www.excellium.online

Author: Muhammad Jahanzaib

Interview Preparation & Essential Concepts for Object-Oriented Programming

Author: Muhammad Jahanzaib

Object Oriented Programming (OOP) - Topics

- Variables and data types
- Control structures (if/else, loops)
- Classes and objects
- Encapsulation
- Inheritance
- Shallow copy *Vs* Deep copy
- Pointers
- Virtual table
- Polymorphism
- Interfaces and abstract classes
- Design patterns
- Object-oriented analysis and design (OOAD)

Author: Muhammad Jahanzaib

Object Oriented Programming (OOP)

Definition:

OOP, or Object-Oriented Programming, is a programming paradigm that is based on the concept of objects, which can contain data and methods that operate on that data. OOP is designed to make software development more modular, flexible, and reusable by providing a structured way to organize code and encapsulate data.

There are four pillars of OOP:

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Abstraction

Author: Muhammad Jahanzaib

Object Oriented Programming (OOP)

Some of the benefits of OOP include:

Modularity:

OOP allows developers to break down complex systems into smaller, more manageable components called objects, which can be developed and tested independently of each other.

Encapsulation:

OOP allows data to be hidden and protected from unauthorized access, reducing the risk of bugs and making the code more secure.

Reusability:

Objects can be reused in different contexts, reducing the need for redundant code and improving the overall efficiency of the software development process.

Flexibility:

OOP allows for easy modification and extension of existing code, reducing the risk of errors and making it easier to adapt to changing requirements.

Author: Muhammad Jahanzaib

Variables

Definition:

A variable is a container for storing data values in computer memory that can be accessed and manipulated by a program. Variables are assigned a data type that determines the type of data they can hold, such as integers, floating-point numbers, or characters.

Key points:

- Variables have a name, data type, and value.
- The value of a variable can be changed during program execution.
- Variables can be used in expressions and statements to perform operations.

```
// integer variable (size: 4 bytes)
int age = 25;

// double-precision floating-point variable (size: 8 bytes)
double pi = 3.14159;

// character variable (size: 1 byte)
char letter = 'A';

// boolean variable (size: 1 byte)
bool is_true = true;

// string variable (size varies)
string name = "John Doe";

// integer array (size varies)
int arr[5] = {1, 2, 3, 4, 5};

// pointer variable (size: 4 bytes on 32-bit systems, 8 bytes on 64-bit systems)
int* ptr = nullptr;

// structure variable (size varies)
struct Person {
    string name;
    int age;
};
Person john = {"John Doe", 25};

// enumeration variable (size: 4 bytes)
enum Color { Red, Green, Blue };
Color favorite_color = Green;

// void variable (size: N/A)
void* data = nullptr;
```

Author: Muhammad Jahanzaib

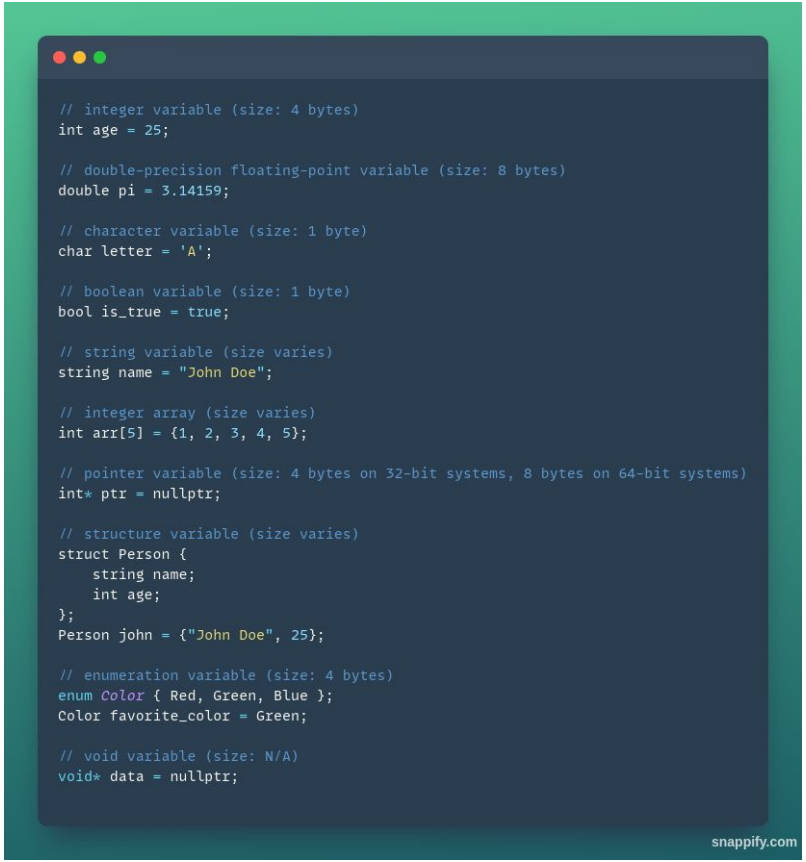
Data types

Definition:

A data type is a classification of data that specifies the type of value a variable can hold. Common data types include integers, floating-point numbers, characters, and booleans.

Key points:

- Data types determine the size and format of data stored in a variable..
- Different programming languages support different data types.
- Data types can be used to perform type checking and casting operations.



Control structures (if/else)

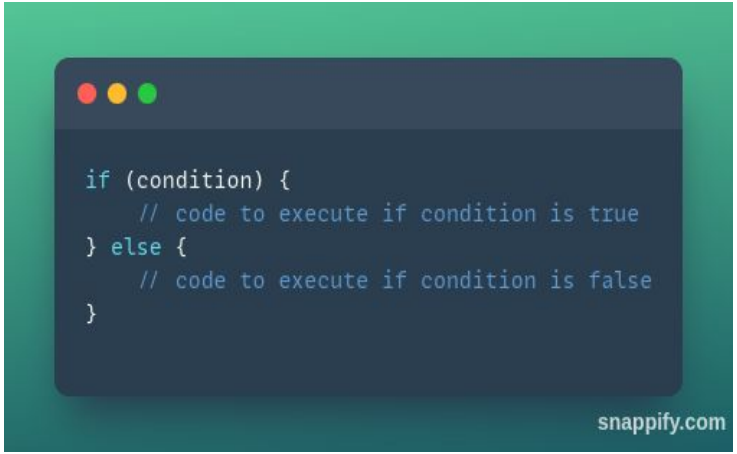
Definition:

Control structures are programming constructs that allow developers to control the flow of execution in a program.

