Day1 Assignment 1

Q1. What is the difference between statically typed and dynamically typed languages?

A- Statically typed languages

A language is statically typed if the type of a variable is known at compile time. For some languages this means that you as the programmer must specify what type each variable is; other languages (e.g.: Java, C, C++) offer some form of type inference, the capability of the type system to deduce the type of a variable (e.g.: OCaml, Haskell, Scala, Kotlin).

The main advantage here is that all kinds of checking can be done by the compiler, and therefore a lot of trivial bugs are caught at a very early stage.

Examples: C, C++, Java, Rust, Go, Scala

Dynamically typed languages

A language is dynamically typed if the type is associated with run-time values, and not named variables/fields/etc. This means that you as a programmer can write a little quicker because you do not have to specify types every time (unless using a statically-typed language with type inference).

Examples: Perl, Ruby, Python, PHP, JavaScript, Erlang

Q2. Difference between Scripting languages Vs Programming Languages.

A-Scripting languages are programming languages that don't require an explicit compilation step.

For example, in the normal case, you have to compile a C program before you can run it. But in the normal case, you don't have to compile a JavaScript program before you run it. So JavaScript is sometimes called a "scripting" language.

This line is getting more and more blurry since compilation can be so fast with modern hardware and modern compilation techniques. For instance, V8, the JavaScript engine in Google Chrome and used a lot outside of the browser as well, actually compiles the JavaScript code on the fly into machine code, rather than interpreting it. (In fact, V8's an optimizing two-phase compiler.)

Also note that whether a language is a "scripting" language or not can be more about the environment than the language. There's no reason you can't write a C interpreter and use it as a scripting language (and people have). There's also no reason you can't compile JavaScript to machine code and store that in an executable file (and people have). The language Ruby is a good example of this: The original implementation was entirely interpreted (a "scripting" language), but there are now multiple compilers for it.

Some examples of "scripting" languages (e.g., languages that are traditionally used without an explicit compilation step):

- Lua
- JavaScript
- VBScript and VBA
- Perl

And a small smattering of ones traditionally used with an explicit compilation step:

- C
- C++
- D
- Java (but note that Java is compiled to bytecode, which is then interpreted and/or recompiled at runtime)
- Pascal

Q3. Programming Paradigm

A- A **programming paradigm** is a style, or "way," of programming.

Some languages make it easy to write in some paradigms but not others.

Some Common Paradigms

- **Imperative**: Programming with an explicit sequence of commands that update state.
- **Declarative**: Programming by specifying the result you want, not how to get it.
- **Structured**: Programming with clean, goto-free, nested control structures.
- **Procedural**: Imperative programming with procedure calls.
- **Functional** (Applicative): Programming with function calls that avoid any global state.
- **Function-Level** (Combinator): Programming with no variables at all.
- **Object-Oriented**: Programming by defining objects that send messages to each other. Objects have their own internal (encapsulated) state and public interfaces. Object orientation can be:
    - **Class-based**: Objects get state and behavior based on membership in a class.
    - **Prototype-based**: Objects get behavior from a prototype object.
- **Event-Driven**: Programming with emitters and listeners of asynchronous actions.
- **Flow-Driven**: Programming processes communicating with each other over predefined channels.
- **Logic** (Rule-based): Programming by specifying a set of facts and rules. An engine infers the answers to questions.
- **Constraint**: Programming by specifying a set of constraints. An engine finds the values that meet the constraints.
- **Aspect-Oriented**: Programming cross-cutting concerns applied transparently.
- **Reflective**: Programming by manipulating the program elements themselves.

- **Array**: Programming with powerful array operators that usually make loops unnecessary.

Q4. Differentiate between HTTP 1.1 with HTTP/2 ?

A- HTTP stands for HYPERTEXT TRANSFER PROTOCOL & it is used in client-server communication. By using HTTP user sends the request to the server & the server sends the response to the user. There are several stages of development of HTTP but we will focus mainly on HTTP/1.1 which was created in 1997 & the new one is HTTP/2 which was created in 2015.

