purpose language; it can't properly be called a *web language* at all. Even so, web applications and web tools in general are among the most common uses of Ruby.

Not only can you write your own SMTP server, FTP daemon, or Web server in Ruby, but you can also use Ruby for more usual tasks such as CGI programming or as a replacement for PHP.

Please spend few minutes with CGI Programming Tutorial for more detail on CGI Programming.

Writing CGI Scripts

The most basic Ruby CGI script looks like this −

```
#!/usr/bin/ruby

puts "HTTP/1.0 200 OK"
puts "Content-type: text/html\n\n"
puts "<html><body>This is a test</body></html>"
```

If you call this script *test.cgi* and uploaded it to a Unix-based Web hosting provider with the right permissions, you could use it as a CGI script.

Here when *test.cgi* is requested from a Web browser, the Web server looks for *test.cgi* on the Web site, and then executes it using the Ruby interpreter. The Ruby script returns a basic HTTP header and then returns a basic HTML document.

Using cgi.rb

Ruby comes with a special library called **cgi** that enables more sophisticated interactions than those with the preceding CGI script.

Let's create a basic CGI script that uses cgi −

```
#!/usr/bin/ruby

require 'cgi'
cgi = CGI.new

puts cgi.header
puts "<html><body>This is a test</body></html>"
```

Here, you created a CGI object and used it to print the header line for you.

HTTP protocol is a stateless protocol. But for a business website, it needs to keep session information between different pages.

If the user site registration process needs to jump page, but want to ensure that the information is not lost before filling.

In this case Cookie good to help us solve the problem.

Cookie How does it work?

Almost all the web designers during the design of the site use the Cookie, because they want to give the user browsing the site to provide a more friendly, human culture browsing environment, but also to more accurately collect visitor information. In many situations, using cookies is the most efficient method of remembering and tracking preferences, purchases, commissions, and other information required for better visitor experience or site statistics.

How It Works?

Your server sends some data to the visitor's browser in the form of a cookie. The browser may accept the cookie. If it does, it is stored as a plain text record on the visitor's hard drive. Now, when the visitor arrives at another page on your site, the cookie is available for retrieval. Once retrieved, your server knows/remembers what was stored.

Cookies are a plain text data record of five variable-

Attribute collection

- **Expires** − The date the cookie will expire. If this is blank, the cookie will expire when the visitor quits the browser.
- **Domain** − The domain name of your site.
- **Path** − The path to the directory or web page that sets the cookie. This may be blank if you want to retrieve the cookie from any directory or page.
- **Secure** − If this field contains the word "secure", then the cookie may only be retrieved with a secure server. If this field is blank, no such restriction exists.
- **Name = Value** − Cookies are set and retrieved in the form of key and value pairs.

Ruby processing Cookies

You can create an object called cookie and store text messages, send the information to the browser, call CGI.out set cookie header:

```
#!/usr/bin/ruby

require "cgi"
cgi = CGI.new("html4")
cookie = CGI::Cookie.new('name' => 'mycookie',
                         'value' => 'Zara Ali',
                         'expires' => Time.now + 3600)
cgi.out('cookie' => cookie) do
   cgi.head + cgi.body { "Cookie stored" }
end
```

Then we go back to this page, and look for cookie values as follows:

```
#!/usr/bin/ruby

require "cgi"
cgi = CGI.new("html4")
cookie = cgi.cookies['mycookie']
cgi.out('cookie' => cookie) do
   cgi.head + cgi.body { cookie[0] }
end
```

CGI :: Cookie object contains an instance of the following parameters:

parameter	description
name	It specifies the name of the cookie.
value	The predetermined value of the cookie.
expire	Provisions of the cookie.
path	Provisions cookie server path.
domain	Provisions of cookie domain.
secure	Specifies whether connections to transfer cookie over a secure HTTPS.

SOAP:

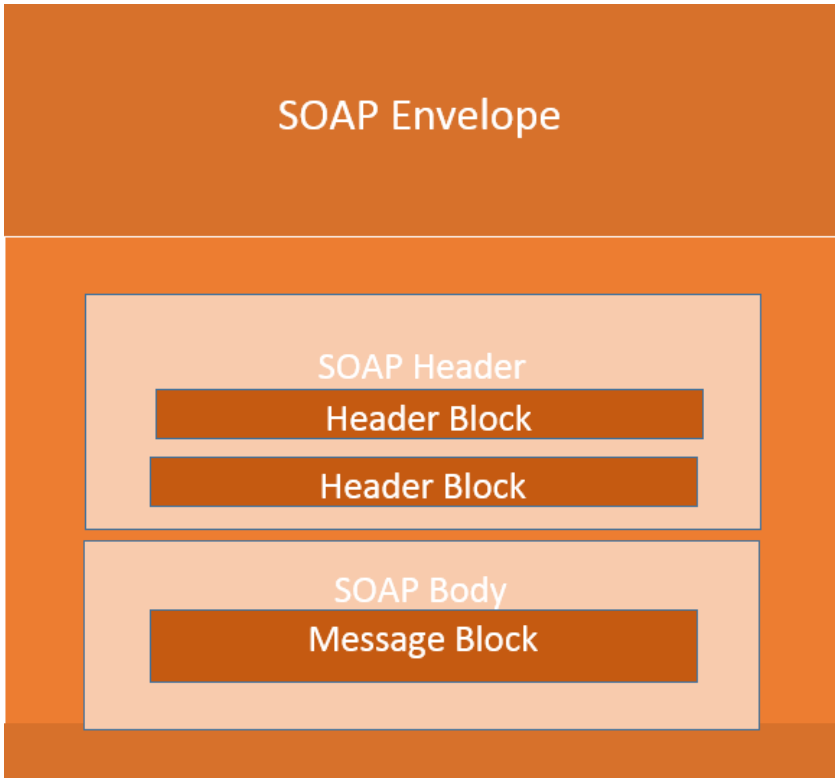
SOAP (Simple Object Access Protocol) is a messaging protocol that allows programs running on different operating systems to communicate with each other over a network. It is based on XML (eXtensible Markup Language) and typically uses HTTP (Hypertext Transfer Protocol) as the underlying transport protocol.

SOAP advantages in brief:

1. Platform and language independence.
2. Extensibility.
3. Standardization.
4. Firewall and proxy friendly.
5. Built-in error handling.
6. Support for web services standards.
7. Widely adopted.

SOAP Building Blocks

The SOAP specification defines something known as a "**SOAP message**" which is what is sent to the web service and the client application.



The SOAP message is nothing but a mere XML document which has the below components.

- An Envelope element that identifies the XML document as a SOAP message – This is the containing part of the SOAP message and is used to encapsulate all the details in the SOAP message. This is the root element in the SOAP message.

```
<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-
envelope/"
soap:encodingStyle="http://www.w3.org/2003/05
/soap-encoding">
  ...
  Message information goes here
  ...
</soap:Envelope>
```

- A Header element that contains header information – The header element can contain information such as authentication credentials which can be used by the calling application. It can also contain the definition of complex types which could be used in the SOAP message. By default, the SOAP message can contain parameters which could be of simple types such as strings and numbers, but can also be a complex object type.

```
<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-
envelope/"
soap:encodingStyle="http://www.w3.org/2003/05
/soap-encoding">

<soap:Header>
  <m:Trans xmlns:m="https://www.w3schools.com
/transaction/"
  soap:mustUnderstand="1">234
  </m:Trans>
</soap:Header>
...
...
</soap:Envelope>
```

- A Body element that contains call and response information – This element is what contains the actual data which needs to be sent between the web service and the calling application. Below is an SOAP web service example of the SOAP body which actually works on the complex type defined in the header section. Here is the response of the Tutorial Name and Tutorial Description that is sent to the calling application which calls this web service.

```
<soap:Body>
   <GetTutorialInfo>
         <TutorialName>Web
Services</TutorialName>
         <TutorialDescription>All about web
services</TutorialDescription>
   </GetTutorialInfo>
</soap:Body>
```

SOAP Message Structure

One thing to note is that SOAP messages are normally auto-generated by the web service when it is called.

Whenever a client application calls a method in the web service, the web service will automatically generate a SOAP message which will have the necessary details of the data which will be sent from the web service to the client application.

As discussed in the previous topic of this SOAP tutorial, a simple SOAP Message has the following elements –

- The Envelope element
- The header element and
- The body element
- The Fault element (Optional)

```
<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soa
p-envelope/"
soap:encodingStyle="http://www.w3.org/200
3/05/soap-encoding">

<soap:Header>
```

```
...
</soap:Header>

<soap:Body>
...
  <soap:Fault>
  ...
  </soap:Fault>
</soap:Body>

</soap:Envelope>
```

The Fault message

When a request is made to a SOAP web service, the response returned can be of either 2 forms which are a successful response or an error response. When a success is generated, the response from the server will always be a SOAP message. But if SOAP faults are generated, they are returned as “HTTP 500” errors.