If/else statements:

- If/else statements are used to execute certain code blocks based on a particular condition.
- Here, condition is a Boolean expression that evaluates to either true or false. If the condition is true, the code inside the first block is executed, and if the condition is false, the code inside the second block is executed.

The basic syntax of an if/else statement



Control structures (loops)

Definition:

Loops are used to execute a block of code repeatedly until a certain condition is met. There are three types of loops in most programming languages: for, while, and do-while loops.

loops:

- In a for loop, the loop variable is initialized, a condition is specified, and an increment or decrement is performed after each iteration of the loop.
- In a while loop, the condition is tested at the beginning of each iteration of the loop.
- In a do-while loop, the condition is tested at the end of each iteration of the loop, so the loop is guaranteed to execute at least once.

The basic syntax of an loop statement

Author: Muhammad Jahanzaib

Control structures - Programming Questions

1. Write a program that prints the numbers from 1 to 100. For multiples of three, print "Fizz" instead of the number and for multiples of five, print "Buzz". For numbers which are multiples of both three and five, print "FizzBuzz".
2. Write a program that prompts the user to enter a positive integer and then prints all the factors of that integer.
3. Write a program that takes two integers as input and then finds the greatest common divisor of the two integers using the Euclidean algorithm.
4. Write a program that prompts the user to enter a sequence of numbers terminated by -1 and then calculates the sum and average of the sequence.
5. Write a program that generates a random integer between 1 and 100 and asks the user to guess the number. The program should keep prompting the user to guess until they correctly guess the number.
6. Write a program that reads in a list of integers and then prints out the largest and smallest integers in the list.
7. Write a program that prompts the user to enter a sequence of numbers and then sorts the numbers in ascending order.
8. Write a program that prompts the user to enter a sequence of strings and then sorts the strings in alphabetical order.
9. Write a program that reads in a list of numbers and then removes any duplicates from the list.
10. Write a program that prompts the user to enter a sentence and then prints out the number of vowels in the sentence.

Author: Muhammad Jahanzaib

Classes & Objects

Definition:
A **class** is a blueprint or template for creating objects that define a set of attributes (data) and methods (functions) that are common to all instances of that class.
An **object** is an instance of a class that contains specific values for the attributes defined in the class.

Key points:

- Classes are used to define the structure of objects and their behaviors.
- Objects are instances of a class that contain specific values for the attributes defined in the class.
- Classes can be used to group related functions and data into a single entity.

Author: Muhammad Jahanzaib

Classes & Objects

Real world example:

A good example of a class and object is a car. The car can be defined as a class that contains attributes such as make, model, color, and year, and methods such as start, stop, and accelerate. An object of the car class would be an instance of the class, with specific values for the attributes (e.g., a red 2010 Honda Civic).

Author: Muhammad Jahanzaib

Classes & Objects

Code Examples:

C++

```cpp
class Car {
private:
    string make;
    string model;
    string color;
    int year;
    int speed;

public:
    void start() {
        // code to start the engine
    }

    void stop() {
        // code to stop the engine
    }

    void accelerate(int newSpeed) {
        this->speed = newSpeed;
    }

    int getSpeed() {
        return this->speed;
    }
};

int main() {
    Car myCar; // create an object of the Car class
    myCar.start(); // call the start() method
    myCar.accelerate(50); // call the accelerate() method
    cout << "Current speed: " << myCar.getSpeed() << endl;
    // print the current speed
    return 0;
}
```


JavaScript

```javascript
class Car {
    constructor(make, model, color, year) {
        this.make = make;
        this.model = model;
        this.color = color;
        this.year = year;
        this.speed = 0;
    }

    start() {
        // code to start the engine
    }

    stop() {
        // code to stop the engine
    }

    accelerate(newSpeed) {
        this.speed = newSpeed;
    }

    getSpeed() {
        return this.speed;
    }
}

let myCar = new Car("Honda", "Civic", "Red", 2010);
// create an object of the Car class
myCar.start();
// call the start() method
myCar.accelerate(50);
// call the accelerate() method
console.log("Current speed:", myCar.getSpeed());
// print the current speed
```

Author: Muhammad Jahanzaib

Constructor

Definition:

A constructor is a special member function that is called when an object of a class is created. It has the same name as the class, and no return type. It is used to initialize the data members of the object with default or user-defined values

Key points:

- Constructors are called automatically when an object is created.
- They have the same name as the class and no return type.
- They can be overloaded with different parameters.
- If no constructor is defined, the compiler will provide a default constructor.

Author: Muhammad Jahanzaib

Constructor

Code Example:

C++

```
class Rectangle {
private:
    int length;
    int width;
public:
    Rectangle() { //default constructor
        length = 0;
        width = 0;
    }
    Rectangle(int l, int w) { //parameterized constructor
        length = l;
        width = w;
    }
};

int main() {
    Rectangle r1; //default constructor called
    Rectangle r2(5, 10); //parameterized constructor called
    return 0;
}
```

Author: Muhammad Jahanzaib

Destructor

Definition:

A destructor is a special member function that is called when an object is destroyed. It is used to free up any resources allocated by the object during its lifetime.

Key points:

- Destructors are called automatically when an object is destroyed.
- They have the same name as the class, preceded by a tilde (~), and no parameters.
- They cannot be overloaded.
- If no destructor is defined, the compiler will provide a default destructor.

Author: Muhammad Jahanzaib

C++

Destructor

Code Example:

Author: Muhammad Jahanzaib

Classes - Interview Questions

1. What is the difference between a class and an object in C++?
2. What is a constructor in C++ and why is it important?
3. Can you explain the concept of object initialization in C++?
4. What is a destructor in C++ and why is it important?
5. What is the difference between a shallow copy and a deep copy in C++?
6. What is an access specifier in C++ and how does it impact class members?
7. How do you prevent copying of a C++ object?
8. What is an abstract class in C++ and how is it different from a regular class?
9. How do you declare a member function of a class as virtual in C++?
10. How can you determine the size of an object of a class in C++?

Author: Muhammad Jahanzaib

Encapsulation

Definition:
Encapsulation is a fundamental principle of object-oriented programming that involves bundling data and methods that operate on that data within a single unit or class, and restricting access to that data to only the methods that are part of that class. Encapsulation provides the ability to hide implementation details and protect data from outside interference or modification.

Key points:

- Encapsulation involves bundling data and methods within a class and restricting access to that data to only the methods that are part of that class.
- Encapsulation provides the ability to hide implementation details and protect data from outside interference or modification.
- Encapsulation improves the modularity, maintainability, and scalability of code.