HTTP/1.1: For better understanding, before sending the request and the response there is a TCP connection established between client & server. Again you make a request to the server for image img.jpg & the server gives a response as an image img.jpg. The connection was not lost here after the first request because we add a keep-alive header which is the part of the request so there is an open connection between the server & client. There is a persistent connection which means several requests & responses are merged in a single connection. These are the drawbacks that lead to the creation of HTTP/2: The first problem is HTTP/1.1 transfer all the requests & responses in the plain text message form. The second one is head of line blocking in which TCP connection is blocked all other requests until the response does not receive. All the information related to the header file is repeated in every request.

HTTP/2: HTTP/2 was developed over the SPDY protocol. HTTP/2 works on the binary framing layer instead of textual that converts all the messages in binary format. It works on fully multiplexed that is one TCP connection is used for multiple requests. HTTP/2 uses HPACK which is used to split data from header. It compresses the header. The server sends all the other files like CSS & JS without the request of the client using the PUSH frame.

Difference between HTTP/1.1 and HTTP/2 are:

HTTP/1.1	HTTP/2
• Ithe usest works on the textual format.	• It works on the binary protocol.
• There is head of line blocking that blocks all the requests behind it until it doesn't get its all resources.	• It allows multiplexing so one TCP connection is required for multiple requests.
• It uses requests resource Inlining for use getting multiple pages	• It uses PUSH frame by server that collects all multiple pages
• It compresses data by itself.	• It uses HPACK for data compression.

Q5. Javascript Objects and its Internal Representation.

A- Javascript Objects and its Internal Representation.
Thumb rule:
=> Everything in JS is an object.
Object as in many other languages is a collection of properties and methods.
Example :

```
var myCar = {
make: 'Ford',
model: 'Mustang',
year: 1969
fullName : function () {
return this.make+ " " + this.model;
}
};
```

Objects are primarily collection of Key, value pairs.

Initialisation:

```
var myCar = news Object();
myCar.make = 'Ford';
myCar.model = 'Mustang';
myCar.year = '1969';
```

The above can also be initialised like,

```
var myCar = {
make: 'Ford',
model: 'Mustang',
year: 1969
};
```

Updating:
New properties and methods can be added to the existing objects.

```
myCar['name'] = 'Ford Mustang';
output:{
make: 'Ford',
model: 'Mustang',
year: 1969,
name: 'Ford Mustang'
};
```

Accessing JS Objects:
Using either dot notation or using square brackets. The dot notation is the most widely used.
e.g.,
myCar.name & myCar['name'] both gives 'Ford Mustang' as the ouput
Deleting:
'delete' operator is used for deleting a property of an objecting.
e.g.,
"delete myCar.name".
output:
{

make: ‘Ford’,
model: ‘Mustang’,
year: 1969
};

Internal Representation:
Javascript is a Prototype based language unlike C++ or java which are class based.
Javascript is also similar to those in that , this also uses constructor functions. But here any object can be used
as a prototype of any other object being created.
JS’s datatypes are broadly classified into two,
i) Primitive
ii) Composite and
iii) Special data types
Objects, arrays and functions fall under Composite data types
Fundamental difference b/w Primitive and Composite is that the composite types such as an Object is “Copied by Reference”.
in the below example,
var myCar = {
make: ‘Ford’,
model: ‘Mustang’,
year: 1969
};
the variable myCar doesn’t store the object itself by the address/reference.
example:
let user = { name: ‘John’ };
let admin = user;
admin.name = ‘Pete’; // changed by the “admin” reference
alert (user.name); // ‘Pete’, changes are seen from the “user” reference
Comparison by reference:
let a = {};
let b = a; // copy the reference
alert(a == b); // true, both variables reference the same object
alert(a === b); // true
But two different objects suh as the below ,
let a = {};
let b = {};
though both a & b are empty objects, if checked
alert(a==b), gives false, because both reference to different addresses.
Bottomline:
Objects are assigned and copied by reference. In other words, a variable stores not the “object value”, but a “reference” (address in memory) for the value. So copying such a variable or passing it as a function argument copies that reference, not the object.
All operations via copied references (like adding/removing properties) are performed on the same single object.