The SOAP Fault message consists of the following elements.

1. **<faultCode>**– This is the code that designates the code of the error. The fault code can be either of any below values
    1. SOAP-ENV:VersionMismatch – This is when an invalid namespace for the SOAP Envelope element is encountered.
    2. SOAP-ENV:MustUnderstand – An immediate child element of the Header element, with the mustUnderstand attribute set to “1”, was not understood.
    3. SOAP-ENV:Client – The message was incorrectly formed or contained incorrect information.
    4. SOAP-ENV:Server – There was a problem with the server, so the message could not proceed.
2. **<faultString>** – This is the text message which gives a detailed description of the error.
3. **<faultActor> (Optional)**– This is a text string which indicates who caused the fault.

4. **<detail>(Optional)** – This is the element for application-specific error messages. So the application could have a specific error message for different business logic scenarios.

Example for Fault Message

An example of a fault message is given below. The error is generated if the scenario wherein the client tries to use a method called TutorialID in the class GetTutorial.

The below fault message gets generated in the event that the method does not exist in the defined class.

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP:Envelope
xmlns:SOAP
="http://schemas.xmlsoap.org/soap/envelop
e/"
xmlns:xsi="http://www.w3.org/1999/XMLSche
ma-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSche
ma">
      <SOAP:Body>
         <SOAP:Fault>
         <faultcode
xsi:type="xsd:string">SOAP-
ENV:Client</faultcode>
        <faultstring
xsi:type="xsd:string">
            Failed to locate method
(GetTutorialID) in class (GetTutorial)
         </faultstring>
    </SOAP:Fault>
   </SOAP:Body>
</SOAP:Envelope>
```

Describe about the lexical and syntactic structure of Ruby with examples

lexical structure： The lexical structure of a programming language defines how the code is organized and structured at the most basic level, including the rules for writing identifiers, keywords, literals, and operators. Let's explore the lexical structure of Ruby with examples:

1. 1.Identifiers:
- An identifier is a name given to a variable, method, class, or other programming elements.
- Rules for identifiers in Ruby:
- Must start with a lowercase letter or an underscore (_).
- Can contain letters, digits, and underscores.
- Case-sensitive.
- Examples:
    - name
    - _count
    - user_id
2. Keywords:
    - Keywords are reserved words in Ruby that have special meanings and cannot be used as identifiers.
    - Examples of Ruby keywords:

```
def
if
class
end
```

1. 3.Literals:
- Literals are representations of fixed values in code.
- Examples of Ruby literals:
- Integer literals: `42`, `-10`, `0xFF` (hexadecimal), `0b101` (binary).
- Floating-point literals: `3.14`, `-0.5`, `1.0e-5`.
- String literals: `"Hello"`, `'World'`, `%q{Ruby}`.
- Boolean literals: `true`, `false`.

- Array literals: `[1, 2, 3]`, `%w(apple banana)`.
- Symbol literals: `:name`, `:success`.
- Regular expression literals: `/pattern/i`.
-

2. 4.Operators:

- Ruby provides various operators for performing operations on data.
- Examples of Ruby operators:

```
+ - * / % # Arithmetic operators
== != > < >= <= # Comparison operators
&& || ! # Logical operators
= += -= *= /= # Assignment operators
```

5.Comments:

- Comments are used for documentation and explanation, and they are ignored by the Ruby interpreter.
- Single-line comments start with a hash (#) symbol.
- Examples:

```
# This is a single-line comment.

# You can write comments to explain your code.
```

Understanding the lexical structure is important as it forms the foundation for writing valid and meaningful code in Ruby. By following the rules and guidelines of the lexical structure, you can create well-formed Ruby programs.

syntactic structure:

The syntactic structure of a programming language defines how statements, expressions, control structures, and other language constructs are organized and structured. Let's explore the syntactic structure of Ruby with examples:

1.Statements and Expressions:

- Statements are instructions that perform an action or control the flow of execution. Expressions produce values.
- Examples of Ruby statements and expressions:

```
x = 5 # Assignment statement
sum = x + 3 # Expression with assignment
puts "Hello, World!" # Method invocation statement
```

2.Control Structures:

- Ruby provides various control structures to conditionally execute code or repeat code blocks.
- Examples of Ruby control structures:
- if-else statement:

```
if condition
  # code to execute if condition is true
else
```

```
  # code to execute if condition is false
end
```

1.
-
- while loop:

```
while condition
  # code to execute repeatedly while condition is true
end
```

for loop:

```
for element in collection
  # code to execute for each element in the collection
end
```

- case statement:

```
case expression
when value1
  # code to execute if expression equals value1
when value2
  # code to execute if expression equals value2
else
  # code to execute if expression does not match any value
end
```

2. 3.Methods:
- Methods in Ruby encapsulate reusable code blocks that can be invoked with specific arguments.
- Examples of Ruby methods:

```
def greet(name)
  puts "Hello, #{name}!"
end

greet("Alice")  # Method invocation
```

3. 4.Classes and Objects:
- Ruby is an object-oriented language, and classes define blueprints for creating objects.
- Examples of Ruby classes and objects:

```
class Person
  def initialize(name)
  @name = name
  end

  def say_hello
    puts "Hello, #{@name}!"
  end
```

```
end

person = Person.new("Alice") # Object creation
person.say_hello             # Object method invocation
```

-

2. Blocks:

- Blocks are code enclosed within `{}` or `do..end`, used to group statements and pass them to methods.
- Examples of Ruby blocks:

```
5.times do |i|
  puts "Iteration #{i}"
end
```

Error Handling:

1. Ruby provides exception handling mechanisms to handle and recover from errors or exceptional situations.
2. Examples of Ruby error handling:

```
begin
  # Code that may raise an exception
rescue ExceptionType
  # Code to handle the exception
ensure
  # Code to execute regardless of whether an exception occurred or not
end
```

Understanding the syntactic structure of Ruby allows you to write well-formed and structured code. By using the appropriate syntax and organizing your code effectively, you can create readable and maintainable Ruby programs.

what is web service's? briefly explain the types of web services in ruby.

Web services are software systems that enable different applications to communicate and exchange data over the internet. They provide a standardized way for different platforms and technologies to interact with each other, regardless of the programming languages or operating systems used.

Types of Web Services

There are two types of web services:

- RESTful Web Servies
- SOAP Web Services

RESTful Web Services

REST stands for **REpresentational State Transfer**. It is developed by **Roy Thomas Fielding** who also developed HTTP. The main goal of RESTful web services is to make web services **more effective**. RESTful web services try to define services using the different concepts that are already present in HTTP. REST is an **architectural approach**, not a protocol.

It does not define the standard message exchange format. We can build REST services with both XML and JSON. JSON is more popular format with REST. The **key abstraction** is a resource in REST. A resource can be anything. It can be accessed through a **Uniform Resource Identifier (URI)**. For example:

The resource has representations like XML, HTML, and JSON. The current state is captured by representational resource. When we request a resource, we provide the representation of the resource. The important methods of HTTP are:

- **GET:** It reads a resource.
- **PUT:** It updates an existing resource.
- **POST:** It creates a new resource.
- **DELETE:** It deletes the resource.

For example, if we want to perform the following actions in the social media application, we get the corresponding results.

POST /users: It creates a user.

GET /users/{id}: It retrieve the detail of one user.

GET /users: It retrieve the detail of all users.

DELETE /users: It delete all users.

DELETE /users/{id}: It delete a user.

GET /users/{id}/posts/post_id: It retrieve the detail of a specific post.

POST / users/{id}/ posts: It creates a post for a user.

GET /users/{id}/post: Retrieve all posts for a user

HTTP also defines the following standard status code:

- **404:** RESOURCE NOT FOUND
- **200:** SUCCESS
- **201:** CREATED
- **401:** UNAUTHORIZED
- **500:** SERVER ERROR

RESTful Service Constraints

- There must be a service producer and service consumer.
- The service is stateless.
- The service result must be cacheable.
- The interface is uniform and exposing resources.
- The service should assume a layered architecture.

Advantages of RESTful web services

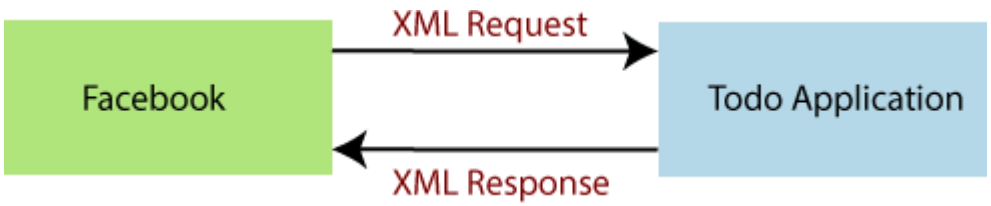
- RESTful web services are **platform-independent**.
- It can be written in any programming language and can be executed on any platform.
- It provides different data format like **JSON, text, HTML,** and **XML**.
- It is fast in comparison to SOAP because there is no strict specification like SOAP.
- These are **reusable**.
- These are **language neutral**.

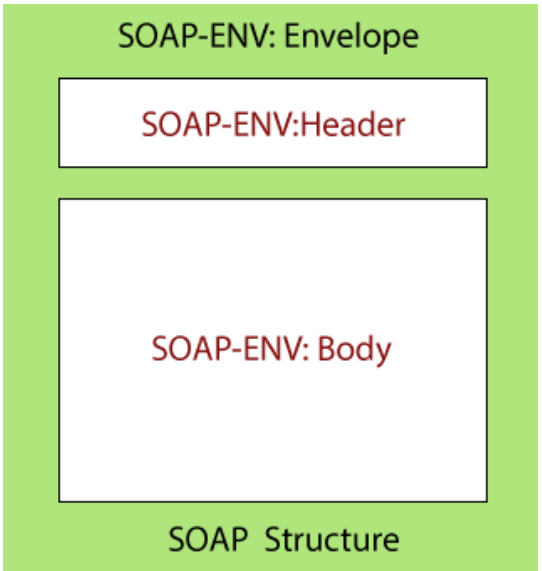
SOAP Web Services

REST defines an architectural approach whereas SOAP poses a restriction on the format of the XML. XML transfer data between the service provider and service consumer. Remember that SOAP and REST are not **comparable**.

SOAP: SOAP acronym for **Simple Object Access Protocol**. It defines the standard XML format. It also defines the way of building web services. We use Web Service Definition Language (WSDL) to define the format of **request XML** and the **response XML**.

For example, we have requested to access the **Todo** application from the **Facebook** application. The Facebook application sends an XML request to the Todo application. Todo application processes the request and generates the XML response and sends back to the Facebook application.





If we are using SOAP web services, we have to use the **structure** of SOAP.In the above figure, the **SOAP-Envelope** contains a **SOAP-Header** and **SOAP-Body**. It contains meta-information needed to identify the request, for example, authentication, authorization, signature, etc. SOAP-Header is optional. The **SOAP- Body** contains the real XML content of request or response. In case of an error, the response server responds back with SOAP-Fault.

Let's understand the SOAP XML request and response structure.

XML Request