Author: Muhammad Jahanzaib

Encapsulation

Real world example:

A real-world example of encapsulation is a bank account. The bank account can be defined as a class that contains attributes such as account number, balance, and owner, and methods such as deposit, withdraw, and transfer. The balance attribute should be protected and only accessible through the methods provided by the bank account class to prevent unauthorized modification of the balance.

Author: Muhammad Jahanzaib

Encapsulation

Code Examples:

C++

```cpp
class BankAccount {
private:
    string accountNumber;
    double balance;
    string owner;

public:
    void deposit(double amount) {
        this->balance += amount;
    }

    void withdraw(double amount) {
        this->balance -= amount;
    }

    void transfer(double amount, BankAccount& toAccount) {
        this->balance -= amount;
        toAccount.deposit(amount);
    }

    double getBalance() const {
        return this->balance;
    }
};

int main() {
    BankAccount myAccount;
    myAccount.deposit(1000);
    myAccount.withdraw(500);
    BankAccount yourAccount;
    myAccount.transfer(200, yourAccount);
    cout << "My balance: " << myAccount.getBalance() << endl;
    cout << "Your balance: " << yourAccount.getBalance() << endl;
    return 0;
}
```


JavaScript

```javascript
class BankAccount {
    constructor(accountNumber, balance, owner) {
        this.accountNumber = accountNumber;
        this.balance = balance;
        this.owner = owner;
    }

    deposit(amount) {
        this.balance += amount;
    }

    withdraw(amount) {
        this.balance -= amount;
    }

    transfer(amount, toAccount) {
        this.balance -= amount;
        toAccount.deposit(amount);
    }

    getBalance() {
        return this.balance;
    }
}

let myAccount = new BankAccount("123456", 1000, "John");
myAccount.deposit(500);
let yourAccount = new BankAccount("654321", 2000, "Jane");
myAccount.transfer(200, yourAccount);
console.log("My balance: " + myAccount.getBalance());
console.log("Your balance: " + yourAccount.getBalance());
```

Author: Muhammad Jahanzaib

Encapsulation - Interview Questions

1. What is encapsulation in object-oriented programming?
2. Why is encapsulation important in software development?
3. How do you achieve encapsulation in C++?
4. What is the difference between private, public, and protected access specifiers in C++?
5. Can you give an example of encapsulation in real-world scenarios?
6. How does encapsulation help in achieving data abstraction?
7. What is the role of constructors and destructors in encapsulation?
8. What is data hiding in encapsulation?
9. How do you implement encapsulation in Java?
10. What are the benefits of encapsulation in software development?

Author: Muhammad Jahanzaib

Inheritance

Definition:

Inheritance is a mechanism that allows a new class to be based on an existing class. The new class inherits the properties and behavior of the existing class, and can also add new properties and behavior or override the existing ones. The existing class is called the base or parent class, and the new class is called the derived or child class.

Key points:

- Inheritance is a mechanism of reusing code and reducing duplication.
- Inheritance creates a hierarchical relationship between classes.
- A child class can access the properties and methods of its parent class.
- A child class can add new properties and methods or override the existing ones.
- Inheritance can be of different types: single, multiple, multilevel, and hierarchical.

Author: Muhammad Jahanzaib

Inheritance

Real world example:

A real-world example of inheritance can be a vehicle class that has a base set of properties and methods, and then specific vehicle types, such as cars, trucks, and motorcycles, can be created by inheriting from the vehicle class and adding their own specific properties and methods.

Author: Muhammad Jahanzaib

Inheritance

Code Examples:

C++

```cpp
class Shape {
protected:
    double width;
    double height;

public:
    Shape(double w, double h) : width(w), height(h) {}
    double getArea() const { return 0.0; }
};

class Rectangle : public Shape {
public:
    Rectangle(double w, double h) : Shape(w, h) {}
    double getArea() const override { return width * height; }
};

class Triangle : public Shape {
public:
    Triangle(double w, double h) : Shape(w, h) {}
    double getArea() const override { return width * height / 2; }
};

int main() {
    Rectangle rect(10, 5);
    Triangle tri(10, 5);
    cout << "Rectangle area: " << rect.getArea() << endl;
    cout << "Triangle area: " << tri.getArea() << endl;
    return 0;
}
```


JavaScript

```javascript
class Shape {
    constructor(width, height) {
        this.width = width;
        this.height = height;
    }

    getArea() {
        return 0.0;
    }
}

class Rectangle extends Shape {
    constructor(width, height) {
        super(width, height);
    }

    getArea() {
        return this.width * this.height;
    }
}

class Triangle extends Shape {
    constructor(width, height) {
        super(width, height);
    }

    getArea() {
        return this.width * this.height / 2;
    }
}

let rect = new Rectangle(10, 5);
let tri = new Triangle(10, 5);
console.log("Rectangle area: " + rect.getArea());
console.log("Triangle area: " + tri.getArea());
```

Author: Muhammad Jahanzaib

Inheritance - Types of Inheritance

There are several types of inheritance in OOP. Some of the most common types include:

Types:

1. Single inheritance
2. Multiple inheritance:
3. Hierarchical inheritance
4. Multilevel inheritance
5. Hybrid inheritance

Each type of inheritance has its own advantages and disadvantages, and the choice of inheritance type depends on the specific requirements of the program being developed.

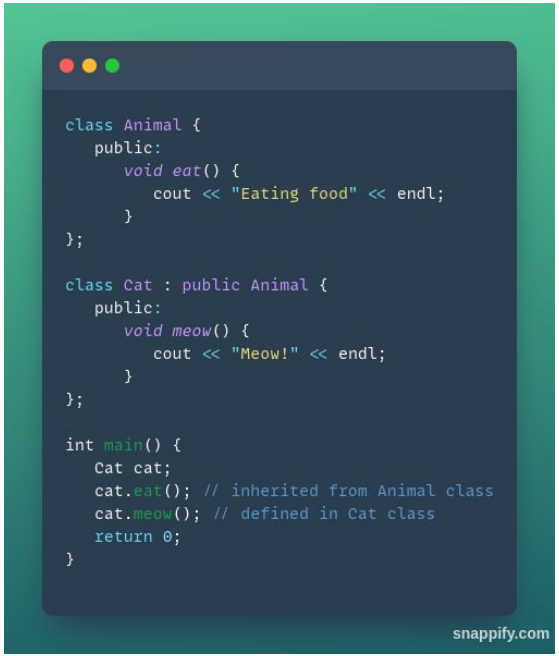
Inheritance - Types of Inheritance - Single inheritance

Definition:

In single inheritance, a subclass or derived class inherits from a single base class or parent class.

