Interview Review

Different architectures
web hooks

Stack vs Heap

- Stack is a segment of memory where data is added and/or removed in a last-in-first-out (LIFO) manner.
- The memory size for the stack segment is predetermined by the operating system before compiling programs.
- Stack is faster than the heap but limited in size.
- Each thread will get its own stack.

  Stack stores:
  - Local variables
  - Methods in execution

  Heap stores:
  - Global and static variables
  - Variables which anonymous functions use
  - The results of async functions

[Stack vs Heap. What's the difference and why should I care?]

Compile-time vs Run-time

- Compile Time: The phase when the code is transformed into machine code or bytecode by the compiler. Syntax and semantic checks are performed, and compile-time errors are detected at this stage.
- Runtime: The phase when the compiled or interpreted program is executed. The program performs its tasks, and any errors or issues that occur during execution are called runtime errors.

Inline Function

An inline function is a concept in programming, particularly in languages like C++ and C, that suggests to the compiler that a certain function's code should be

inserted directly at the point where the function is called, rather than performing a typical function call. This is done to improve performance by reducing the overhead of function calls.

When a function is called, there is a certain amount of overhead involved in terms of pushing arguments onto the stack, creating a stack frame, and jumping to the function's code. For small and simple functions, this overhead can be significant compared to the actual work done by the function. Inlining helps avoid this overhead by inserting the function's code directly where it's called.

```
inline int Add(int a, int b) {
return a + b;
}
int main() {
int result = Add(5, 3); // The Add function's code is inserted here
return 0;
}
```

Friend Function

1. Access to Private Members: Friend functions can access private and protected members of a class, as if they were part of the class itself.
2. Not a Member Function: Friend functions are not actually member functions of the class they are friends with. They are defined outside the class, but they are granted special access.
3. Declaration and Definition: Friend functions must be declared inside the class that grants friendship and defined outside the class.
4. Keyword: The friend keyword is used to declare a function as a friend function within a class.
5. No Inheritance: Friendship is not inherited. A friend function of a class does not have access to the private members of derived classes.
6. No Implicit 'this': Friend functions do not have access to the this pointer, as they are not member functions. Instead, they receive objects as explicit parameters.
7. Scope: Friend functions are not bound to the scope of the class. They can be defined anywhere in the same namespace.

8. Purpose: Friend functions are useful for providing specific external functions access to private members, without exposing those members publicly.

```
class MyClass {
private:
int privateData;
public:
MyClass(int value) : privateData(value) {}
// Declare the friend function
friend void FriendFunction(const MyClass& obj);
};
// Define the friend function
void FriendFunction(const MyClass& obj) {
std::cout << "Friend function can access private data: " << obj.privateData << std::endl;
}
int main() {
MyClass obj(42);
FriendFunction(obj); // Call the friend function
return 0;
}
```

Operator Overloading

In languages like C++ and C#, operator overloading provides a way to make your classes more intuitive and expressive. Instead of relying solely on functions with names like add(), subtract(), and equals(), you can use familiar operators to perform those actions on your custom objects.

```
int main() {
Point p1(1.0, 2.0);
Point p2(3.0, 4.0);
Point result = p1 + p2; // Calls the overloaded + operator
return 0;
}
```

SDLC

Waterfall Model:

- Linear and sequential SDLC approach.

- Phases: Requirements, Design, Implementation, Testing, Deployment, Maintenance.
- Clear documentation at each phase.
- Well-suited for stable and fixed requirements.
- Limited flexibility for changes after phases.
- Project delays if errors found later.

Agile Model:

- Iterative and incremental approach.
- Emphasizes collaboration, customer feedback, working software, and adapting to change.
- Principles: Individuals over processes, working software over documentation, customer collaboration, responding to change.
- Methodologies: Scrum, Kanban, XP.
- Flexibility for changing requirements.
- Early and frequent delivery of functional software.
- Requires active customer involvement.
- May lack comprehensive documentation.

Why do we need OOP?

OOP helps us think in terms of real-world objects.
For example, a patient is an object and has properties like name, age, and the doctor that will be treating them.

Constructor and Destructor

The constructor is a special member function that is called automatically when an object is created. It initializes the object's data members and sets its initial state.
The destructor is another special member function that is called automatically when an object goes out of scope or is explicitly deleted. It is responsible for cleaning up resources and performing necessary cleanup tasks.

- Constructor Types in Java:
  1. Default Constructor:
     - Automatically provided if no constructors are defined.

    - Initializes attributes with default values.
    - No arguments.
2. Parameterized Constructor:
    - Accepts arguments to initialize object attributes.
    - Enables custom object initialization.
3. Copy Constructor:
    - Creates a new object by copying attributes from another object.
    - Can take an object of the same class or alternative parameters.
4. Chained Constructor:
    - Allows one constructor to call another within the same class.
    - Reduces code duplication and provides multiple initialization paths.
5. Private Constructor:
    - Restricts direct object instantiation from outside the class.
    - Used in singleton patterns or utility classes.

Ways to prevent a class from being instantiated:

1. Private Constructor:
   Defining a private constructor prevents the class from being instantiated directly from outside the class. This is one of the most common methods used to achieve this.
2. Abstract Class:
   Declaring a class as abstract means it can't be instantiated on its own. Subclasses that inherit from an abstract class can be instantiated, but the abstract class itself cannot.
3. Static Class (Utility Class):
   By defining all constructors as private and providing only static methods, you prevent instances of the class from being created. This type of class is often used to group related utility methods.