```
<Envelop xmlns=?http://schemas.xmlsoap.org/soap/envelop/?>
   <Body>
      <getCourseDetailRequest xmlns=?http://udemy.com/course?>
         <id>course1</id>
      <getCourseDetailRequest>
   </Body>
</Envelop>
```

XML Response

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV=?http://schemas.xmlsoap.org/soap/envelope/?>
      <SOAP-ENV:Header />          <!?empty header-->

      <SOAP-ENV:Body>          <!?body begin-->
         <ns2:getCourseDetailsResponse xmlns:ns2=?http://in28mi> <!--
content of the response-->
            <ns2:course>
             <ns2:id>Course1</ns2:id>
             <ns2:name>Spring<ns2:name>
             <ns2:description>10 Steps</ns1:description>
             </ns2:course>
             </ns2:getCourseDetailResponse>
         </SOAP-ENV:Body>      <!?body end-->
</SOAP-ENV:Envelope>
```

Points to remember

- SOAP defines the format of **request** and **response**.
- SOAP does not pose any restriction on transport. We can either use **HTTP** or **MQ** for communication.
- In SOAP, service definition typically done using **Web Service Definition Language (WSDL)**. WSDL defines **Endpoint, All Operations, Request Structure,** and **Response Structure**.

The **Endpoint** is the connection point where HTML or ASP pages are exposed. It provides the information needed to address the Web Service endpoint. The operations are the services that are allowed to access. Request structure defines the structure of the request, and the response structure defines the structure of the response.

UNIT-2

RUBY OBJECTS IN C

Ruby is a dynamic, object-oriented programming language known for its simplicity and flexibility. Ruby objects play a central role in the language and are created and manipulated using various constructs and methods. While Ruby itself is primarily implemented in C, understanding how Ruby objects are represented in C can provide insights into the underlying mechanisms of the language.

Ruby objects in C are represented using a data structure called VALUE, which is a pointer to a heap-allocated object in the Ruby interpreter. In C, Ruby objects are manipulated using a set of macros and functions provided by the Ruby API. To work with Ruby objects in C, you first need to initialize the Ruby interpreter using the ruby_init() function. This function initializes the interpreter and sets up the Ruby runtime environment, including the global object space, class hierarchy, and garbage collector. Once the interpreter has been initialized, you can create and manipulate Ruby objects using functions like rb_str_new_cstr() to create a new string object, rb_int_new() to create a new integer object, and rb_ary_new() to create a new array object. These functions return a VALUE pointer to the newly-created object. You can also call Ruby methods on objects from C code using the rb_funcall() function. This function takes a VALUE pointer to the object to call the method on, a symbol representing the name of the method to call, and any arguments to pass to the method. The function returns a VALUE pointer to the result of the method call. In addition to creating and manipulating Ruby objects, you can also define new Ruby classes and methods in C using the Ruby API. This involves creating a new class object using the rb_define_class() function, defining methods on the class using the rb_define_method() function, and registering the class with the Ruby interpreter using the rb_define_module() or rb_define_module_under() functions.

In C, objects are typically represented using structures. Similarly, Ruby objects in C are represented using a structure called `VALUE`. The `VALUE` structure is defined in the Ruby source code as follows:

```
typedef unsigned long VALUE;
```

The `VALUE` structure serves as a generic container for Ruby objects. It can store different types of values, including integers, pointers, and references to other objects. This flexibility is achieved through a technique called "tagging" or "boxing," where the lower bits of the `VALUE` structure are used to store information about the type and properties of the object it represents.

In Ruby, every object is an instance of a class. In C, the `VALUE` structure represents both the object itself and its class. The class information is stored in the `VALUE` structure by using a technique called "object header" or "RVALUE" in the Ruby implementation.

The object header contains various fields, including the class information, flags, and additional metadata about the object. The class information stored in the object header allows the Ruby interpreter to determine the object's class and perform appropriate operations on it.

To access the object's class in C, you can use the `RBASIC_CLASS` macro, which extracts the class information from the `VALUE` structure. Here's an example:

```
#include "ruby.h"

void print_object_class(VALUE obj) {

  VALUE obj_class = RBASIC_CLASS(obj);

  const char* class_name = rb_class2name(obj_class);

  printf("Object class: %s\n", class_name);

}

int main() {

  ruby_init();  // Initialize the Ruby interpreter

VALUE string_obj = rb_str_new_cstr("Hello, Ruby!");  // Create a Ruby string object

  print_object_class(string_obj);

  ruby_cleanup(0);  // Clean up the Ruby interpreter

  return 0;

}
```

In the above example, we include the "ruby.h" header file, which provides the necessary functions and macros to work with Ruby objects in C. We initialize the Ruby interpreter using `ruby_init()` and create a Ruby string object using `rb_str_new_cstr()`. The `print_object_class()` function extracts the object's class using `RBASIC_CLASS` macro and then retrieves the class name using `rb_class2name()`. Finally, we print the class name to the console.

It's important to note that directly working with Ruby objects in C requires a good understanding of the Ruby C API and its conventions. The API provides a wide range of functions and macros to create, manipulate, and interact with Ruby objects, classes, and modules in C.

Discuss in detail about Ruby type system

Ruby is a dynamically-typed language, which means that the data type of a variable is determined at runtime, rather than at compile-time. This allows for greater flexibility and ease of use, but also requires extra care to ensure that types are used correctly. In Ruby, every value is an object, and every object has a class. Classes define the attributes and methods that objects of that class can have. Ruby also has modules, which are similar to classes but cannot be instantiated. Modules can be used to define common behavior that can be included in multiple classes.

1.Dynamic Typing: In Ruby, you don't need to explicitly declare the type of a variable. The type is inferred based on the value assigned to it. Variables can hold objects of any type, and they can be reassigned to different types during runtime. For example:

```
x = 10       # x is an integer
x = "hello"  # x is now a string
```

2.Object-Oriented: Everything in Ruby is an object, including primitive types like integers and booleans. Objects have classes that define their behavior and provide methods that can be called on them. Ruby supports single inheritance, allowing classes to inherit from one superclass, but it also includes modules that provide a mechanism for multiple inheritance through mixins.

3.Duck Typing: Ruby follows the principle of "duck typing," which focuses on the object's behavior rather than its specific type. If an object quacks like a duck (responds to the same set of methods), then it can be treated as a duck. This means that you can invoke methods on objects without explicitly checking their types. If the method exists, it will be executed; otherwise, it will raise an error. This flexibility enables polymorphism and code reuse. For example:

```
def print_info(object)
puts object.info
end
class Person
def info
"I am a person."
end
end
class Car
def info
"I am a car."
end
end
print_info(Person.new)  # Output: "I am a person."
print_info(Car.new)     # Output: "I am a car."
```

4.Strong Typing: Ruby is a strongly typed language, meaning that it performs type checking during execution. Operations and method calls must be compatible with the object's type, or an error will be raised. However, Ruby provides many built-in conversion methods and implicit type coercion to handle common type conversions automatically.