Real world example:

A cat is an animal. It inherits characteristics from the animal class, such as the ability to eat and move, but also has its own unique characteristics, such as meowing.



Author: Muhammad Jahanzaib

Inheritance - Types of Inheritance - Multiple inheritance

Definition:

In multiple inheritance, a subclass or derived class inherits from multiple base classes or parent classes.

Real world example:

A drone that has both hovering capabilities like a bird and can also chase after objects like a dog.

Inheritance - Types of Inheritance - Hierarchical inheritance

Definition:

In hierarchical inheritance, a subclass or derived class inherits from a single base class or parent class, but multiple subclasses can inherit from the same base class.

Real world example:

A pet store with both cats and dogs. Both animals share some common traits like eating, but also have their own unique traits like meowing and barking.

Author: Muhammad Jahanzaib

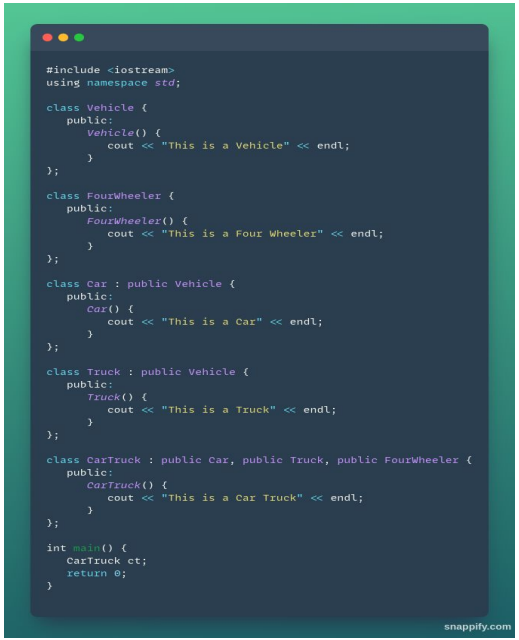
Inheritance - Types of Inheritance - Hybrid inheritance

Definition:

Hybrid inheritance combines two or more types of inheritance, such as single and multiple inheritance, to achieve a specific programming goal.

Real world example:

A real-world example of hybrid inheritance might be a vehicle rental company's class hierarchy. The Vehicle class could contain general information about the vehicle, such as the make and model, while the FourWheeler class could contain information specific to four-wheeled vehicles. The Car and Truck classes could inherit from Vehicle to get access to the general information, and then CarTruck could inherit from both of them to create a class for rental vehicles that are both cars and trucks.



Author: Muhammad Jahanzaib

Inheritance - Diamond Problem

The diamond problem is a common issue in inheritance where a subclass extends from two separate superclasses, which themselves share a common superclass. This creates ambiguity in the subclass, as it is unclear which superclass's method should be used.

In this example, **Platypus** inherits from both **Mammal** and **Bird**, which themselves inherit from **Animal**. However, since **Platypus** inherits from both **Mammal** and **Bird**, it now has two copies of the **move()** method from **Animal**. This ambiguity can cause issues when calling the **move()** method on **Platypus**.

Author: Muhammad Jahanzaib

Inheritance - Virtual Functions

Definition:

A virtual function is a member function of a class that can be overridden in a subclass. It allows the subclass to provide its own implementation of the method, which is called at runtime instead of the implementation provided by the base class.

Key points:

- A virtual function is marked with the virtual keyword in the base class, and can be overridden in the subclass using the override keyword.
- When a virtual function is called on an object, the actual implementation used depends on the type of the object at runtime, rather than the type of the reference to the object.
- Virtual functions allow for polymorphism, where a single interface can be used to represent multiple types of objects.

Author: Muhammad Jahanzaib

Inheritance - Virtual Functions

In this example, **Shape** is a base class with a virtual **area()** function that is overridden in the **Rectangle** and **Circle** subclasses. An array of **Shape** objects is created, and the **area()** function is called on each object. Since **area()** is a virtual function, the appropriate implementation is called for each object at runtime.

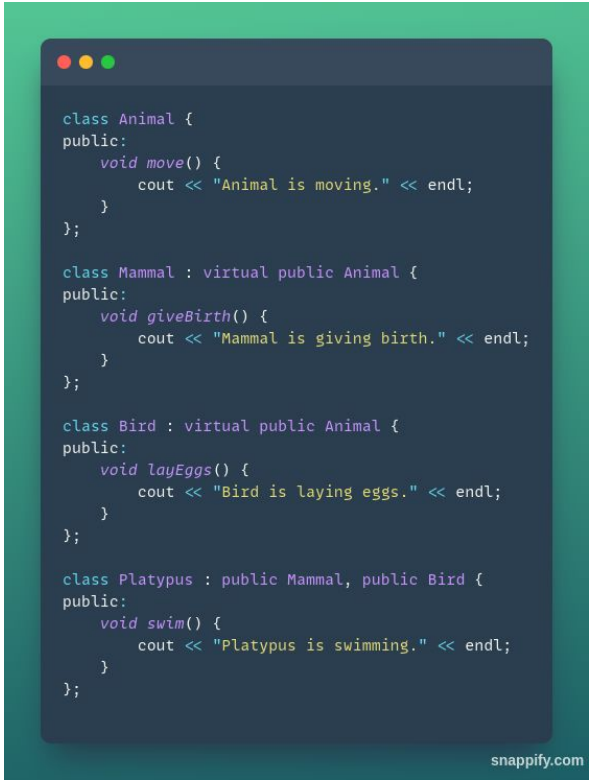
```cpp
class Shape {
public:
    virtual float area() { return 0; }
};

class Rectangle : public Shape {
public:
    float area() override { return length * width; }
private:
    float length;
    float width;
};

class Circle : public Shape {
public:
    float area() override { return 3.14 * radius * radius; }
private:
    float radius;
};

int main() {
    Shape* shapes[] = {new Rectangle(), new Circle()};
    for (Shape* shape : shapes) {
        std::cout << shape->area() << std::endl;
    }
    return 0;
}
```

Author: Muhammad Jahanzaib

Inheritance - Diamond Problem

One solution to the diamond problem is to use virtual inheritance. With virtual inheritance, only one copy of the shared base class is created, and all other subclasses inherit from that one copy.

With virtual inheritance, **Platypus** now only has one copy of the **move()** method from **Animal**, and the ambiguity is resolved.



Inheritance - Dynamic Memory Allocation:

Definition:

Dynamic memory allocation is the process of allocating memory at runtime, rather than at compile time. It allows programs to allocate memory as needed, rather than having to reserve memory in advance.