Polymorphism

- Static (compile-time)

  Static polymorphism is also known as compile-time polymorphism. It is achieved through function overloading and templates (in languages like C++). The method to

be executed is determined at compile-time, based on the number or types of arguments passed to the function.
Function overloading: In static polymorphism, you can have multiple functions with the same name but different parameter lists. The correct function to be called is resolved at compile-time based on the number or types of arguments passed.
Early error detection: Any issues related to calling the wrong function or passing the wrong arguments are detected at compile-time.

- Dynamic (run-time)
  Dynamic polymorphism is also known as run-time polymorphism. It is achieved through inheritance and virtual functions. The method to be executed is determined at run-time, based on the actual type of the object being referred to, rather than the reference or pointer type.
  In dynamic polymorphism, you have a base class and derived classes. The base class defines a virtual function, which can be overridden by the derived classes. When a method is called through a base class reference or pointer, the actual implementation of the function in the derived class is determined at run-time.
  In OOP, a virtual method is a function or method that is declared in a base class and can be overridden by derived classes. When a method is declared as "virtual," it allows the derived classes to provide their own implementation of the method, instead of using the implementation provided in the base class.

```
// Static Polymorphism.

public class Calculator {
// Method to add two integers
public int add(int a, int b) {
return a + b;
}
// Method to add three integers
public int add(int a, int b, int c) {
return a + b + c;
}
// Method to add two floating-point numbers
public double add(double a, double b) {
return a + b;
}
public static void main(String[] args) {
Calculator calculator = new Calculator();
int result1 = calculator.add(5, 10);
int result2 = calculator.add(2, 3, 5);
```

```
double result3 = calculator.add(3.14, 2.71);
System.out.println("Result 1: " + result1); // Output: Result 1: 15
System.out.println("Result 2: " + result2); // Output: Result 2: 10
System.out.println("Result 3: " + result3); // Output: Result 3: 5.85
}
}
```

```
// Dynamic Polymorphism.

class Shape {
public void draw() {
System.out.println("Drawing a Shape.");
}
}
class Circle extends Shape {
@Override
public void draw() {
System.out.println("Drawing a Circle.");
}
}
class Rectangle extends Shape {
@Override
public void draw() {
System.out.println("Drawing a Rectangle.");
}
}
public class DrawingApp {
public static void main(String[] args) {
Shape shape1 = new Circle();
Shape shape2 = new Rectangle();
shape1.draw(); // Output: Drawing a Circle.
shape2.draw(); // Output: Drawing a Rectangle.
}
}
```

Inheritance

1. Single Inheritance:
    - Class inherit from another class
2. Multiple Inheritance
    - Class inherit from multiple classes
3. Multilevel Inheritance:

- Involves a chain of classes where each derived class becomes the base class for the next one.
- Creates a hierarchical structure where each level inherits attributes and behaviors from its ancestor.
- Offers a clear parent-child relationship and allows gradual specialization.
- Example: Animal -> Mammal -> Human.
  Grand-Fa -> Father -> Son

1. Hierarchical Inheritance:
   - Involves multiple classes inheriting from a single base class.
   - Creates a tree-like structure where multiple derived classes branch out from a single root class.
   - Each derived class may add specific attributes and behaviors to the common base class.
   - Offers a way to group similar classes under a common parent.
   - Example: Shape -> Circle, Rectangle.
2. Hybrid Inheritance:
   - Combines multiple inheritance and multilevel inheritance.
   - Involves inheriting from multiple classes while also forming a hierarchy of inheritance.
   - Can lead to complex relationships and potential ambiguities.
   - Requires careful design to avoid issues like the "diamond problem."
   - Example: Fish (extends Animal, implements Swimmer) and Human (implements Swimmer).

Association

Association represents a relationship between two or more classes where objects of one class are connected to objects of another class. It's a general term that can cover various types of relationships, including one-to-one, one-to-many, and many-to-many.

Types Of Association:

- Aggregation

  Aggregation is a weaker form of association where a class (the whole) is associated with other classes (the parts), but the parts can exist

independently. The whole may or may not own the parts, and their lifecycles are not necessarily tied together.

```
class Department {
private String name;
// Other attributes and methods...
}
class Employee {
private String name;
private Department department;
public Employee(String name, Department department) {
this.name = name;
this.department = department;
}
// Other attributes and methods...
}
```

- Composition

  Composition is a strong form of association where a class (the whole) contains or is composed of other classes (the parts). The parts cannot exist independently of the whole, and when the whole is destroyed, its parts are also destroyed.

```
class Engine {
// Engine attributes and methods...
}
class Car {
private Engine engine;
public Car() {
this.engine = new Engine();
}
// Car attributes and methods...
}
```

REST API

1. Statelessness:
   - Each request is self-contained.

    - No client context is stored on the server.
    - Example: Creating a new blog post by providing all details in the request.
2. Client-Server Architecture:
    - Client handles user interface.
    - Server handles data storage and processing.
    - Example: Web browser displays blog posts, server manages data.
3. Uniform Interface:
    - HTTP methods and URLs have specific meanings.
    - Examples:
        - GET /api/posts: Retrieve all blog posts.
        - POST /api/posts: Create a new blog post.
        - PUT /api/posts/{post_id}: Update a blog post.
        - DELETE /api/posts/{post_id}: Delete a blog post.
4. Resource-Based:
    - Resources are entities like blog posts.
    - Identified by URLs.
    - Example: URL /api/posts/123 points to blog post with ID 123.
5. Cacheability:
    - Responses can be cached for performance.
    - Reduces repeated requests for unchanged data.
    - Example: Cache list of blog posts for a period.
6. Layered System:
    - Multiple layers with specific roles.
    - Enhances scalability and flexibility.
    - Example: Authentication, business logic, and database layers.
7. Code on Demand (Optional):
    - Server can send executable code to client.
    - Enhances client-side functionality.
    - Example: Sending JavaScript code for UI enhancements.