5.Nil and Boolean Values: Ruby has a special value called `nil`, which represents the absence of an object. It is often used to indicate the absence of a valid value. Additionally, Ruby treats `false` and `nil` as false in conditional statements, and all other values as true.
6.Type Checking: Ruby provides mechanisms to check the type of an object at runtime. You can use methods like `is_a?`, `kind_of?`, or the `instance_of?` methods to perform type checks. For example:

```
value = 42
puts value.is_a?(Integer)  # Output: true
```

7.Open Classes: One unique aspect of Ruby's type system is the ability to modify classes and objects dynamically. You can add methods to existing classes or modify existing methods, even at runtime. This feature, known as "monkey patching," allows you to extend or change the behavior of built-in or third-party classes.

```
class String
def exclamation
self + "!"
end
end
puts "Hello".exclamation  # Output: "Hello!"
```

These are some of the key characteristics of the Ruby type system. Its dynamic and object-oriented nature, combined with duck typing, make Ruby a highly expressive and flexible language, enabling developers to write concise and powerful code.

7

EMBEDDED RUBY INTERPRETER

An embedded Ruby interpreter refers to the integration of the Ruby programming language into a host application or system. It allows developers to incorporate and execute Ruby code within their application, providing a seamless combination of Ruby's expressive and dynamic features with the functionality of the host application.

When an application embeds a Ruby interpreter, it gains the ability to execute Ruby scripts, interact with Ruby objects, and leverage the extensive Ruby ecosystem, including libraries and frameworks. This integration enables developers to extend the functionality of their application using Ruby or provide scripting capabilities to end users.

The embedded Ruby interpreter typically provides a programming interface or API that allows the host application to interact with the Ruby runtime. This API exposes functions and methods for executing Ruby code, manipulating Ruby objects, and exchanging data between the host application and the Ruby interpreter.

Benefits of using an embedded Ruby interpreter include:

1. Dynamic Extensibility: Embedding Ruby allows developers to dynamically extend the functionality of their application without modifying or recompiling the entire codebase. New features or behavior can be implemented in Ruby scripts and seamlessly integrated into the host application.
2. Rapid Prototyping and Scripting: By embedding Ruby, developers can provide a scripting interface that enables end users or other developers to write scripts to automate tasks, customize behavior, or add functionality to the application. This empowers users with greater flexibility and reduces the need for recompilation.
3. Access to Ruby Libraries and Frameworks: Embedding Ruby provides access to the rich ecosystem of Ruby libraries, frameworks, and gems. This allows developers to leverage existing Ruby code and take advantage of the vast range of available functionality without reinventing the wheel.
4. Interoperability: An embedded Ruby interpreter facilitates interoperability between the host application and Ruby code. Data can be exchanged between the host application and Ruby, enabling seamless integration and communication between the two environments.

Embedded Ruby interpreters are commonly used in various scenarios, including game development, automation tools, web servers, and graphical user interface (GUI) frameworks. Examples of embedded Ruby interpreters include Ruby's C API, which allows embedding Ruby in C/C++ applications, and frameworks like mruby, which provide lightweight and embeddable Ruby interpreters optimized for resource-constrained environments.

Overall, embedding a Ruby interpreter provides developers with the ability to leverage Ruby's expressive power, dynamic nature, and vast ecosystem within their own applications, expanding functionality and customization options.

mruby is a lightweight implementation of the Ruby programming language designed for embedded systems or situations where a full-fledged Ruby interpreter might be too resource-intensive. It allows you to embed Ruby scripts directly into your C application.

Here's an example of how you can embed an mruby interpreter and execute Ruby code:

Set up mruby: First, you need to download and set up the mruby library in your C application. You can find the necessary resources and instructions on the official mruby website.

1.Include the necessary headers: In your C application, include the mruby headers to access the mruby API:

2.Initialize the mruby interpreter: Before executing any Ruby code, you need to initialize the mruby interpreter:

3.Load and execute Ruby code: You can load and execute Ruby code using the `mrb_load_string()` function

The `mrb_load_string()` function takes the mruby interpreter (`mrb`) and a string containing the Ruby code. It returns a value representing the result of the execution.

If an exception occurs during the execution of the Ruby code, you can use `mrb_print_error()` to print the error message.

4.Convert Ruby values to C values (if needed): If you want to access the result of the Ruby code in your C application, you may need to convert it from a mrb_value to a C value. You can use the provided conversion functions, such as `mrb_string_value_cstr()`, to extract the desired data from the mrb_value.

5.Clean up and shut down: Once you're done executing Ruby code, it's important to clean up and shut down the mruby interpreter:
The `mrb_close()` function releases resources used by the mruby interpreter.
Here's the complete example:

```
#include <mruby.h>

#include <mruby/compile.h>

#include <stdio.h>



int main() {

  mrb_state* mrb = mrb_open();



  mrb_value result = mrb_load_string(mrb, "puts 'Hello, Ruby from C!'");



  if (mrb->exc) {

    mrb_print_error(mrb);

  }



  mrb_value str_result = mrb_funcall(mrb, result, "to_s", 0);

  const char* result_str = mrb_string_value_cstr(mrb, &str_result);



  printf("Result: %s\n", result_str);



  mrb_close(mrb);
```

```
  return 0;

}
```

When you compile and run this C application with mruby, it will embed the mruby interpreter, execute the specified Ruby code, and print "Hello, Ruby from C!" as the result.

This example showcases how to embed an mruby interpreter in a C application, execute Ruby code, and interact with the results in C. It demonstrates the ability to seamlessly integrate Ruby scripts within your C application using a lightweight

Example:2

```
#include <stdio.h>

#include <ruby.h>



int main() {

    // Initialize the Ruby interpreter

    ruby_init();



    // Load and execute Ruby code

    rb_eval_string("puts 'Hello, Ruby from C!'");



    // Clean up and shut down the Ruby interpreter

    ruby_cleanup(0);

    ruby_stop(0);



    return 0;

}
```

In this example, we're embedding the Ruby interpreter in a C application using the Ruby C API. The steps involved are as follows:

Include the necessary headers: Include the `<ruby.h>` header to access the Ruby C API functions.

Initialize the Ruby interpreter: Call `ruby_init()` to initialize the Ruby interpreter.
Load and execute Ruby code: Use the `rb_eval_string()` function to load and execute Ruby code. In this example, we're executing a simple Ruby code that prints "Hello, Ruby from C!".
Clean up and shut down: After executing the Ruby code, call `ruby_cleanup()` and `ruby_stop()` to clean up and shut down the Ruby interpreter.

When you compile and run this C application, it will embed the Ruby interpreter, execute the specified Ruby code, and print "Hello, Ruby from C!" as the output.

This example demonstrates a basic use case of embedding a Ruby interpreter in a C application using the Ruby C API. It showcases how you can execute Ruby code and leverage Ruby's capabilities within your C application.

UNIT-3

CHARCTERISTICS:

All scripting languages are programming languages. The scripting language is basically a language where instructions are written for a run time environment. They do not require the compilation step and are rather interpreted. It brings new functions to applications and glue complex system together. A scripting language is a programming language designed for integrating and communicating with other programming languages.

Advantages of scripting languages:

- Rapid Development: Scripting languages prioritize quick prototyping and development, allowing developers to build applications faster and iterate more rapidly.
- Ease of Use: Scripting languages have simple syntax and high-level abstractions, making them easy to learn and use for both beginners and experienced developers.
- Increased Productivity: The concise syntax, extensive libraries, and built-in functions of scripting languages help developers write code more efficiently, resulting in increased productivity.
- Enhanced Readability: Scripting languages often prioritize readability, making code easier to understand and maintain. This improves collaboration among team members and reduces the time spent on debugging and troubleshooting.
- Platform Independence: Many scripting languages are platform-independent, allowing scripts to run on multiple operating systems without modification. This enhances portability and reduces compatibility issues.