Key points:

- In C++, dynamic memory allocation is performed using the new operator to allocate memory on the heap
- The new operator returns a pointer to the allocated memory, which can be stored in a pointer variable.
- Memory allocated using new must be explicitly deallocated using the delete operator to avoid memory leaks.
- Improper use of dynamic memory allocation can lead to memory leaks, segmentation faults, and other issues.

Author: Muhammad Jahanzaib

Inheritance - Virtual Functions

Definition:

A virtual function is a member function of a class that can be overridden in a subclass. It allows the subclass to provide its own implementation of the method, which is called at runtime instead of the implementation provided by the base class.

Key points:

- A virtual function is marked with the virtual keyword in the base class, and can be overridden in the subclass using the override keyword.
- When a virtual function is called on an object, the actual implementation used depends on the type of the object at runtime, rather than the type of the reference to the object.
- Virtual functions allow for polymorphism, where a single interface can be used to represent multiple types of objects.

Author: Muhammad Jahanzaib

Inheritance - Interview Questions

1. What is inheritance in OOP and why is it important?
2. What is the difference between single, multiple, and multilevel inheritance? Can you give examples of each?
3. What is the diamond problem in multiple inheritance and how do you solve it?
4. What is the purpose of virtual functions in C++? How do they relate to inheritance?
5. How do you prevent inheritance in C++? When might you want to do this?
6. What is the difference between public, protected, and private inheritance? When might you use each?
7. What is the Liskov substitution principle and how does it relate to inheritance?
8. How can you use inheritance to make your code more modular and reusable?
9. What is the difference between abstract classes and interfaces in OOP? When might you use each?
10. Can you give an example of a real-world scenario where inheritance would be useful in software design?

Author: Muhammad Jahanzaib

Shallow copy *Vs* Deep copy

Shallow copy

- Creates a new object that points to the same memory addresses as the original object
- Only copies the top-level values of the original object, not the values of nested objects
- Changes made to the copy will affect the original object
- Uses less memory than deep copy
- May result in unexpected behavior if not used carefully

Deep copy

- Creates a new object that has its own memory addresses, independent of the original object
- Copies all the values of the original object, including nested objects
- Changes made to the copy will not affect the original object
- Uses more memory than shallow copy
- Ensures that the original object is not affected by changes made to the copy

Author: Muhammad Jahanzaib

Shallow copy *Vs* Deep copy

Code example:

```cpp
#include <iostream>

class Person {
public:
    Person(std::string n, int a) : name(n), age(a) {}

    // Shallow copy constructor
    Person(const Person& other) : name(other.name), age(other.age) {}

    // Deep copy constructor
    Person(const Person& other) : name(other.name), age(other.age) {
        if (other.address != nullptr) {
            address = new std::string(*other.address);
        }
    }

    // Destructor
    ~Person() {
        delete address;
    }

    std::string getName() const {
        return name;
    }

    int getAge() const {
        return age;
    }

    std::string getAddress() const {
        return *address;
    }

    void setAddress(std::string a) {
        *address = a;
    }

private:
    std::string name;
    int age;
    std::string* address = nullptr;
};

int main() {
    // Shallow copy
    Person p1("Alice", 25);
    Person p2 = p1;
    std::cout << "Shallow copy:\n";
    std::cout << "p1 name: " << p1.getName() << ", age: " << p1.getAge() << std::endl;
    std::cout << "p2 name: " << p2.getName() << ", age: " << p2.getAge() << std::endl;
    p2.setName("Bob");
    std::cout << "After changing p2 name:\n";
    std::cout << "p1 name: " << p1.getName() << ", age: " << p1.getAge() << std::endl;
    std::cout << "p2 name: " << p2.getName() << ", age: " << p2.getAge() << std::endl;

    // Deep copy
    Person p3("Charlie", 30);
    p3.setAddress("123 Main St");
    Person p4 = p3;
    std::cout << "Deep copy:\n";
    std::cout << "p3 name: " << p3.getName() << ", age: " << p3.getAge() << ", address: " << p3.getAddress() << std::endl;
    std::cout << "p4 name: " << p4.getName() << ", age: " << p4.getAge() << ", address: " << p4.getAddress() << std::endl;
```

Author: Muhammad Jahanzaib

Shallow copy *Vs* Deep copy - Interview Questions

1. What is the difference between shallow copy and deep copy in C++?
2. How do you implement shallow copy in a class?
3. Can you give an example of a situation where shallow copy can cause issues?
4. What are some ways to implement deep copy in C++?
5. Can you explain the copy constructor and its role in deep copy?

Author: Muhammad Jahanzaib

Pointers

Definition:

A pointer is a variable that stores the memory address of another variable. Pointers allow you to manipulate data indirectly by referring to the memory location rather than the data itself.

Key points:

- Pointers are declared using the * symbol before the variable name.
- The & symbol can be used to get the memory address of a variable.
- Pointers can be used to pass parameters to functions by reference.
- Pointers can be used to dynamically allocate memory using the new operator.
- Pointer arithmetic can be used to navigate through memory.

Author: Muhammad Jahanzaib

Pointers

Real world example:

A GPS system that stores the memory addresses of various locations and uses pointers to navigate between them.

C++

```
int main() {
    int num = 10;
    int* ptr = &num;
    // ptr now points to the memory address of num

    cout << *ptr << endl;
    // prints the value stored at the memory address pointed to by ptr

    int* arr = new int[5];
    // dynamically allocate memory for an array of 5 integers
    arr[0] = 1;
    // store 1 in the first element of the array
    *(arr+1) = 2;
    // use pointer arithmetic to store 2 in the second element of the array

    delete[] arr;
    // free the memory allocated for the array
    return 0;
}
```

Author: Muhammad Jahanzaib

Pointers - Interview Questions

1. What is a pointer in C++? How is it different from a regular variable?
2. What is a null pointer? Why would you use one?
3. What is the difference between a pointer and a reference in C++?
4. What is a dangling pointer? How can you avoid creating one?
5. Can a pointer point to itself? If so, why might you want to do that?
6. What is the difference between a stack-based and heap-based pointer?
7. What is a smart pointer in C++? How does it work?
8. How can you use pointers to pass parameters to functions by reference?
9. How can you use pointers to dynamically allocate memory in C++?
10. What are some common mistakes that programmers make when using pointers? How can you avoid them?

Author: Muhammad Jahanzaib

Virtual table

Definition:

Virtual table, also known as a v-table or a virtual function table, is a mechanism used in object-oriented programming (OOP) to support dynamic dispatch or late binding of polymorphic functions. It is a table of function pointers maintained by the compiler for each class that has one or more virtual functions.