Characteristics of a scripting language:

1. Interpreted Execution: Scripting languages are typically interpreted rather than compiled. They are executed directly by an interpreter without the need for a separate compilation step. This allows for rapid development and immediate execution of scripts.
2. Dynamic Typing: Scripting languages often employ dynamic typing, where variables are not bound to a specific data type. They allow variables to hold values of different types during runtime, providing flexibility and ease of use. Dynamic typing eliminates the need for explicit type declarations and enables more concise code.
3. High-Level Abstractions: Scripting languages provide high-level abstractions and built-in data structures that simplify common programming tasks. They offer libraries and frameworks that handle complex operations such as file I/O, string manipulation, regular expressions, and network communication. This allows developers to focus on solving specific problems without worrying about low-level implementation details.
4. Rapid Development: Scripting languages prioritize rapid development and ease of use. They often have concise syntax, expressive constructs, and extensive libraries that enable developers to quickly prototype ideas and build applications with fewer lines of code. This characteristic makes scripting languages popular for scripting tasks, automation, and web development.
5. Scriptability and Flexibility: Scripting languages are designed to be highly scriptable, allowing users to write scripts that control and automate various aspects of a system or application. They provide hooks and APIs for system interaction, enabling users to customize and extend functionality easily. This characteristic makes scripting languages ideal for tasks such as system administration, configuration management, and testing.
6. Portability: Scripting languages are typically designed to be platform-independent and portable. Scripts written in a scripting language can be executed on different operating systems and architectures without requiring modifications. This portability allows scripts to be easily shared and run on various systems, enhancing their versatility and usability.
7. Dynamic Memory Management: Scripting languages abstract away low-level memory management from developers by employing automatic memory management techniques like garbage collection. This relieves developers from the burden of manual memory allocation and deallocation, reducing the risk of memory-related errors such as memory leaks and buffer overflows.
8. Integration with other Languages: Scripting languages often provide mechanisms to interact with code written in other languages. They offer interfaces and bindings to enable interoperability with existing libraries and systems written in different languages. This characteristic allows developers to leverage the strengths of multiple languages and reuse existing code.

Overall, scripting languages prioritize ease of use, rapid development, and automation. They provide a higher level of abstraction, dynamic typing, and built-in functionality to simplify common programming tasks. These characteristics make scripting languages suitable for a wide range of applications, including automation, web development, system administration, and prototyping.

DIFF BTW SCRIPT AND PROG:

	Scripting Language	Programming Language
1.	A scripting language is a language that uses a naive method to bring codes to a runtime environment	A Programming language is a language which is used by humans to navigate their communication with computers.
2.	These are made for a particular runtime environment.	Programming languages are of three types -: • low-level Programming language • Middle-level Programming language • High-level Programming language
3.	They are used to create dynamic web applications	Programming languages are used to write computer programs.
4.	Scripting languages contain different libraries	They are high-speed languages.
5.	Example -: Bash, Ruby, Python, JavaScript etc.	Example -: C++, Java, PHP High-level etc.
6.	Scripting languages can be easily ported among various operating systems.	Programming languages are translation free languages
7.	These languages requires a host.	These languages are self executable.
8.	Do not create a ***.exe*** file.	These generate ***.exe*** files.
9.	Most of the scripting languages are interpreted language.	Most of the programming languages are compiled languages.
10.	All the scripting languages are programming languages.	All the programming languages are not scripting languages.
11.	It is easier to learn than programming language.	It can take significant amount of time to learn.
12.	It is less code intensive when compared with programming language.	It is code intensive.
13.	It does not create any binary files.	It does creates binary files.

14.	It is easy for the beginner to write and understand the code.	It is difficult for the beginner to write and understand the code.
15.	It is run inside another program.	It is independently run.
16.	It needs lesser line of codes.	It needs numerous lines of code.
17.	It has low maintenance cost.	It has high maintenance cost.

Applications of Scripting Languages :
1. To automate certain tasks in a program
2. Extracting information from a data set
3. Less code intensive as compared to traditional programming languages

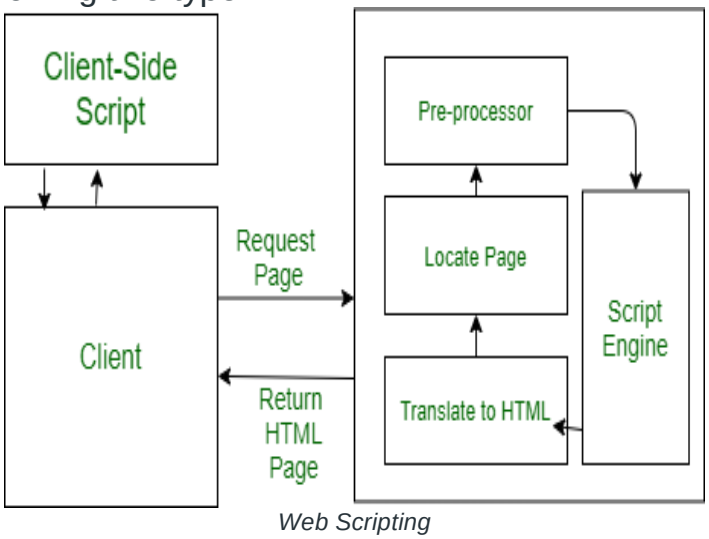
Applications of Programming Languages :
1. They typically run inside a parent program like scripts
2. More compatible while integrating code with mathematical models
3. Languages like JAVA can be compiled and then used on any platform

Web Scripting and its Types:

The process of creating and embedding scripts in a web page is known as **web-scripting**. A script or a computer-script is a list of commands that are embedded in a web-page normally and are interpreted and executed by a certain program or scripting engine.

- Scripts may be written for a variety of purposes such as for automating processes on a local-computer or to generate web pages.
- The programming languages in which scripts are written are called scripting language, there are many scripting languages available today.
- Common scripting languages are VBScript, JavaScript, ASP, PHP, PERL, JSP etc.

Types of Script :
Scripts are broadly of following two type :



Web Scripting

Client-Side Scripts :

1. Client-side scripting is responsible for interaction within a web page. The client-side scripts are firstly downloaded at the client-end and then interpreted and executed by the browser (default browser of the system).
2. The client-side scripting is browser-dependent. i.e., the client-side browser must be scripting enables in order to run scripts
3. Client-side scripting is used when the client-side interaction is used. Some example uses of client-side scripting may be :
   - To get the data from user's screen or browser.
   - For playing online games.
   - Customizing the display of page in browser without reloading or reopening the page.
4. Here are some popular client-side scripting languages VBScript, JavaScript, Hypertext Processor(PHP).

Server-Side Scripts :

1. Server-side scripting is responsible for the completion or carrying out a task at the server-end and then sending the result to the client-end.
2. In server-side script, it doesn't matter which browser is being used at client-end, because the server does all the work.
3. Server-side scripting is mainly used when the information is sent to a server and to be processed at the server-end. Some sample uses of server-scripting can be :
   - Password Protection.
   - Browser Customization (sending information as per the requirements of client-end browser)
   - Form Processing
   - Building/Creating and displaying pages created from a database.
   - Dynamically editing changing or adding content to a web-page.
4. Here are some popular server-side scripting languages PHP, Perl, ASP (Active Server Pages), JSP (Java Server Pages).

Origin of scripting:

The use of the word 'script' in a computing context dates back to the early 1970s,when the originators of the UNIX operating system create the term 'shell script' for sequence of commands that were to be read from a file and follow in sequence as if they had been typed in at the keyword. e.g. an 'AWKscript', a 'perl script' etc.. the name 'script ' being used for a text file that was intended to be executed directly rather than being compiled to a different form of file prior to execution. Other early occurrences of the term 'script' can be found. For example, in a DOS-based system, use of a dial-up connection to a remote system required a communication package that used proprietary language to write scripts to automate the sequence of operations required to establish a connection to a remote system. Note that if we regard a scripts as a sequence of commands to control an application or a device, a configuration file such as a UNIX 'make file' could be regard as a script. However, scripts only become interesting when they have the added value that comes from using programming concepts such as loops and branches.

The universe of scripting languages:

Scripting can be traditional or modern scripting, and Web scripting forms an important part of modern scripting. Scripting universe contains multiple overlapping worlds:

- the original UNIX world of traditional scripting using Perl and Tcl
- the Microsoft world of Visual Basic and Active controls
- the world of VBA for scripting compound documents
- the world of client-side and server-side Web scripting.

The overlap is complex, for example web scripting can be done in VBScript, JavaScript/Jscript, Perl or Tcl. This universe has been enlarged as Perl and Tcl are used to implement complex applications for large organizations e.g Tcl has been used to develop a major banking system, and Perl has been used to implement an enterprisewide document management system for a leading aerospace company

UNIT-4

PACK AND UNPACK:

In Perl, the functions `pack` and `unpack` are used for converting data between binary strings and structured data. These functions are commonly used when working with binary data, such as network protocols or file formats that require precise control over data representation.

1. `pack` Function: The `pack` function converts data into a binary string according to a specified format. It takes a template string that defines the desired format and a list of values to be packed. The template string consists of format specifiers that represent the type and size of the data to be packed.

Syntax

```
pack Expr, List
```

Here's an example that demonstrates the `pack` function:

```
#!/usr/bin/perl

use strict;
use warnings;

my $number = 42;
my $packed_data = pack("N", $number);

print "Packed Data: $packed_data\n";
```

OUTPUT: Packed Data: *

In this example, we have a variable `$number` assigned with the value 42. The `pack` function is used to pack the value into a binary string using the format specifier "N", which represents an unsigned long integer in network byte order. The resulting packed data is stored in the variable `$packed_data`. Finally, we print the packed data.

2. `unpack` Function: The `unpack` function does the reverse operation of `pack`. It takes a template string that describes the format of the packed data and a binary string containing the packed data. It returns a list of unpacked values based on the specified format.

Syntax

unpack TEMPLATE, STRING

Here's an example that demonstrates the `unpack` function:

```
#!/usr/bin/perl

use strict;
use warnings;

my $packed_data = "\x00\x00\x00\x2A";
my ($number) = unpack("N", $packed_data);

print "Unpacked Number: $number\n";
```

OUTPUT: Unpacked Number: 42

In this example, the binary string "\x00\x00\x00\x2A" is unpacked using the format specifier "N". The unpacked value is 42, which is printed as the output.

Please note that the outputs may vary depending on the system's byte order and encoding format. The examples provided assume a system using network byte order and ASCII encoding.

Perl File Handling:

File handling is the most important part in any programming language. A filehandle is an internal Perl structure that associates with a file name.

Perl File handling is important as it is helpful in accessing file such as text files, log files or configuration files.

Perl filehandles are capable of creating, reading, opening and closing a file.

Perl Create File

We are creating a file, **file1.txt** with the help of open() function.

The $fh (file handle) is a scalar variable and we can define it inside or before the open() function. Here we have define it inside the function. The '>' sign means we are opening this file for writing. The **$filename** denotes the path or file location.

Once file is open, use $fh in print statement. The print() function will print the above text in the file.

Now we are closing $fh. Well, closing the file is not required in perl. Your file will be automatically closed when variable goes out of scope.

```
1. my $filename = 'file1.txt';
2. open(my $fh, '>', $filename) or die "Could not open file '$filename' $!";
3. print $fh "Hello!! We have created this file as an example\n";
4. close $fh;
5. print "done\n";
```

Output:

```
done.
```

A file file1.txt will be created in our system.

Perl Open File

We can open a file in following ways:

(<) Syntax

The < sign is used to open an already existing file. It opens the file in read mode.

```
1. open FILE, "<", "fileName.txt" or die $!
```

(>) Syntax

The > sign is used to open and create the file if it doesn't exists. It opens the file in write mode.

```
1. open FILE, ">", "fileName.txt" or die $!
```

The "<" sign will empty the file before opening it. It will clear all your data of that file. To prevent this use (+) sign before ">" or "<" characters.

(+>+<) Syntax

```
1. open FILE, "+<", "fileName.txt" or die $!
2. open FILE, "+>", "fileName.txt" or die $!
```

(>>) Syntax

The >> sign is used to read and append the file content. It places the file pointer at the end of the file where you can append the information. Here also, to read from this file, you need to put (+) sign before ">>" sign.

```
1. open FILE, "<", "fileName.txt" or die $!
```

Perl Read File

You can read a complete file at once or you can read it one line at a time. We'll show an example for both. Opening a file to read is similar to open a file to write. With only one difference that ">" is used to write and "<" is used to read the file.

We have created a file **file1.txt** with the following content:

```
1. This is the First Line.
2. This is the Second Line.
3. This is the Third Line.
4. This is the Fourth Line.
```

To read Single line at a time

First line of file1.txt will be displayed. Content of $row will be printed with "done" to make it clear that we reached at the end of our program.

```
1. use strict;
2. use warnings;
3. my $filename = 'file1.txt';
4. open(my $fh, '<:encoding(UTF-8)', $filename)
5.   or die "Could not open file '$filename' $!";
6. my $row = <$fh>;
7. print "$row\n";
8. print "done\n";
```

Output:

```
This is the First Line.
Done.
```

To read Multi lines at a time

Now we know to read single line from a file. To read multiple lines put $row in a while loop.

Every time, when while loop will reach its condition, it will execute **my $row = <$fh>**. It will read the next line from the file. At the last line, $fh will return undef which is false and loop will terminate.

```
1. use strict;
2. use warnings;
3. my $filename = 'file1.txt';
4. open(my $fh, '<:encoding(UTF-8)', $filename)
5.   or die "Could not open file '$filename' $!";
6. while (my $row = <$fh>) {
7.   chomp $row;
8.   print "$row\n";
9. }
10. print "done\n";
```

Output:

```
This is the First Line.
This is the Second Line.
This is the Third Line.
This is the Fourth Line.
Done.
```

Perl Write File

Through file writing, we'll append lines in the file1.txt. As already stated, new lines will be added at the last of the file.

```
1. open (FILE, ">> file1.txt") || die "problem opening $file1.txt\n";
2. print FILE "This line is added in the file1.txt\n";
3. # FILE array of lines is written here
4. print FILE @lines1;
5. # Another FILE array of lines is written here
6. print FILE "A complete new file is created";
7. # write a second array of lines to the file
8. print FILE @lines2;
```

Output:

```
This line is added in the file1.txt
A complete new file is created
```

Perl Close File

Perl close file is used to close a file handle using close() function. File closing is not compulsory in perl. Perl automatically closes file once the variable is out of scope.

```
1. open FILE1, "file1.txt" or die $!;
2. ...
3. close FILE1;
```

Perl File Handle Operator, <FILEHANDL>

File handle operator is the main method to read information from a file. It is used to get input from user. In scalar context, it returns a single line from the filehandle and in line context, it returns a list of lines from the filehandle.

```
1. print "What is your age?\n";
2. $age = <STDIN>;
3. if($age >= 18)
4. {
5.    print "You are eligible to vote.\n";
6. } else {
7.    print "You are not eligible to vote.\n";
8.     }
```

Output:

```
1. What is your age?
       18
       You are eligible to vote
2.       What is your age?
       16
        You are not eligible to vote.
```

Perl File Handle print() function

The print() function prints back the information given through filehandle operators.

```
1. print "Welcome to my site\n";
```

Output:

```

Welcome to my site
```

Perl Copying a File

We can copy content of one file into another file as it is. First open file1 then open file2. Copy the content of file 1 to file2 by reading its line through a while loop.

```
# Opening file1 to read
open(File1Data, "<file1.txt");
# Opening new file to copy content of file1
open(File2Data, ">file2.txt");
# Copying data from file1 to file2.
while(<File1Data>)
{
   print File2Data $_;
}
close( File1Data );
close( File2Data );
```

Output:

```
done
```

A new file file2.pl will be created in the location where file1.pl exists.

Perl File Test Operators

There are different test operators to check different information about a file. Some of the test operators are as follows:

Test operators	Description
-A	Return total access time of file since the program started.
-b	Check whether file is block device or not.
-B	Check whether it is a binary file.
-c	Check whether file is character device.
-C	Return inode change time of file since the program started.
-d	Check whether file is a directory.
-e	Check whether file exists or not.
-f	Check type of file whether it is regular, symbolic link or other type of file.
-g	Check whether file have setgid bit set.

-k	Check whether file have sticky bit set.
-l	Check if file is a symbolic link. In dos, it always return false.
-M	Return file modification time in days since the program started.
-o	Check if file is owned by an effective uid, in dos, it always return true.
-O	Check if file is owned by a real uid, in dos, it always return true.
-p	Check whether file is a named pipe.
-r	Check whether file is readable or not.
-R	Check whether file is readable by real uid or not. In dos, it is same as -r.
-s	Return the file size in bytes.
-S	Check if file is a socket.
-t	Check if file is opened to a tty (terminal)
-T	Check if file is a text file.
-u	Check if file has setuid bit set.
-w	Check if file is writable or not.
-W	Check if file is writable by real uid/gid.
-x	Check if file can be executed or not.
-X	Check if file can be executed by real uid/gid.
-z	Check if file size is zero or not.

Perl Using File Test Operators

To test different features within Perl, a series of test operators are present. In the given example, we have tested different features of file1.txt. All the outcomes are merged with join() function.

```
1. my $a = "/Users/javatpoint/Desktop/file1.txt";
2. my (@description, $size);
3. if (-e $a)
4. {
5.    push @description, 'binary' if (-B _);
6.    push @description, 'a socket' if (-S _);
7.    push @description, 'a text file' if (-T _);
8.    push @description, 'a block special file' if (-b _);
9.    push @description, 'a character special file' if (-c _);
```

```
10.    push @description, 'a directory' if (-d _);
11.    push @description, 'executable' if (-x _);
12.    push @description, (($size = -s _)) ? "$size bytes" : 'empty';
13.    print "file1.txt is ", join(', ',@description),"\n";
14. }
```

Output:

```
file1.txt is a text file, 67 bytes
```

What are Packages?

The **package** statement switches the current naming context to a specified namespace (symbol table). Thus −

- A package is a collection of code which lives in its own namespace.
- A namespace is a named collection of unique variable names (also called a symbol table).
- Namespaces prevent variable name collisions between packages.
- Packages enable the construction of modules which, when used, won't clobber variables and functions outside of the modules's own namespace.
- The package stays in effect until either another package statement is invoked, or until the end of the current block or file.
- You can explicitly refer to variables within a package using the **::** package qualifier.