Key points:

- A virtual table is a mechanism used to support dynamic dispatch or late binding of polymorphic functions in OOP.
- It is a table of function pointers that is maintained by the compiler for each class that has one or more virtual functions.
- Function overloading allows multiple functions with the same name but different parameters to be defined in a class.
- When a class inherits from a base class that has virtual functions, it inherits the base class's virtual table and adds its own virtual functions to it.
- The virtual table is used at runtime to determine which function to call when a virtual function is invoked through a pointer or reference to a base class.

Author: Muhammad Jahanzaib

Virtual table

Code Explanation:

In the above example, the **Animal** class has a virtual function **makeSound()**. When the **Dog** class inherits from the **Animal** class, it adds its own implementation of the **makeSound()** function to the virtual table. At runtime, when we call **makeSound()** through a pointer to an **Animal** object, the virtual table is used to determine which implementation of the function to call, based on the actual type of the object (in this case, **Dog**).

Author: Muhammad Jahanzaib

Virtual table - Interview Questions

1. What is a virtual table in C++?
2. How is a virtual table created and used in C++?
3. Can you explain how virtual tables enable polymorphism in C++?
4. How does the concept of virtual table differ from virtual functions in C++?
5. Can you give an example of how virtual tables are used in a real-world application?

Author: Muhammad Jahanzaib

Polymorphism

Definition:

Polymorphism is the ability of objects of different classes to be treated as if they were objects of a common parent class. It allows methods to be written to handle objects of a general type, and then specific implementations can be created in derived classes to handle their unique behavior.

Key points:

- Polymorphism allows a single interface to be used to represent different types of objects
- Polymorphism can be achieved through function overloading and function overriding
- Function overloading allows multiple functions with the same name but different parameters to be defined in a class.
- Function overriding allows a derived class to provide its own implementation of a method defined in the base class.
- Polymorphism helps to write more flexible and reusable code.

Author: Muhammad Jahanzaib

Polymorphism

Real world example:

A real-world example of polymorphism can be a media player that can play different types of media, such as videos, music, and podcasts. Each type of media has its own specific behavior, but they can all be played using a common **"play"** method.

Author: Muhammad Jahanzaib

Polymorphism

Code Examples:

C++

```cpp
class Shape {
public:
    virtual double getArea() const { return 0.0; }
};

class Rectangle : public Shape {
protected:
    double width;
    double height;

public:
    Rectangle(double w, double h) : width(w), height(h) {}
    double getArea() const override { return width * height; }
};

class Triangle : public Shape {
protected:
    double width;
    double height;

public:
    Triangle(double w, double h) : width(w), height(h) {}
    double getArea() const override { return width * height / 2; }
};

void printArea(const Shape& shape) {
    cout << "Area: " << shape.getArea() << endl;
}

int main() {
    Rectangle rect(10, 5);
    Triangle tri(10, 5);
    printArea(rect);
    printArea(tri);
    return 0;
}
```


JavaScript

```javascript
class Shape {
    getArea() {
        return 0.0;
    }
}

class Rectangle extends Shape {
    constructor(width, height) {
        super();
        this.width = width;
        this.height = height;
    }

    getArea() {
        return this.width * this.height;
    }
}

class Triangle extends Shape {
    constructor(width, height) {
        super();
        this.width = width;
        this.height = height;
    }

    getArea() {
        return this.width * this.height / 2;
    }
}

function printArea(shape) {
    console.log("Area: " + shape.getArea());
}

let rect = new Rectangle(10, 5);
let tri = new Triangle(10, 5);
printArea(rect);
printArea(tri);
```

Author: Muhammad Jahanzaib

Function overriding

Definition:

Function overriding is a feature in object-oriented programming where a child class provides a different implementation of a method that is already defined in its parent class. The overridden method in the child class has the same signature (name, return type, and parameters) as the method in the parent class.

Key points:

- Function overriding is used in inheritance to provide a specific implementation of a method in a child class.
- The overridden method in the child class must have the same signature as the method in the parent class.
- The overridden method must have the same return type or a covariant return type, which means that it can return a subclass of the return type in the parent method.
- Function overriding is used to implement polymorphism.

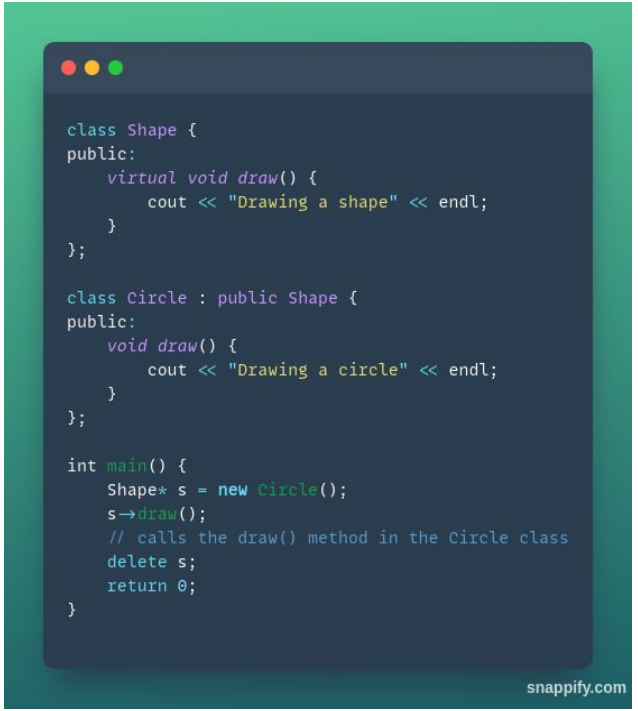
Author: Muhammad Jahanzaib

C++

Function overriding

Code Example:

In this example, we have a base class **Animal** and two derived classes **Cat** and **Dog**. Each class has a **makeSound** method, but the implementation in each subclass overrides the implementation in the base class. We create two objects, one of type **Cat** and one of type **Dog**, and call the **makeSound** method on each object. The output shows that the method implementation from the respective subclass is called.



Author: Muhammad Jahanzaib

Polymorphism - Interview Questions

1. What is Polymorphism? Can you explain with an example?
2. What are the types of Polymorphism in OOP?
3. Can you explain the difference between Compile-time Polymorphism and Runtime Polymorphism?
4. What is function overloading? Can you provide an example?
5. What is function overriding? Can you provide an example?
6. How does Polymorphism help in achieving abstraction in OOP?
7. What is the difference between Abstract Class and Interface? How is Polymorphism related to them?
8. Can you explain the concept of Virtual functions and how they are related to Polymorphism?
9. How can we achieve Polymorphism using pointers and references in C++?
10. Can you explain the importance of Polymorphism in Object-Oriented Programming?