Following is an example having main and Foo packages in a file. Here special variable __PACKAGE__ has been used to print the package name.

```
#!/usr/bin/perl

# This is main package
$i = 1;
print "Package name : " , __PACKAGE__ , " $i\n";

package Foo;
# This is Foo package
$i = 10;
print "Package name : " , __PACKAGE__ , " $i\n";

package main;
# This is again main package
$i = 100;
print "Package name : " , __PACKAGE__ , " $i\n";
print "Package name : " , __PACKAGE__ ,  " $Foo::i\n";

1;
```

When above code is executed, it produces the following result −

```
Package name : main 1
Package name : Foo 10
Package name : main 100
```

BEGIN and END Blocks

You may define any number of code blocks named BEGIN and END, which act as constructors and destructors respectively.

```
BEGIN { ... }
END { ... }
BEGIN { ... }
END { ... }
```

- Every **BEGIN** block is executed after the perl script is loaded and compiled but before any other statement is executed.
- Every END block is executed just before the perl interpreter exits.
- The BEGIN and END blocks are particularly useful when creating Perl modules.

Following example shows its usage −

```
#!/usr/bin/perl

package Foo;
print "Begin and Block Demo\n";

BEGIN {
  print "This is BEGIN Block\n"
}

END {
  print "This is END Block\n"
}

1;
```

When above code is executed, it produces the following result −

```
This is BEGIN Block
Begin and Block Demo
This is END Block
```

What are Perl Modules?

A Perl module is a reusable package defined in a library file whose name is the same as the name of the package with a .pm as extension.

A Perl module file called **Foo.pm** might contain statements like this.

```
#!/usr/bin/perl

package Foo;
sub bar {
  print "Hello $_[0]\n"
}

sub blat {
  print "World $_[0]\n"
}
1;
```

Few important points about Perl modules

- The functions **require** and **use** will load a module.
- Both use the list of search paths in **@INC** to find the module.
- Both functions **require** and **use** call the **eval** function to process the code.
- The **1;** at the bottom causes eval to evaluate to TRUE (and thus not fail).

The Require Function

A module can be loaded by calling the **require** function as follows −

```
#!/usr/bin/perl

require Foo;

Foo::bar( "a" );
Foo::blat( "b" );
```

You must have noticed that the subroutine names must be fully qualified to call them. It would be nice to enable the subroutine **bar** and **blat** to be imported into our own namespace so we wouldn't have to use the Foo:: qualifier.

The Use Function

A module can be loaded by calling the **use** function.

```
#!/usr/bin/perl

use Foo;

bar( "a" );
blat( "b" );
```

Notice that we didn't have to fully qualify the package's function names. The **use** function will export a list of symbols from a module given a few added statements inside a module.

```
require Exporter;
@ISA = qw(Exporter);
```

Then, provide a list of symbols (scalars, lists, hashes, subroutines, etc) by filling the list variable named **@EXPORT**: For Example −

```
package Module;

require Exporter;
@ISA = qw(Exporter);
@EXPORT = qw(bar blat);

sub bar { print "Hello $_[0]\n" }
sub blat { print "World $_[0]\n" }
sub splat { print "Not $_[0]\n" }  # Not exported!

1;
```

Creating Internet Ware Applications: The internet is a rich source of information, held on web servers, FTP servers, POP/IMAP mail servers, news servers etc. A web browser can access information on web servers and FTP servers, and clients access mail and news servers. however, this is not the way of to the information: an 'internet-aware' application can access a server and collect the information withoutmanual intervention. For suppose that a website offers 'lookup' facility in which the user a query by filling in a then clicks the 'submit' button . the data from the form in sent to a CGI program on the server(probably written in which retrieves the information, formats it as a webpage, and returns the page to the browser. A perl application can establish a connection to the server, send the request in the format that the browser would use, collectthe returned HTML and then extract the fields that form the answer to the query. In the same way, a perl application can establish a connection to a POP3 mail server and send a request which will reult inthe server returning a message listing the number of currently unread messages. Much of the power of scripting languages comes from the way in which they hide the complexity of operations, and this isparticularly the case when we make use of specialized modules: tasks that might pages of code in C are achieved in few lines. The LWP (library for WWW access in perl) collection of modules is a very good case in point it makes the kind of interaction described above almost trivial.The LWP:: simple module is a interface to web servers.it can be achieved by exploiting modules, LWP::simple we can retrieve the contents of a web page in a statement: use LWP::simple $url=...http://www.somesite.com/index.html..; $page=get($url);

Dirty Hands Internet Programming: Modules like LWP: : Simple and LWP: :User Agent meet the needs of most programmers requiring web access, and there are numerous other modules for other types of Internet access. EX:- Net: : FTP for access to FTP servers Some tasks may require a lower level of access to the network, and this is provided by Perl both in the form of modules(e.g IO: : Socket) and at an even lower level by built-in functions. Support for network programming in perl is so complete that you can use the language to write any conceivable internet application Access to the internet at this level involves the use of sockets, and we explain what a socket is before getting down to details of the programming. Sockets are network communication channels, providing a bi-directional channel between processes on different machines. Sockets were originally a feature of UNIX:other UNIX systems adopted them and the socket became the de facto mechanism of network communication in the UNIX world. The popular Winsock provided similar functionality for Windows, allowing Windows systems to communicate over the network with UNIX systems, and sockets are a built-in feature of Windows 9X and WindowsNT4. From the Perl programmer’s point a network socket can be treated like an open file it is identified by a you write to it with print, and read it from operator. The socket interface is based on the TCP/IP protocol suite, so that all information is handled automatically. In TCP a reliable channel, with automatic recovery from data loss or corruption: for this reason a TCP connection is often described as a virtual circuit. The socket in Perl is an exact mirror of the UNIX and also permits connections using UDP(Unreliable Datagram Protocol).

1. **File system in Perl:** The file system in Perl refers to the way Perl interacts with files and directories on the operating system. Perl provides various built-in functions and modules to handle file operations such as reading, writing, appending, and manipulating file paths.

   Question: How does Perl handle file operations like reading, writing, and manipulating file paths?

2. **Eval in Perl:** The `eval` function in Perl allows you to dynamically evaluate and execute a block of Perl code at runtime. It takes a string argument containing Perl code and executes it as if it were a regular Perl program. This feature is particularly useful when you want to handle errors gracefully or when you need to evaluate code that is generated dynamically.

   Question: What is the purpose of the `eval` function in Perl, and how can it be used?

3. **Glob command in Perl:** The `glob` function in Perl is used to expand filenames or generate a list of files based on a specified pattern. It operates similarly to shell globbing, allowing you to match files using wildcards and retrieve a list of matching filenames.

   Question: How does the `glob` function in Perl work, and what are some use cases for it?

4. **Namespace:** A namespace is a container that holds a collection of symbols, such as variables, functions, or classes, to prevent naming conflicts. It provides a way to organize and differentiate symbols based on their context or origin. Namespaces are widely used in programming languages to maintain code modularity, encapsulation, and avoid naming collisions.

   Question: What is a namespace, and how does it help in managing naming conflicts and organizing code in programming languages?

5. **Hashes in Perl:** In Perl, a hash is an unordered collection of key-value pairs. It is also known as an associative array or dictionary in other programming languages. Hashes provide a convenient way to store and retrieve data based on unique keys. The keys are typically strings, and each key is associated with a corresponding value.

   Question: How are hashes represented in Perl, and what are their primary uses and features?

6. **Security issues in Perl:** Like any programming language, Perl has its own set of security concerns. Common security issues in Perl programs include code injection vulnerabilities, unvalidated input, insecure system commands, inadequate handling of sensitive data, and improper file permissions. It is essential to follow security best practices, sanitize inputs, validate user data, and use secure coding techniques to mitigate these risks.

   Question: What are some common security issues that can arise in Perl programs, and how can developers address them?

7. **Nuts and Bolts in TCL:** The term "nuts and bolts" in TCL refers to the fundamental elements or basic building blocks of the TCL language. These include commands, variables, procedures, control structures, and data types. Understanding the nuts and bolts of TCL is crucial for effectively using the language and creating TCL scripts.

Question: What are the key components or "nuts and bolts" of the TCL language, and how do they contribute to the functionality of TCL scripts?

8. **Dirty Hands Internet Programming in Perl:** The term "Dirty Hands Internet Programming" is not a well-known concept, and there is no specific definition for it in relation to Perl. It seems to suggest a hands-on, practical approach to internet programming that prioritizes functionality over strict adherence to best practices or clean code principles.

Question: Can you provide an explanation or define "Dirty Hands Internet Programming" in the context of Perl?

9. **Procedures in TCL:** In TCL, a procedure is a named block of code that can be called or invoked multiple times within a script. Procedures help in code reusability, modularity, and organizing complex code into smaller,

1. **Difference between a Gem and a Plugin in Ruby:** A gem in Ruby is a packaged library or application that can be easily installed and managed using the RubyGems package manager. It provides reusable code and functionality that can be integrated into Ruby projects. A gem typically follows a specific structure and includes documentation, dependencies, and versioning.

On the other hand, a plugin in Ruby is a piece of code that extends or modifies the behavior of an existing Ruby application or framework. Plugins are often specific to a particular application or framework and are typically designed to be easily integrated or plugged into the existing codebase. They can add new features, modify existing functionality, or provide additional configuration options.

Question: What is the difference between a gem and a plugin in Ruby, and how do they contribute to extending or enhancing Ruby applications?

2. **Explanation of various environment variables in Ruby:** In Ruby, environment variables are special variables that hold information about the runtime environment. They are used to pass configuration settings, system paths, or other relevant data to Ruby programs. Some commonly used environment variables in Ruby include:

- `PATH`: Specifies the directories where the system looks for executables.
- `HOME`: Indicates the user's home directory.
- `RUBYLIB`: Sets additional directories to be included in the Ruby library search path.
- `GEM_HOME` and `GEM_PATH`: Specify the location of Ruby gems.

Question: Can you explain the purpose and usage of various environment variables in Ruby, such as PATH, HOME, RUBYLIB, GEM_HOME, and GEM_PATH?

3. **Method that allocates memory for objects in Ruby:** In Ruby, memory allocation for objects is handled by the Ruby interpreter itself through a process called garbage collection. The garbage collector automatically manages memory and deallocates objects that are no longer in use. Ruby uses various strategies, such as mark-and-sweep or generational garbage collection, to efficiently handle memory allocation and deallocation.

Question: Which method is responsible for allocating memory for objects in Ruby, and how does the garbage collector manage memory deallocation in Ruby?

4. **How Ruby manages memory:** Ruby uses automatic memory management through a garbage collector. The garbage collector tracks objects in memory and identifies those that are no longer referenced by the program. It reclaims memory occupied by these unreferenced objects, freeing it up for reuse.

Ruby's garbage collector uses different algorithms, such as mark-and-sweep or generational garbage collection, to efficiently manage memory. These algorithms determine which objects are reachable and in use by the program and which can be safely deallocated. The garbage collector runs periodically or when memory thresholds are reached to perform the necessary memory cleanup.

Question: How does Ruby manage memory through its garbage collector, and what are the main strategies used to efficiently reclaim memory?

5. **Explanation of various characteristics of Perl:** Perl is a high-level, general-purpose programming language known for its flexibility and strong text processing capabilities. Some key characteristics of Perl include:

- Expressive syntax: Perl emphasizes readability and conciseness, allowing developers to write expressive code using a rich set of built-in functions and operators.
- Regular expression support: Perl has robust support for regular expressions, making it a powerful language for string manipulation and pattern matching.
- CPAN ecosystem: Perl has a vast collection of reusable modules available through the Comprehensive Perl Archive Network (CPAN), providing extensive functionality for various domains.
- Platform independence: Perl is available on multiple platforms, including Unix/Linux, macOS, and Windows, making it highly portable.
- Rapid prototyping: Perl's quick development cycle and dynamic nature make it suitable for rapid prototyping and scripting tasks.

Question: Can you explain some of the key characteristics of Perl, including its syntax, regular expression support, CPAN

1. **Definition of canvas with an example:** In programming, a canvas is a graphical user interface (GUI) element used to draw and display graphics, images, and other visual elements. In web development, the canvas element in HTML5 provides a similar functionality for creating graphics and animations on web pages.

For example, in JavaScript, the canvas element can be created using the following code:

javascriptCopy code
```
var canvas = document.createElement('canvas'); canvas.width = 500; canvas.height = 500; document.body.appendChild(canvas);
```

This code creates a canvas element with a width and height of 500 pixels and appends it to the body of the HTML document. The canvas can then be used to draw graphics using various JavaScript functions.

Question: Can you define the canvas element in programming, and provide an example of how it can be used in web development?

2. **Usage of whitespaces in Ruby with an example:** In Ruby, whitespaces are used to separate and delimit statements, keywords, and expressions. For example, whitespace is used to separate variable names and values, method calls and arguments, and conditional statements and expressions.

For example, in Ruby, the following code uses whitespace to separate the `if` keyword and the conditional expression:

rubyCopy code
```ruby
if x > 10 puts "x is greater than 10" end
```

Here, the whitespace before the `if` keyword separates it from the conditional expression `x > 10`, making the code more readable and easier to parse.

Question: How are whitespaces used in Ruby to separate and delimit statements, expressions, and keywords, and can you provide an example of their usage?

3. **Definition of allocation function:** In programming, an allocation function is a function that is responsible for allocating memory for variables or data structures. In languages like C and C++, the `malloc()` and `new` functions are examples of allocation functions used to dynamically allocate memory at runtime.

In languages like Ruby and Python, allocation functions are handled by the language's runtime environment and are typically hidden from the programmer. In these languages, variables and objects are dynamically allocated and managed by the interpreter or virtual machine.

Question: Can you define an allocation function in programming, and provide examples of commonly used allocation functions in languages like C and C++?

4. **Explanation of static linking:** In programming, static linking is the process of linking libraries or object code directly into an executable file, creating a self-contained binary that can be executed on any system without requiring external dependencies.

Static linking is commonly used in languages like C and C++ to create standalone applications or libraries that can be distributed without worrying about runtime dependencies. However, static linking can result in larger executable files and may cause issues with version compatibility or security patches.

Question: What is static linking in programming, and how is it commonly used in languages like C and C++ to create standalone executables or libraries?

5. **Definition of Hashes in Perl and method to get the size of a hash:** In Perl, a hash is a data structure used to store key-value pairs, where each key is associated with a unique value. A hash can be created using the `%` symbol followed by a list of key-value pairs enclosed in curly braces `{}`.

For example, the following code creates a hash named `%colors` with three key-value pairs:

```
perl Copy code
%colors = ('red', '#FF0000', 'green', '#00FF00', 'blue', '#0000FF');
```

To get the size of a hash in Perl, the `keys()` function can be used to return an array of all keys in the hash context. The size of the hash can then be determined by using the `scalar()` function to

1. **Calling and identifying a subroutine in Perl:** In Perl, a subroutine is a named block of code that can be defined and called to perform a specific task. To call a subroutine, you simply use its name followed by parentheses `()`.

Example of calling a subroutine in Perl:

```
sub greet {
  print "Hello, World!\n";
}

# Calling the greet subroutine
greet();
```

To identify a subroutine, you can check if it has been defined using the `defined()` function. This function returns true if the subroutine has been defined and false otherwise.

Example of identifying a subroutine in Perl:

```
if (defined(&greet)) {
  print "The greet subroutine is defined.\n";
} else {
  print "The greet subroutine is not defined.\n";
}
```

Question: How can you call a subroutine in Perl, and what approach can be used to identify if a subroutine has been defined?

2. **File handle and syntax to create a file handle in Perl:** In Perl, a file handle is a named reference to a file or input/output stream. It allows you to read from or write to a file. File handles are typically represented by uppercase letters or underscores.

To create a file handle in Perl, you can use the `open()` function, which associates a file handle with a file or a stream.

Syntax to create a file handle in Perl:

```
open(my $file_handle, '<', 'filename.txt') or die "Cannot open file: $!";
```

In the above example, the `open()` function opens the file 'filename.txt' in read mode (`<`) and assigns it to the file handle `$file_handle`. The `or die` statement is used to handle any errors that occur while opening the file.

Question: What is a file handle in Perl, and can you provide the syntax to create a file handle using the `open()` function?

3. **Comparison between pack and unpack in Perl:** In Perl, `pack()` and `unpack()` are functions used for converting data between binary strings and formatted data structures.

- `pack()`: The `pack()` function is used to pack values into a binary string according to a specified template. It takes a template string and a list of values as arguments and returns a binary string.
- `unpack()`: The `unpack()` function is used to extract values from a binary string based on a specified template. It takes a template string and a binary string as arguments and returns a list of values.

While `pack()` converts values into a binary string, `unpack()` performs the reverse operation by extracting values from a binary string.

Question: What is the difference between the `pack()` and `unpack()` functions in Perl, and what are their respective purposes in data conversion?

4. **Security issues when using Tel on web applications:** When developing web applications using Perl, some common security issues to consider with the `tel://` URL scheme include:

- Tel URI injection: Attackers can manipulate the `tel://` URL to inject malicious phone numbers or other unexpected data, potentially causing unintended actions or vulnerabilities.
- Cross-Site Scripting (XSS): If user-supplied input is used directly in the `tel://` URL, it can be manipulated to inject malicious scripts, leading to XSS attacks.
- Privacy concerns: By including telephone numbers in the source code or exposing them through the `tel://` URL, there can be privacy implications for users if their phone numbers are accessible to unauthorized parties.

To mitigate these security issues, it is important to properly sanitize and validate user input, enforce input restrictions, and consider using appropriate encoding or escaping techniques to prevent injection attacks.

Question: What are some security issues to consider when using the