Author: Muhammad Jahanzaib

Function overloading

Definition:

Function overloading is a feature of object-oriented programming that allows creating multiple functions with the same name but with different parameters or argument types.

Key points:

- Function overloading allows a programmer to write several functions with the same name but with different parameters.
- The compiler identifies the function call based on the number and types of arguments passed
- The return type of the function does not play any role in function overloading.

Author: Muhammad Jahanzaib

Function overloading

Code Examples:

In the above examples, we have defined two functions with the same name **"add"**, but with different parameter types. When the function is called with different arguments, the compiler identifies the correct function to call based on the argument types.
Function overloading is useful when we want to perform similar operations on different types of data. It improves code readability and reduces code duplication.

C++

```cpp
#include <iostream>
using namespace std;

int add(int a, int b) {
    return a + b;
}

double add(double a, double b) {
    return a + b;
}

int main() {
    cout << add(5, 10) << endl;
    // calls int add(int a, int b)
    cout << add(3.5, 2.5) << endl;
    // calls double add(double a, double b)
    return 0;
}
```


JavaScript

```javascript
function add(a, b) {
    if (typeof a === 'number' && typeof b === 'number') {
        return a + b;
    } else if (typeof a === 'string' && typeof b === 'string') {
        return a.concat(b);
    } else {
        return null;
    }
}

console.log(add(5, 10));
// returns 15
console.log(add('hello', 'world'));
// returns 'helloworld'
```

Author: Muhammad Jahanzaib

Function overloading - Interview Questions

1. What is function overloading?
2. Can you explain the concept of static polymorphism using function overloading?
3. How is function overloading different from function overriding?
4. How do you decide which overloaded function to call during runtime?
5. Can you provide an example of function overloading in C++?

Author: Muhammad Jahanzaib

Interfaces & abstract classes

Definition:

Interfaces and abstract classes are used to define common behavior for a group of classes.
An interface is a collection of abstract methods that a class can implement,
while an abstract class is a class that cannot be instantiated and has one or more abstract methods.

Key points:

- An interface is a contract that a class agrees to follow, and it only contains abstract methods.
- An abstract class can contain both abstract and concrete methods, and it provides a base implementation that derived classes can extend or override.
- An interface allows multiple inheritance, while a class can only inherit from one class (abstract or not).
- Abstract classes are useful when we want to define a base class that should not be instantiated but provides a common behavior for its derived classes.
- .Interfaces are useful when we want to define a common behavior that can be implemented by multiple classes that are not necessarily related by inheritance.

Interfaces & abstract classes

Real world example:

A real-world example of an interface can be a payment gateway. An interface can define a set of methods that a payment gateway should implement, such as **processPayment**, **refundPayment**, and **getTransactionStatus**. Different payment gateways can implement this interface in their own way, but they should all provide these methods.

An abstract class can be used to define a base class for different types of vehicles, such as cars, bikes, and trucks.

The abstract class can provide common functionality such as **startEngine, stopEngine,** and **changeGear**, while the derived classes can implement their own specific behavior.

Author: Muhammad Jahanzaib

Interfaces & abstract classes

Code Examples:

C++

```cpp
class Drawable {
public:
    virtual void draw() = 0;
};

class Circle : public Drawable {
protected:
    double radius;

public:
    Circle(double r) : radius(r) {}
    void draw() override { cout << "Drawing circle with radius " << radius << endl; }
};

class Square : public Drawable {
protected:
    double side;

public:
    Square(double s) : side(s) {}
    void draw() override { cout << "Drawing square with side " << side << endl; }
};

void drawShapes(vector<Drawable*>& shapes) {
    for (auto shape : shapes) {
        shape->draw();
    }
}

int main() {
    Circle c(5);
    Square s(10);
    vector<Drawable*> shapes = { &c, &s };
    drawShapes(shapes);
    return 0;
}
```


JavaScript

```javascript
class Drawable {
    draw() {}
}

class Circle extends Drawable {
    constructor(radius) {
        super();
        this.radius = radius;
    }

    draw() {
        console.log("Drawing circle with radius " + this.radius);
    }
}

class Square extends Drawable {
    constructor(side) {
        super();
        this.side = side;
    }

    draw() {
        console.log("Drawing square with side " + this.side);
    }
}

function drawShapes(shapes) {
    for (let shape of shapes) {
        shape.draw();
    }
}

let c = new Circle(5);
let s = new Square(10);
let shapes = [ c, s ];
drawShapes(shapes);
```

Author: Muhammad Jahanzaib

Interfaces & abstract classes - Interview Questions

1. What is an abstract class?
2. How is an abstract class different from a concrete class?
3. Can you provide an example of an abstract class in C++?
4. What is the difference between an abstract class and an interface?
5. How do you implement multiple inheritance using abstract classes and interfaces?

Author: Muhammad Jahanzaib

Design patterns

Definition:

Design patterns are general solutions to common problems in software design. They are reusable templates that can help us solve similar problems in different contexts. Design patterns can be classified into three categories: creational patterns, structural patterns, and behavioral patterns.

Types of design patterns

- **Creational patterns** deal with object creation mechanisms and aim to abstract the object creation process.
- **Structural patterns** deal with the composition of classes and objects and focus on creating larger structures from individual objects..
- **Behavioral patterns** focus on communication between objects and the assignment of responsibilities.

Author: Muhammad Jahanzaib

Design patterns

Real world example:

A real-world example of a design pattern can be the Singleton pattern. This pattern is used when we need to ensure that there is only one instance of a class, and this instance should be globally accessible. A good example of this is a database connection. We want to make sure that there is only one connection to the database, and all other classes in the system should use this connection.

Author: Muhammad Jahanzaib

Design patterns - Creational Patterns

Definition:

Creational patterns deal with object creation mechanisms and aim to abstract the object creation process.

Types of design patterns

- **Singleton**: Ensures only one instance of a class is created and provides a global point of access to it.
- **Factory Method**: Creates objects without specifying the exact class to create.
- **Abstract Factory**: Creates families of related objects without specifying the concrete classes.
- **Builder**: Separates the construction of a complex object from its representation, allowing the same construction process to create different representations.
- **Prototype**: Creates new objects by cloning an existing object.

Author: Muhammad Jahanzaib

Singleton

Design patterns - Creational Patterns

Ensures that only one instance of a class is created and provides a global point of access to it. This pattern is useful when you want to limit the number of instances of a class, or when you need to ensure that all objects in a system can access a common resource. For example, a logging class can be implemented as a singleton because you only need one instance of it to handle all logging in the system.

Author: Muhammad Jahanzaib

Factory Method

Design patterns - Creational Patterns

Creates objects without specifying the exact class to create. This pattern is useful when you want to decouple the creation of objects from their implementation. For example, a payment system can have a factory method that creates payment objects based on the payment method selected by the user.

```cpp
class Payment {
public:
    virtual void pay() = 0;
};

class CreditCardPayment : public Payment {
public:
    void pay() {
        cout << "Paying with credit card" << endl;
    }
};

class PayPalPayment : public Payment {
public:
    void pay() {
        cout << "Paying with PayPal" << endl;
    }
};

class PaymentFactory {
public:
    virtual Payment* createPayment() = 0;
};

class CreditCardPaymentFactory : public PaymentFactory {
public:
    Payment* createPayment() {
        return new CreditCardPayment();
    }
};

class PayPalPaymentFactory : public PaymentFactory {
public:
    Payment* createPayment() {
        return new PayPalPayment();
    }
};

int main() {
    PaymentFactory* factory = new CreditCardPaymentFactory();
    Payment* payment = factory->createPayment();
    payment->pay();

    delete payment;
    delete factory;

    return 0;
}
```

Author: Muhammad Jahanzaib

Design patterns - Structural Patterns

Definition:

Structural patterns deal with the composition of classes and objects and focus on creating larger structures from individual objects.

Types of design patterns

- **Adapter:** Converts the interface of a class into another interface that clients expect.
- **Bridge:** Decouples an abstraction from its implementation, allowing them to vary independently.
- **Composite:** Composes objects into tree structures to represent part-whole hierarchies.
- **Decorator:** Dynamically adds responsibilities to objects without changing their structure.
- **Facade:** Provides a unified interface to a set of interfaces in a subsystem.
- Flyweight: Shares objects to support large numbers of fine-grained objects efficiently.
- Proxy: Provides a placeholder for another object to control access, reduce complexity, or enhance functionality.

Author: Muhammad Jahanzaib

Design patterns - Behavioral Patterns

Definition:

Behavioral patterns focus on communication between objects and the assignment of responsibilities.

Types of design patterns

- **Chain of Responsibility:** Avoids coupling the sender of a request to its receiver by giving multiple objects the chance to handle the request.
- **Command:** Encapsulates a request as an object, allowing it to be parameterized, queued, logged, and undoable.
- **Interpreter:** Defines a representation for grammar along with an interpreter to interpret the grammar.
- **Iterator:** Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- **Observer:** Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Author: Muhammad Jahanzaib

Design patterns - Interview Questions

1. What are design patterns and why are they important?
2. Can you give an example of a creational design pattern and explain how it works?
3. Can you give an example of a structural design pattern and explain how it works?
4. Can you give an example of a behavioral design pattern and explain how it works?
5. What is the Singleton design pattern and when would you use it?
6. What is the Observer design pattern and when would you use it?
7. What is the Decorator design pattern and when would you use it?
8. What is the Adapter design pattern and when would you use it?
9. How do design patterns relate to the SOLID principles?
10. Can you explain the difference between the Factory Method and Abstract Factory design patterns?

Author: Muhammad Jahanzaib

Object-oriented analysis and design (OOAD)

Definition:

Object-oriented analysis and design (OOAD) is a software engineering approach that involves identifying objects in a system, analyzing their relationships, and designing a system that meets the requirements of its stakeholders.

Key Points:

1. Requirements Gathering
2. Analysis
3. Design
4. Implementation
5. Testing
6. Maintenance

Author: Muhammad Jahanzaib

OOAD - Requirements Gathering

Definition:

This is the first step in OOAD, which involves identifying the requirements of the system from the stakeholders. This is done through various techniques like interviews, surveys, questionnaires, etc.

Example:

The stakeholders of the banking system are identified, and their requirements are gathered through interviews and surveys. The requirements include the ability to open new accounts, deposit and withdraw money, view account statements, and transfer money between accounts.

Author: Muhammad Jahanzaib

OOAD - Analysis

Definition:

In this step, the requirements gathered in the previous step are analyzed to identify the objects in the system, their properties, and relationships. Use cases, activity diagrams, and sequence diagrams are used to represent the analysis.

Example:

The requirements gathered in the previous step are analyzed to identify the objects in the system, including customer, account, transaction, and bank. The properties and relationships of these objects are identified, and use cases, activity diagrams, and sequence diagrams are created to represent the analysis.

Author: Muhammad Jahanzaib

OOAD - Design

Definition:

In this step, the analysis is used to design the architecture of the system, including the classes, interfaces, and their relationships. Class diagrams, object diagrams, and state diagrams are used to represent the design.

Example:

The analysis is used to design the architecture of the banking system, including the classes, interfaces, and their relationships. Class diagrams, object diagrams, and state diagrams are created to represent the design.

Author: Muhammad Jahanzaib

OOAD - Implementation

Definition:

This is the stage where the design is translated into code, typically using an object-oriented programming language like C++, Java, or Python.

Example:

The design is translated into code, using an object-oriented programming language like C++ or Java.

Author: Muhammad Jahanzaib

OOAD - Testing

Definition:

In this stage, the software is tested to ensure that it meets the requirements of its stakeholders. Testing includes various techniques like unit testing, integration testing, and system testing.

Example:

The banking system is tested to ensure that it meets the requirements of its stakeholders. Testing includes unit testing, integration testing, and system testing.

Author: Muhammad Jahanzaib

OOAD - Maintenance

Definition:

Once the software is deployed, it enters the maintenance stage, where it is maintained and updated to meet changing requirements and fix any bugs that are discovered.

Example:

Once the banking system is deployed, it enters the maintenance stage, where it is maintained and updated to meet changing requirements and fix any bugs that are discovered.

Author: Muhammad Jahanzaib

Design patterns - Interview Questions

1. What is Object-Oriented Analysis and Design (OOAD)?
2. What are the key concepts of OOAD?
3. What is the difference between OOAD and OOP?
4. Can you describe the OOAD process?
5. What are the benefits of using OOAD in software development?
6. What are some common tools and techniques used in OOAD?
7. Can you explain the difference between use case and activity diagrams?
8. How do you handle non-functional requirements in OOAD?
9. Can you discuss the SOLID principles and how they relate to OOAD?
10. How do you validate and test an OOAD model?